Лекция 1. Введение в формальные методы верификации программ

Большая формальность подходит для проектов с большой критичностью.

А. Коберн.
Быстрая разработка программного обеспечения

Переход от неформального к формальному существенно неформален.

М.Р. Шура-Бура

Дается представление о корректности программ, т.е. их соответствии требованиям, и верификации — процессе проверки корректности. Делается обзор базовых методов верификации, включая как формальные (проверка эквивалентности, проверка моделей, дедуктивный анализ), так и неформальные (экспертиза, тестирование). Рассматривается общая схема формальной верификации: программе ставится в соответствие формальная модель программы, требованиям — формальная модель требований; проверка корректности программы сводится к формальной процедуре, устанавливающей соответствие между первой и второй моделями. Приводятся некоторые типы моделей и отношений соответствия.

1.1. Ошибки в программах

Программистский фольклор гласит, что в каждой программе (даже тщательно отлаженной) есть хотя бы одна ошибка[1]. Эта шутка, как и всякая другая, появилась не на пустом месте: при высокой сложности компьютерных систем (миллиарды транзисторов и миллионы строк кода) избежать ошибок — несоответствий системы требованиям — невозможно. Заметим, что речь идет не только о прикладном «софте», разрабатываемом в спешке не всегда компетентными специалистами, но и ответственных компонентах, включая аппаратуру (микропроцессоры, устройства ввода-вывода) и системное ПО (операционные системы, компиляторы). Очевидно, что системы, в проектировании и реализации которых допущены ошибки, могут в той или иной ситуации повести себя непредсказуемым образом и привести к серьезным последствиям. В литературе описано множество примеров такого рода (см., например, [Адж98] и [Кар10]). Вот некоторые из них.

[1] Это высказывание известно как первая аксиома М.Р. Шуры-Буры (1918-2008). Полностью эта шутливая «аксиоматика» выглядит так:

Аксиома 1. В каждой программе есть хотя бы одна ошибка.

Аксиома 2. Если ошибки нет в программе, она есть в реализуемом алгоритме.

Аксиома 3. Если ошибки нет и в алгоритме, такая программа никому не нужна.

- В 1982 г. канадской фирмой Atomic Energy of Canada Limited был запущен в серию медицинский аппарат «Therac-25», предназначенный для лучевой терапии — облучения раковой опухоли. В «редких» ситуациях из-за ряда ошибок, допущенных при проектировании и реализации встроенной системы управления, интенсивность облучения возрастала на 2 порядка и более. За время эксплуатации прибора (период с июня 1985 г. по январь 1987 г.) как минимум два пациента умерли, а несколько человек остались инвалидами[2].

- 2 сентября 1988 г. была потеряна связь с советской межпланетной станцией «Фобос-1», запущенной 7 июля для исследования Марса и его спутника Фобоса. 29 августа с Земли была передана некорректная команда, которая была ошибочно воспринята бортовой системой как команда на отключение системы ориентации и стабилизации. Станция перестала ориентировать солнечные батареи на Солнце, вследствие чего ее аккумуляторы истощились. Через полгода была потеряна связь с «Фобос-2», дублером «Фобос-1». Основная задача обеих миссий — доставка на поверхность Фобоса долгоживущей автономной станции — осталась невыполненной[3].

- 13 июня 1994 г. была обнаружена ошибка в реализации инструкции деления чисел с плавающей точкой в микропроцессоре Pentium фирмы Intel (в таблице начальных приближений модуля деления присутствовало несколько некорректных значений). И хотя рядовых пользователей персональных компьютеров эта ошибка, казалось бы, не должна была беспокоить, компании, для сохранения имиджа, пришлось осуществить бесплатную замену выпущенных микросхем. Затраты на это составили 475 миллионов долларов.

- 4 июня 1996 г. на 39-й секунде после старта взорвалась ракета-носитель «Ариан 5», разработанная Европейским космическим агентством. Бортовая система частично

---

[2] Случай с аппаратом «Therac-25» не является единственным в своем роде — в 1991 г. в Испании при использовании аппарата «Sagitar-35» подверглись передозировке не менее 25 пациентов, из которых как минимум трое умерли.

[3] Неудачная участь постигла и межпланетную станцию «Фобос-Грунт», запущенную 9 ноября 2011 г., — из-за нештатной ситуации станция не смогла покинуть окрестности Земли и 15 января 2012 г. сгорела в плотных слоях атмосферы (предположительно, некоторые фрагменты станции затонули в Тихом океане).

использовала ПО предыдущей версии ракеты, «Ариан 4», в частности модуль управления инерциальной навигационной системой[4]. При выполнении преобразования из 64-битного числа с плавающей точкой, представляющего наклон ракеты, в 16-битное целое возникло переполнение, которое не было обработано модулем. При разработке модуля предполагалось, что переполнение невозможно из-за физических ограничений ракеты, но прогресс, как известно, не стоит на месте — в «Ариан 5» использовались новые двигатели.

- 23 марта 2003 г. во время войны в Ираке американский зенитно-ракетный комплекс «Patriot» ошибочно идентифицировал британский бомбардировщик «Tornado», летевший близ кувейтской границы, как приближающуюся вражескую ракету и автоматически произвел залп. Погибли два пилота. Спустя полторы недели, 2 апреля, «дружественным огнем» был уничтожен американский палубный истребитель-бомбардировщик F/A-18 «Hornet». Еще один пилот погиб.

Важно понимать, что ошибки в компьютерных системах не являются чем-то исключительным. Согласно статистике, среднее число ошибок на тысячу строк неотлаженного кода колеблется в пределах 10-50[5] [Mcc04]. Более того, наблюдается тенденция к деградации качества проектирования (по всей видимости, это следствие возрастающей сложности систем и оптимизации расходов на их создание): «С точки зрения способности все запутать и испортить компьютер с каждым днем становится все больше похож на человека» [Llo07]. Вместе с тем наша жизнь все чаще зависит от компьютеров, поэтому обеспечение корректности и надежности компьютерных систем (другими словами, верификация) приобретает все бóльшее значение.

1.2. Верификация программ

Что же такое *верификация*? Верификацией называется проверка соответствия программы (или некоторого промежуточного результата проектирования: прототипа, модели и т.п.)

[4] Кстати говоря, ПО в Европейском космическом агентстве разрабатывается на языке программирования Ada, который задумывался как язык создания надежных программ.

[5] Очевидно, что ошибки распределены по программе неравномерно; более того, они имеют тенденцию образовывать скопления [Gla02]. Согласно статистике, приведенной в [Boe01], примерно 80% ошибок находятся в 20% модулей.

предъявляемым к ней требованиям (проектной документации). Если программа соответствует требованиям, она называется *корректной* (*правильной*); в противном случае программа называется *некорректной* (*ошибочной*), а сам факт несоответствия программы требованиям — *ошибкой*[6]. Таким образом, верификация — это анализ программы на предмет наличия или отсутствия в ней ошибок. Говоря утрированно, результатом верификации является *вердикт*: *отрицательный*, если в программе найдена хотя бы одна ошибка, или *положительный*, если доказано, что ошибок нет. Возможен также *неопределенный* вердикт, когда ошибки не найдены, но их отсутствие не доказано (это распространенная ситуация).

Выше было дано формальное определение — на практике же необходимо учитывать множество нюансов. Во-первых, требования, как правило, формулируются неформально, на естественном языке, поэтому не всегда можно однозначно определить, соответствует им программа или нет. Во-вторых, даже если требования формализованы, доказать отсутствие ошибок в программе крайне трудно с математической точки зрения (эта работа не вполне поддается автоматизации). В большинстве проектов задача доказательства корректности или некорректности ПО не ставится[7]. Типичной целью является разработка качественного продукта, где под «качеством» понимается сложная и расплывчатая характеристика[8].

Методы верификации можно разделить на три основные группы (более подробная классификация приводится, например, в обзоре [Кул08]):

1. *формальные методы*, использующие математически строгий анализ модели программы и модели требований;
2. *методы тестирования*, осуществляющие проверку реального поведения программы на некотором наборе сценариев;
3. *экспертизу*, выполняемую людьми на основе их знаний и опыта непосредственно над результатами проектирования (например, *инспекция кода*).

---

[6] Слово «ошибка» может пониматься по-разному: и как несоответствие программы требованиям — отклонение поведения программы от ожидаемого (failure), и как причина несоответствия — ошибка в коде (fault). Обычно значение понятно из контекста.

[7] В этом нет ничего удивительного, принимая во внимание первую аксиому М.Р. Шуры-Буры.

[8] Обычно под качеством ПО понимают совокупность свойств, включающую переносимость, надежность, эффективность, удобство в использовании, тестируемость, понятность и модифицируемость [Gla02].

Каждая из указанных групп методов имеет свои достоинства и недостатки, у каждой — своя область применимости. Полноценная верификация сложных систем ответственного назначения невозможна без совместного использования разных подходов. Методы верификации, совмещающие различные техники, называются *гибридными* или *синтетическими* [Кул08]. Примеры таких методов, а именно методы тестирования, использующие техники формальной верификации, рассматриваются в лекции 15.

Пособие, которое вы держите в руках, посвящено формальным методам верификации программ. Безусловно, эти методы не покрывают всех потребностей индустрии разработки ПО, но как математика является языком науки, так и формальные методы являются языком программной инженерии. Приведем слова Джона Маккарти (John McCarthy, 1927-2011) — выдающегося американского информатика, автора языка Lisp, лауреата премии Тьюринга: «Разумно ожидать, что связи между вычислительной техникой и математической логикой окажутся столь же плодотворными в следующем столетии [XXI в.], какими были связи между математическим анализом и физикой в столетии предыдущем [XIX в.]».

1.3. Формальная верификация программ

Формальная верификация основывается на математическом (логическом) моделировании программ и требований к ним. Идея здесь в точности такая, как при использовании моделей в других областях знаний: создается модель — идеализированное описание исследуемого объекта или явления; модель исследуется с применением математических методов; результаты исследования переносятся на реальный объект или явление. Безусловно, применимость такого подхода определяется используемыми моделями — нужно четко понимать условия их адекватности.

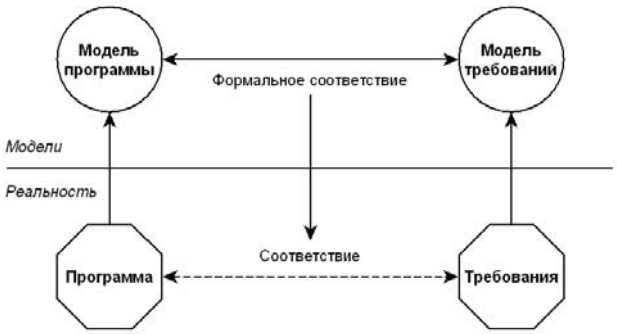


Рисунок 1.1. Общая схема формальной верификации

Общая схема формальной верификации показана на рис. 1.1: создается формальная модель программы; создается формальная модель требований; формально проверяется соответствие модели программы модели требований; на основании результатов проверки делается вывод о соответствии или несоответствии реальной программы реальным требованиям (другими словами, об отсутствии или наличии ошибок в программе). Для представления моделей программ и моделей требований используются соответственно языки формальной спецификации программ (языки моделирования) и языки формальной спецификации требований; примеры таких языков будут рассмотрены в пособии.

Методы формальной верификации различаются типами моделей программ (конечные автоматы, сети Петри, размеченные системы переходов), типами моделей требований (программные контракты, продукционные правила, темпоральные утверждения), отношениями соответствия (эквивалентность, симуляция, включение трасс) и техниками проверки соответствия (исследование пространства состояний, символический анализ, дедуктивный вывод).

Ниже мы рассмотрим подробнее такие разновидности методов формальной верификации, как проверку эквивалентности, проверку моделей и дедуктивный анализ.

1.3.1. Проверка эквивалентности

В *проверке эквивалентности* модель программы и модель требований однотипны (например, обе модели — конечные автоматы, или обе модели — машины Тьюринга), а в качестве отношения соответствия используется одно из отношений эквивалентности, определенное для моделей рассматриваемого типа. При проверке эквивалентности исполнимых моделей (автоматов, программ и т.п.) модель требований называют *эталонной моделью* или *эталонной реализацией*.

Проиллюстрируем подход сначала на примере конечных автоматов, а затем на примере последовательных программ.

Конечным автоматом Мили, или просто *автоматом*, называется шестерка $\langle S, X, Y, s_0, \delta, \lambda \rangle$, в которой S — множество состояний, X — входной алфавит (множество стимулов), Y — выходной алфавит (множество реакций), $s_0 \in S$ — начальное состояние, $\delta: S \times X \to S$ — функция переходов, $\lambda: S \times X \to Y$ — функция выходов (все упомянутые множества полагаются конечными и непустыми).

Семантика автомата определяется отображением цепочек символов входного алфавита в цепочки символов выходного алфавита. Два автомата над общими входным и выходным алфавитами называются *эквивалентными*, если соответствующие им отображения совпадают.

Пример 1.1

Рассмотрим два автомата A и B (их диаграммы состояний показаны на рис. 1.2: начальные состояния отмечены стрелками, ведущими в них из «ниоткуда»):

$A = \langle\{0,1\},\{0,1\},\{0,1\},0,\delta_A,\lambda_A\rangle$, где

$$\delta_A(s,x) = x, \text{ для всех } x,s \in \{0,1\}; \qquad \lambda_A(s,x) = s, \text{ для всех } x,s \in \{0,1\};$$

$B = \langle\{a,b,c,d\},\{0,1\},\{0,1\},a,\delta_B,\lambda_B\rangle$, где

$$\delta_B(s,x) = \begin{cases} a, & \text{если } s \in \{b,d\} \wedge x = 0; \\ b, & \text{если } s \in \{a,c\} \wedge x = 0; \\ c, & \text{если } s \in \{a,d\} \wedge x = 1; \\ d, & \text{если } s \in \{b,c\} \wedge x = 1; \end{cases} \quad \lambda_B(s,x) = \begin{cases} 0, & \text{если } s \in \{a,b\}; \\ 1, & \text{если } s \in \{c,d\}. \end{cases}$$

Очевидно, что автоматы A и B эквивалентны — это две разные реализации элемента единичной задержки — D-триггера (оба автомата выдают предыдущий стимул вне зависимости от текущего; начальная реакция — 0).

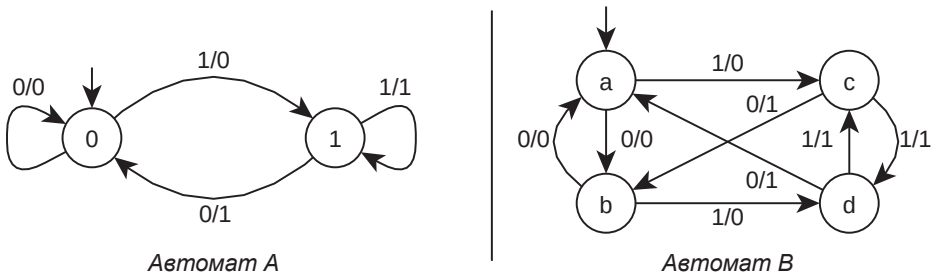


Рисунок 1.2. Пример эквивалентных конечных автоматов

□

Каким образом можно установить эквивалентность или неэквивалентность двух конечных автоматов? Во-первых, это можно сделать с помощью методов тестирования. Основной результат в этой области — теорема Мура о различении состояний автоматов, определяющая

ограничение на длину теста (различающего эксперимента) в зависимости от числа состояний сравниваемых автоматов[9] (эта теорема рассматривается во вводном курсе дискретной математики: см., например, [Але13]). Заметим, что тестирование применимо, даже когда автоматы являются «черными ящиками», т.е. когда их структура не известна. Другой подход — статический анализ отношений переходов. На этот счет есть другая теорема Мура — теорема об эквивалентности автоматов [Кар03]: два конечных автомата A и B эквивалентны тогда и только тогда, когда в любом достижимом состоянии их прямого произведения (s_A, s_B) для всех стимулов x выполнено равенство $\lambda_A(s_A, x) = \lambda_B(s_B, x)$. Понятие прямого произведения автоматов определяется в лекции 12 в контексте теоретико-автоматного подхода к проверке моделей.

Перейдем теперь к более сложным объектам — программам. Две (детерминированные) последовательные программы с одинаковыми наборами входных и выходных переменных называются (функционально) *эквивалентными*, если их области определения совпадают (любые входные данные, допустимые для одной программы, являются допустимыми и для другой) и для любых допустимых входных данных они возвращают одинаковые результаты.

Пример 1.2

Ниже в таблице 1.1 приведены две программы P и Q, каждая из которых имеет две входные целочисленные переменные a и b, причем $a > 0$ и $b > 0$, и одну выходную целочисленную переменную c:

Таблица 1.1. Две реализации алгоритма Евклида

Программа P	Программа Q

`x := a;`
`y := b;`
`while` `x ≠ y` **`do`**
  `if` `x > y` **`then`**
  `x := x − y`
	`x := a;`
	`y := b;`
	`while` `y ≠ 0` **`do`**
	  `z := x;`
	  `x := y;`
	  `y := z % y`

[9] Теорема Мура (о различении состояний автоматов) утверждает, что для любых двух конечных автоматов и любых двух различимых состояний этих автоматов (в частности, начальных, если автоматы не эквивалентны) существует различающий эксперимент длины не большей $n + m - 1$, где n и m — мощности множеств состояний данных автоматов; другими словами, существует последовательность стимулов длины, не превосходящей $n + m - 1$, применение которой к данным автоматам в данных состояниях приводит к разным последовательностям реакций.

```
  else
    y := y - x
  end
end;
c := x
```

```
end;
c := x
```

Нетрудно видеть, что программы P и Q эквивалентны — они обе вычисляют наибольший общий делитель (НОД) двух натуральных чисел (это разные реализации алгоритма Евклида). Как формально доказать их эквивалентность?

Можно показать, следуя нашей догадке, что обе программы вычисляют $\text{НОД}(a, b)$ и записывают результат в переменную c, т.е. для каждой из них справедливо следующее соотношение между входными и выходными переменными:

$$\exists m, n \in \mathbb{N}\big((a = c \cdot m) \wedge (b = c \cdot n)\big) \wedge$$
$$\forall c', m', n' \in \mathbb{N}\Big(\big((a = c' \cdot m') \wedge (b = c' \cdot n')\big) \rightarrow (c' \leq c)\Big).$$

Как это делается, рассматривается в лекциях 3-6, посвященных дедуктивной верификации. Однако, говоря об автоматической проверке, все же неясно, как заставить компьютер «понять», что делает программа, какой у нее «смысл». Это фундаментальная проблема, далеко выходящая за рамки пособия.

Можно пойти другим путем — не вникая в смысл сравниваемых программ, попытаться преобразовать одну к другой (или каждую к какой-то третьей), применяя эквивалентные преобразования. Преобразуем, например, Q к P:

1. Заменим присваивание $y := z \,\%\, y$ на $y := z \,\%\, x$. Очевидно, это можно сделать, поскольку $x = y$.
2. Присваивание $y := z \,\%\, x$ заменим на цикл $y := z$; while $y \geq x$ do $y := y - x$ end. Здесь операция взятия остатка от деления реализована через вычитание. Получим:

```
...
while y ≠ 0 do
  z := x; x := y; y := z;
  while y ≥ x do y := y - x end
end;
...
```

3. Нестрогое сравнение $y \geq x$ во внутреннем цикле заменим на строгое $y > x$, соответственно изменив сравнение во внешнем цикле с $y \neq 0$ на $y \neq x$ или, что равносильно, на $x \neq y$.

4. Далее выполним менее очевидное преобразование. Заметим, что код перед внутренним циклом меняет местами значения переменных x и y — внутренний цикл поочередно применяется то для пары (x, y), то для (y, x). Удалим последовательность присваиваний $z := x; x := y; y := z$, а вместо нее добавим внутренний цикл, идентичный существующему, но в котором переменная x заменена на y, а y на x (на каждой итерации внешнего цикла запускается только один внутренний):

```
...
while x ≠ y do
  while x > y do x := x - y end;
  while y > x do y := y - x end
end;
...
```

5. Внутренние циклы while заменим на условные операторы $\text{if} - \text{then}$.

6. Внешнее условие $x \neq y$ позволяет объединить операторы $\text{if} - \text{then}$ с условиями $x > y$ и $y > x$ в один оператор $\text{if} - \text{then} - \text{else}$. В результате получается требуемая программа P:

```
...
while x ≠ y do
  if x > y then
    x := x - y
  else
    y := y - x
  end
end;
...
```

□

Несмотря на то, что для выполнения таких преобразований не требуется вникать в суть программ (в преобразованиях учитываются лишь локальные свойства), приведение программ друг к другу с трудом поддается автоматизации. Это следствие алгоритмической неразрешимости задачи проверки функциональной эквивалентности программ. Практические исследования в этой области связаны с применением более строгих отношений эквивалентности, так называемых *аппроксимирующих отношений*, а также с сужением рассматрива-

емых классов программ. Тема проверки эквивалентности выходит за рамки пособия; читателям, заинтересовавшихся этой проблематикой, мы рекомендуем монографию [Кот91] и диссертацию [Зах12].

1.3.2. Проверка модели

В *проверке моделей* требования к программам представляются логическими формулами определенного вида, а сами программы — структурами, интерпретирующими формулы этого вида; отношение соответствия — истинность формулы на структуре (программа удовлетворяет требованиям тогда и только тогда, когда соответствующая структура являются моделью[10] соответствующей формулы).

Часто для представления требований используются *темпоральные*, или *временны́е логики*, — логики, позволяющие задать взаимосвязи событий во времени, например, логика линейного времени LTL (Linear-time Temporal Logic) или логика ветвящегося времени CTL (Computational Tree Logic). Соответственно для представления программ используются структуры Крипке и родственные им формализмы (размеченные системы переходов).

Пусть задано множество элементарных высказываний AP (*Atomic Propositions*). *Структурой Крипке* над множеством AP называется четверка $\langle S, S_0, R, L\rangle$, где S — множество состояний, $S_0 \subseteq S$ — множество начальных состояний, $R \subseteq S \times S$ — отношение переходов, $L: S \to 2^{AP}$ — функция разметки, помечающая каждое состояние множеством истинных в этом состоянии высказываний [Кар10].

Структуры Крипке используются для моделирования *реагирующих систем* — систем, работающих в «бесконечном цикле» и взаимодействующих со своим окружением. Поведение системы моделируется *вычислением в структуре Крипке* — бесконечной последовательностью состояний, первое из которых принадлежит множеству S_0, а каждая пара последовательных состояний входит в отношение R.

Для описания требований может использоваться пропозициональная темпоральная логика, в частности LTL. Помимо обычных связок в этой логике используются темпоральные

[10] Слово «модель» здесь используется в логическом понимании: моделью формулы называется всякая интерпретация, в которой эта формула истинна.

операторы **G** (*Globally*), **F** (*in the Future*), **X** (*neXt time*) и **U** (*Until*), которые интерпретируются следующим образом:

- $\mathbf{G}\varphi$ — условие φ истинно *всегда*;
- $\mathbf{F}\varphi$ — условие φ истинно *сейчас* или станет истинным *когда-нибудь в будущем*;
- $\mathbf{X}\varphi$ — условие φ истинно *в следующий момент времени*;
- $\varphi\ \mathbf{U}\ \psi$ — условие φ истинно *до тех пор, пока* не станет истинным условие ψ.

Полноценное определение синтаксиса и семантики логики LTL будет дано в лекции 9.

Пример 1.3

Пусть множество элементарных высказываний имеет вид $\{locked, lock, unlock\}$. Рассмотрим структуру Крипке $M = \langle\{s_0, s_1, s_2, s_3\}, \{s_0\}, R, L\rangle$, моделирующую дверь с замко́м (см. рис. 1.3: начальное состояние помечено входящей стрелкой; рядом с каждым состоянием указано множество истинных в нем высказываний). Состояния структуры размечены следующим образом:

- $L(s_0) = \emptyset$ — дверь открыта; с замком не производится никаких действий;
- $L(s_1) = \{lock\}$ — дверь открыта; ее закрывают;
- $L(s_2) = \{locked\}$ — дверь закрыта, с замком не производится никаких действий;
- $L(s_3) = \{locked, unlock\}$ — дверь закрыта; ее открывают.

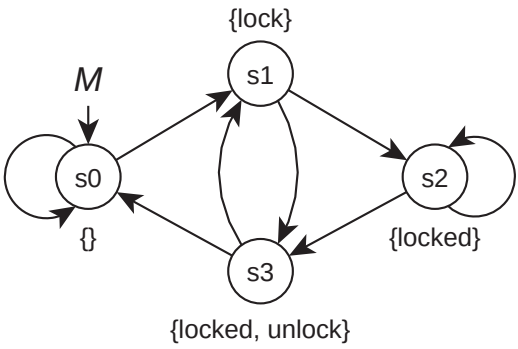


Рисунок 1.3. Структура Крипке, моделирующая дверной замок

Ниже приведены примеры требований к системе, выраженные в логике LTL:

- $\mathbf{G}\{\neg(lock \wedge unlock)\}$ — *никогда* нельзя одновременно и открывать и закрывать дверь;
- $\mathbf{G}\{\neg(locked \wedge lock)\}$ — *никогда* нельзя закрывать закрытую дверь;
- $\mathbf{G}\{lock \rightarrow \mathrm{X}(locked)\}$ — *всегда* если дверь закрывают, в *следующий момент времени* она станет закрытой.

□

Теме проверке моделей, а именно теоретику-автоматному подходу к проверке моделей для логики LTL, будут посвящены лекции 11-13. Пока лишь отметим, что основным методом установления корректности или некорректности модели является поиск в пространстве состояний (обход графа состояний). В лекции 14 рассматривается более современный класс методов — символическая проверка моделей.

1.3.3. Дедуктивный анализ

В последнем из рассматриваемых подходов, *дедуктивном анализе*, моделью программы является код на языке с формализованной семантикой (сама программа, если язык программирования формализован, псевдокод или блок-схема), а моделью требований — *программный контракт*, т.е. пара логических формул: *пред-* и *постусловие*; отношение соответствия — *полная* или *частичная корректность* программы относительно контракта [And79], [Буз14].

Для удобства будем считать, что переменные программы делятся на три типа: *входные*, *внутренние* и *выходные*: входные переменные содержат исходные данные и не меняются во время исполнения программы; внутренние переменные используются для хранения промежуточных результатов вычислений; выходные переменные предназначены для выдачи полученных результатов.

Говорят, что программа P является *частично корректной* относительно предусловия φ и постусловия ψ (обозначается: $\{\varphi\}P\{\psi\}$), если для всех значений входных переменных, для которых истинно условие φ, в случае завершения P для значений выходных переменных истинно условие ψ.

Говорят, что программа P является *полностью корректной* относительно предусловия φ и постусловия ψ (обозначается: $\langle\varphi\rangle P\langle\psi\rangle$ или $\{\varphi\}P\{\psi\}$, если из контекста ясно, что речь идет

о полной корректности), если для всех значений входных переменных, для которых истинно условие φ, P завершается и для значений выходных переменных истинно условие ψ.

Пример 1.4

Ниже приведена программа DIV, имеющая две входные целочисленные переменные, a и b, и две выходные целочисленные переменные, q и r. Для программы задано предусловие

$$\varphi \equiv (a \geq 0) \wedge (b > 0),$$

уточняющее, что программа определена только для неотрицательных значений a и положительных значений b, и постусловие

$$\psi \equiv (a = q \cdot b + r) \wedge (0 \leq r < b),$$

утверждающее, что программа осуществляет деление a на b и сохраняет частное от деления в переменную q, а остаток — в переменную r.

```
{(a ≥ 0) ∧ (b > 0)}
q := 0;
r := a;
while r ≥ b do
  q := q + 1;
  r := r - b
end
{(a = q·b + r) ∧ (0 ≤ r < b)}
```

Справедливо следующее утверждение: программа DIV является полностью корректной относительно предусловия φ и постусловия ψ:

$$\vDash \langle (a \geq 0) \wedge (b > 0) \rangle \; DIV \; \langle (a = q \cdot b + r) \wedge (0 \leq r < b) \rangle^{11}.$$

□

Теме дедуктивной верификации программ (а именно методам Флойда, Хоара и Дейкстры) будут посвящены лекции 3-6. Краеугольным камнем этой группы методов является поиск так называемых инвариантов циклов. Однако обо всем по порядку.

[11] Символ ⊨, как это принято в логике, используется для обозначения того, что стоящая справа от него формула истинна.

1.4. Вопросы и упражнения

1. Определите следующие понятия: «ошибка в программе», «корректность программы», «верификация программы». На каком этапе разработки ПО осуществляется верификация?
2. Перечислите известные вам подходы к верификации программ. В чем достоинства и в чем недостатки каждого из них?
3. Чем формальные методы отличаются от других подходов к верификации программ? Приведите примеры формальных методов.
4. Назовите известные вам формализмы, используемые для моделирования программ и требований к ним. В чем, на ваш взгляд, состоит основное отличие между этими двумя классами формализмов?
5. Определите формально семантику конечного автомата Мили.
6. Докажите, используя теорему Мура об эквивалентности конечных автоматов, эквивалентность двух реализаций D-триггера, приведенных в лекции (автоматов A и B).
7. Докажите доступными вам средствами эквивалентность программы P, приведенной в лекции, следующей программе:

```
x := a;
y := b;
while x ≠ 0 ∧ y ≠ 0 do
  if x > y then
    x := x % y
  else
    y := y % x
  end
end;
c := x + y
```

8. Пополните набор требований к системе, описываемой приведенной в лекции структурой Крипке (структурой M). Докажите, что структура M удовлетворяет сформулированным требованиям (является их моделью).
9. Смоделируйте структурой Крипке систему управления стиральной машиной. Машина имеет бак для белья, через который также подаются моющие средства, клапаны для забора и слива воды, датчик наличия воды, мотор, нагревающий элемент, таймер и панель управления с кнопками «Пуск» и «Останов». Предполагается следующий сценарий

использования машины: открыть дверцу бака, поместить белье и моющие средства в бак, закрыть дверцу, нажать на кнопку «Пуск». Машина открывает клапан для забора воды, набирает воду и закрывает клапан; подогревает воду; заводит таймер и запускает мотор, вращающий бак; при срабатывании таймера сливает воду. Дверца бака блокируется, пока в баке есть вода. Пользователь имеет возможность в любой момент нажать на кнопку «Останов», чтобы принудительно остановить стирку белья и слить воду.

10. Определите как можно более полный набор требований к системе управления стиральной машиной, описанной в предыдущем задании. Выразите эти требования в логике LTL.

11. Определите программный контракт (пред- и постусловие) для программы, вычисляющей с заданной точностью квадратный корень числа (x — входное число, y — результат, ε — точность).

12. Определите программный контракт (пред- и постусловие) для программы сортировки числового массива (x — входной массив, y — результат сортировки, N — размер массива).

13. Докажите доступными вам средствами завершимость приведенной в лекции программы целочисленного деления DIV.

Лекция 2. Формализация семантики языков программирования

Алгоритмы сами по себе суть объекты весьма специфического типа и обладают свойством, нетипичным для математических объектов, а именно семантическим свойством «иметь смысл» ⟨…⟩ Теория, изучающая алгоритмы, может трактоваться как своего рода лингвистика повелительных предложений.

В.А. Успенский, А.Л. Семенов.
Теория алгоритмов: основные открытия и приложения

На примере простого императивного языка — языка while — рассматриваются два метода формализации семантики языков программирования: *операционная семантика* (структурная операционная семантика Плоткина) и *аксиоматическая семантика* (дедуктивная система Хоара и метод Дейкстры, основанный на понятии слабейшего предусловия). Показывается, что программа — код на языке с формализованной семантикой — является математическим объектом, к которому применимы строгие логические рассуждения, в частности о его корректности или некорректности (конечно, при наличии формальных спецификаций требований).

2.1. Язык программирования while

Подходы к формализации семантики языков программирования мы будем рассматривать на примере языка while — простого императивного языка программирования [Apt09]. Сначала сделаем два замечания. Первое — в обычных языках программирования *сигнатура*, т.е. множество *функциональных* и *предикатных символов* (вместе с их арностью[12]), используемых при составлении формул ($0^{(0)}, 1^{(0)}, +^{(2)}, -^{(2)}, =^{(2)}, <^{(2)}, >^{(2)}$), фиксирована; мы же будем считать, что в нашем распоряжении есть любые нужные нам «операции». Второе — в языке while нет средств декларации переменных: множество переменных предполагается заданным. Таким образом, уместнее говорить не об одном языке while, а о семействе языков, параметризованном сигнатурой и множеством переменных.

Синтаксис языка while для заданной сигнатуры и множества переменных определяется грамматикой, представленной в таблице 2.1.

[12] Символы констант — это функциональные символы арности 0.

Таблица 2.1. Грамматика языка программирования while

$P ::=$		
skip	*пустой оператор*	(1)
$\mid x \coloneqq t$	*оператор присваивания*	(2)
$\mid P; P$	*последовательная композиция*	(3)
$\mid$ **if** B **then** P **else** P **end**	*условный оператор*	(4)
$\mid$ **while** B **do** P **end**	*оператор цикла*	(5)

Здесь P — цель грамматики (while-программа); x — переменная; t — терм (или, в программистской терминологии, выражение); B — формула (условие).

Термы и формулы определяются обычным образом:

- переменная есть терм;
- константа (функциональный символ арности 0) есть терм;
- если f — функциональный символ арности $k > 0$, а $t_1, \dots, t_k$ — термы, то $f(t_1, \dots, t_k)$ есть терм;
- логическая константа (предикатный символ арности 0) есть формула;
- если p — предикатный символ арности $k > 0$, а $t_1, \dots, t_k$ — термы, то $p(t_1, \dots, t_k)$ есть формула;
- если φ — формула, то $\neg\varphi$ есть формула;
- если φ и ψ — формулы, то $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ и $(\varphi \to \psi)$ есть формулы.

Никакие другие объекты термами и формулами не являются.

Сокращение: если программа, указанная в ветви **else** условного оператора, является пустой (точнее, состоит из оператора **skip**), то ветвь **else** может быть опущена.

Пример 2.1

Пусть $\Sigma = \langle\{+^{(2)}, -^{(2)}, 0^{(0)}, 1^{(0)}\}, \{\geq^{(2)}\}\rangle$ и $V = \{a, b, q, r\}$. Ниже приведена while-программа над сигнатурой Σ и множеством переменных V, осуществляющая целочисленное деление a на b и сохраняющая частное от деления в переменную q, а остаток — в переменную r (это программа DIV из лекции 1).

```
q := 0;
r := a;
while r ≥ b do
  q := q + 1;
  r := r - b
end
```

□

Прежде чем приступить к основной части лекции, интересно рассмотреть возможные постановки задач, связанные с вопросами семантики (смысла, функциональности) программ:

1. *задача спецификации* — описать желаемую семантику программы (определить требования к программе)[13];
2. *задача синтеза* (*прямой инженерии*) — сконструировать программу с заданной семантикой (разработать программу по требованиям);
3. *задача анализа* (*обратной инженерии*) — вывести семантику заданной программы (восстановить «логику работы» программы);
4. *задача верификации* — проверить, имеет ли заданная программа заданную семантику (доказать или опровергнуть корректность программы).

Нас интересуют задачи спецификации и верификации. Применительно к программе DIV они формулируются следующим образом:

- задача спецификации — описать, что программа DIV осуществляет не что иное, как целочисленное деление;
- задача верификации — проверить, соответствует ли программа DIV спецификации, т.е. осуществляет ли она целочисленное деление.

[13] В этой и других задачах семантика программы и требования к программе, по сути, отождествляются, хотя это не одно и то же. Требования можно понимать как ограничения на семантику программы, выраженные на достаточно высоком уровне абстракции, как правило, декларативно. Семантика же программы может определяться и на низком уровне, например, заданием схемы трансляции языка программирования в промежуточный код, интерпретируемый формализованным исполнителем — абстрактной машиной.

В контексте формальных методов предполагается, что описание (спецификация) и проверка (верификация) осуществляются формально, с применением методов математической логики. Для этого и требования и программа должны допускать формальную интерпретацию.

2.2. Формальная семантика

Интуитивно понятно, что каждая программа (если только она не зацикливается) осуществляет отображение одного состояния, т.е. множества значений переменных, в другое[14]. Таким образом, семантика программы может быть задана с помощью соотношений между значениями входных и выходных переменных. Такой подход, называемый *декларативным*, активно используется при описании требований и выводе свойств программ. Другой способ — сопоставление программе последовательности действий некоторого исполнителя (автомата, машины). Такой подход, называемый *интерпретационным*, широко применяется при разработке компиляторов и интерпретаторов языков программирования[15].

Для формализации семантики языка программирования прежде всего необходимо определить семантику используемых в нем функциональных и предикатных символов. Это может быть сделано заданием *алгебраической системы* — набора аксиом, связывающих символы сигнатуры (такой способ используется в декларативном подходе) — или заданием *процедуры*, позволяющей вычислять значения выражений по значениям переменных (такой способ используется в интерпретационном подходе). В дальнейшем алгебраические системы и вычислительные процедуры, используемые для смыслового наполнения сигнатуры, мы будем называть *предметной областью* (*domain*).

Пример 2.2

Проиллюстрируем понятие алгебраической системы на примере арифметики Пеано (Giuseppe Peano, 1858-1932); сигнатура — $\langle\{+^{(2)},\cdot^{(2)},0^{(0)},1^{(0)}\},\{=^{(2)}\}\rangle$.

---

[14] Здесь рассматриваются исключительно программы преобразования данных.

[15] Можно дать более детальную классификацию семантик (см. примечание Н.Н. Непейводы [Mit96, рус. пер., с. 23]): *интерпретационная семантика* (язык определяется через описание способа исполнения его конструкций), *трансляционная семантика* (один язык определяется через другой путем трансляции), *трансформационная семантика* (язык определяется с помощью правил преобразования его конструкций), *алгебраическая семантика* (язык определяется посредством уравнений над его конструкциями и методов их решения) и *логическая семантика* (язык определяется через аксиомы и правила вывода, сопоставленные его конструкциям).

1. $0 \neq x + 1$;
2. $(x + 1) = (y + 1) \to x = y$;
3. $x + 0 = x$;
4. $(x + y) + 1 = x + (y + 1)$;
5. $x \cdot 0 = 0$;
6. $x \cdot (y + 1) = (x \cdot y) + x$;
7. $\frac{\varphi(0), \varphi(x) \to \varphi(x+1)}{\varphi(x)}$.

Здесь и далее $x \neq y$ есть сокращение от $\neg(x = y)$. Запись $\frac{\varphi_1, \ldots, \varphi_n}{\varphi}$, как это принято в логике, обозначает правило вывода: $\varphi_1, \ldots, \varphi_n$ — посылки, φ — следствие. Предполагается, что переменные x и y, как и предикат φ в схеме индукции, связаны квантором всеобщности.

□

Будем говорить, что для языка программирования $\mathcal{L}$ задана *формальная семантика*, если определено отображение M (от англ. *meaning* — значение, смысл), ставящее в соответствие каждой программе $P \in \mathcal{L}$ и каждому ее состоянию s множество возможных результирующих состояний $M[\![P]\!](s)$:

$$M[\![P]\!]: S \to 2^S,$$

где S — множество всех возможных состояний программы P.

Под *состоянием* программы понимается отображение, сопоставляющее каждой переменной программы ее значение. Отображения такого типа обычно называют *функциями означивания* или просто *означиваниями*.

Для обозначения факта зацикливания программы мы будем использовать специальный символ ω. Так, запись $M[\![P]\!](s) = \{\omega\}$ означает, что программа P, будучи запущенной в состоянии s, зацикливается.

Заметим: в определении отображения $M[\![P]\!]$ говорится не об одном результирующем состоянии, а об их множестве. Это связано с тем, что язык может допускать разные интерпретации одной и той же конструкции, из-за чего возможны разные результаты исполнения программы. Для *детерминированных программ* будем считать, что $M[\![P]\!](s)$ — не множество состояний, а одно состояние (или специальное значение ω).

2.2.1. Операционная семантика

Операционная семантика языка определяет смысл программы как последовательность действий, которые необходимо выполнить заданному исполнителю (микропроцессору, интерпретатору, виртуальной машине). Исполнитель довольно «тупой» и не понимает «Божественного предназначения» [Кро01] исполняемых им программ, но он четко знает, в чем состоит то или иное действие.

Рассмотрим так называемую *структурную операционную семантику* — подход, предложенный в 1981 г. Гордоном Плоткиным (Gordon Plotkin, род. в 1946 г.) [Plo81].

Операционная семантика задается отношением переходов между конфигурациями абстрактной машины [Apt09]. *Конфигурацией* называется пара $\langle P, s\rangle$, в которой P — программа, а s — состояние. Переход из конфигурации $\langle P, s\rangle$ в конфигурацию $\langle P', s'\rangle$ (обозначение: $\langle P, s\rangle \to \langle P', s'\rangle$) соответствует исполнению одного шага программы P в состоянии s: в результате состояние меняется на s', при этом машине «остается» исполнить программу P'. Чтобы выразить условие останова, вводится «пустая» программа ε:

$$P;\varepsilon \;\equiv\; \varepsilon;P \;\equiv\; P, \text{ для любой программы } P.$$

Запись $\langle P, s\rangle \to \langle \varepsilon, s'\rangle$ означает, что исполнение одного шага программы P в состоянии s вызывает останов в состоянии s'.

Идея структурной операционной семантики состоит в определении семантики программы через семантику ее фрагментов. Для этого используется *система переходов языка* — специальная *дедуктивная система* (множество аксиом и правил вывода), сформулированная для операторов языка: переход $\langle P, s\rangle \to \langle P', s'\rangle$ возможен тогда и только тогда, когда он выводим в системе переходов. Более подробно понятие дедуктивной системы будет разобрано в лекции 3, а пока рассмотрим систему переходов языка while [Apt09] (таблица 2.2).

Таблица 2.2. Система переходов языка программирования while

(1)	$\langle \mathbf{skip}, s\rangle \to \langle \varepsilon, s\rangle$	Исполнение оператора **skip** состоит из одного шага и не изменяет состояния
(2)	$\langle x := t, s\rangle \to \langle \varepsilon, s[x := s(t)]\rangle$	Исполнение оператора присваивания состоит из одного шага и приводит к соответствующему изменению состояния
(3)	$\dfrac{\langle P_1, s_1\rangle \to \langle P_2, s_2\rangle}{\langle P_1; P, s_1\rangle \to \langle P_2; P, s_2\rangle}$	Допускается последовательная композиция программ пре- и пост-конфигураций перехода с одной и той же произвольной программой
(4)	$\langle \mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2\ \mathbf{end}, s\rangle \to \langle P_1, s\rangle$, если $s \vDash B$	Исполнение условного оператора, если условие истинно, сводится к исполнению ветви **then** в том же состоянии

(5)	$\langle \textbf{if } B \textbf{ then } P_1 \textbf{ else } P_2 \textbf{ end}, s\rangle \to \langle P_2, s\rangle$, если $s \vDash \neg B$	Исполнение условного оператора, если условие ложно, сводится к исполнению ветви **else** в том же состоянии
(6)	$\langle \textbf{while } B \textbf{ do } P \textbf{ end}, s\rangle \to \langle P; \textbf{while } B \textbf{ do } P \textbf{ end}, s\rangle$, если $s \vDash B$	Исполнение оператора цикла, если условие истинно, сводится к исполнению тела цикла, а затем — цикла еще раз
(7)	$\langle \textbf{while } B \textbf{ do } P \textbf{ end}, s\rangle \to \langle \varepsilon, s\rangle$, если $s \vDash \neg B$	Исполнение оператора цикла, если условие ложно, занимает один шаг и не меняет состояние

В описании системы переходов используются следующие обозначения:

- $s(t)$ — значение терма t в состоянии s: значение терма определяется индуктивно через значения подтермов с использованием функций предметной области;
- $s[x := v]$ — состояние, полученное из состояния s сопоставлением переменной x значения v: $s[x := v](x) = v$ и $s[x := v](y) = s(y)$, если $y \neq x$;
- $s \vDash B$ — истинность логической формулы B в состоянии s.

Последовательностью переходов программы P в состоянии s называется конечная или бесконечная цепочка конфигураций $\{\langle P_i, s_i\rangle\}_{i\geq 0}$, такая что:

1. $P_0 = P$;
2. $s_0 = s$;
3. $\langle P_i, s_i\rangle \to \langle P_{i+1}, s_{i+1}\rangle$, для всех $i \geq 0$
   (кроме последнего индекса, если последовательность конечна).

Вычислением программы P в состоянии s называется последовательность переходов, которая не может быть продолжена. Говорят, что вычисление π *завершается* в состоянии s', если π конечно и его последняя конфигурация имеет вид $\langle \varepsilon, s'\rangle$. Говорят, что программа P *может зациклиться* в состоянии s, если существует бесконечное вычисление P в s.

Справедливо утверждение: для любой while-программы в любом состоянии существует в точности одно вычисление (другими словами, while-программы детерминированы).

С использованием введенных понятий и обозначений формальная семантика языка программирования задается следующим образом:

$$M[\![P]\!](s) = \{\, s' \mid \langle P, s\rangle \to^* \langle \varepsilon, s'\rangle \,\} \cup \begin{cases} \{\omega\}, & \text{если } P \text{ может зациклиться в } s; \\ \emptyset, & \text{в противном случае.} \end{cases}$$

Здесь $\to^*$ — рефлексивное и транзитивное замыкание отношения переходов.

Пример 2.3

Вернемся к программе DIV. Рассмотрим состояние $s = \{a \mapsto 3, b \mapsto 2, q \mapsto -1, r \mapsto -1\}$, или сокращенно $(3, 2, -1, -1)$. Нетрудно проверить, что вычисление программы DIV в состоянии s конечно и имеет следующий вид:

0. $\langle q \coloneqq 0; r \coloneqq a; \textbf{while } r \geq b \textbf{ do } q \coloneqq q + 1; r \coloneqq r - b \textbf{ end}, (3, 2, -1, -1)\rangle$,
1. $\langle r \coloneqq a; \textbf{while } r \geq b \textbf{ do } q \coloneqq q + 1; r \coloneqq r - b \textbf{ end}, (3, 2, 0, -1)\rangle$,
2. $\langle \textbf{while } r \geq b \textbf{ do } q \coloneqq q + 1; r \coloneqq r - b \textbf{ end}, (3, 2, 0, 3)\rangle$,
3. $\langle q \coloneqq q + 1; r \coloneqq r - b; \textbf{while } r \geq b \textbf{ do } q \coloneqq q + 1; r \coloneqq r - b \textbf{ end}, (3, 2, 0, 3)\rangle$,
4. $\langle r \coloneqq r - b; \textbf{while } r \geq b \textbf{ do } q \coloneqq q + 1; r \coloneqq r - b \textbf{ end}, (3, 2, 1, 3)\rangle$,
5. $\langle \textbf{while } r \geq b \textbf{ do } q \coloneqq q + 1; r \coloneqq r - b \textbf{ end}, (3, 2, 1, 1)\rangle$,
6. $\langle \varepsilon, (3, 2, 1, 1)\rangle$.

□

2.2.2. Аксиоматическая семантика

Если операционная семантика формализует поведение исполнителя программ, то *аксиоматическая*, или *дедуктивная*, *семантика* формализует процесс доказательства корректности. Основополагающей работой в этой области является статья Тони Хоара (Charles Antony Richard Hoare, род. в 1934 г.) 1969 г., ставшая классикой [Hoa69].

Аксиоматическая семантика языка программирования базируется на дедуктивной системе, так называемой *логике Хоара*, заданной для операторов языка. В ней, в отличие от системы переходов языка, выводятся не действия исполнителя, а условия корректности, так называемые *тройки Хоара*. Аксиомы описывают семантику элементарных операторов (оператора **skip**, оператора присваивания), а правила вывода — семантику управляющих операторов (условного оператора, оператора цикла). Аксиоматическая семантика игнорирует способ исполнения программы, давая возможность рассуждать о «высоких» свойствах.

Вспомним определения, данные в лекции 1.

Условием корректности называется тройка $\{\varphi\}P\{\psi\}$, где P — программа, а φ и ψ — логические формулы, называемые соответственно *пред-* и *постусловием*, а вместе — *про-*

граммным контрактом. Говорят, что программа P *корректна* относительно φ и ψ (обозначается: $\vDash \{\varphi\}P\{\psi\}$), если для любого начального состояния, в котором истинно условие φ, если вычисление P завершается[16], то в конечном состоянии истинно условие ψ:

$$\text{если } s \vDash \varphi \text{ и } M[\![P]\!](s) \neq \omega, \text{ то } M[\![P]\!](s) \vDash \psi, \text{ для любого состояния } s.$$

Заметим: речь идет только о свойствах *частичной корректности* (методы доказательства завершимости программ будут представлены в лекции 6).

Пример 2.4

Рассмотрим тривиальные тройки Хоара:

- $\vDash \{false\}P\{\psi\}$, для любой программы P и любого постусловия ψ;
- $\vDash \{\varphi\}P\{true\}$, для любой программы P и любого предусловия φ.

□

Ниже в таблице 2.3 определяется аксиоматическая семантика языка while [Ben12] и даются неформальные пояснения к составляющим ее аксиомам и правилам вывода.

Таблица 2.3. Аксиоматическая семантика языка программирования while

(1)	$\{\varphi\}\mathbf{skip}\{\varphi\}$	**Аксиома пустого оператора**: условие, истинное перед исполнением пустого оператора, остается истинным и после исполнения оператора
(2)	$\{\varphi[x := t]\}x := t\{\varphi\}$	**Аксиома оператора присваивания**: для того чтобы после исполнения присваивания условие стало истинным, необходимо достаточно чтобы перед исполнением оператора было истинно то же условие, но с соответствующей подстановкой
(3)	$\dfrac{\{\varphi\}P_1\{\chi\},\ \{\chi\}P_2\{\psi\}}{\{\varphi\}P_1; P_2\{\psi\}}$	**Правило последовательной композиции**: если постусловие одной программы совпадает с предусловием другой, то их последовательная композиция корректна относительно предусловия первой программы и постусловия второй («склейка» контрактов)
(4)	$\dfrac{\{\varphi \wedge B\}P_1\{\psi\},\ \{\varphi \wedge \neg B\}P_2\{\psi\}}{\{\varphi\}\mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2\ \mathbf{end}\{\psi\}}$	**Правило условного оператора**: для корректности условного оператора необходимо и достаточно, чтобы ветвь **then** была корректна относительно предусловия, усиленного условием ветвления, а ветвь **else** — относительно предусловия, усиленного отрицанием условия ветвления
(5)	$\dfrac{\{\varphi \wedge B\}P\{\varphi\}}{\{\varphi\}\mathbf{while}\ B\ \mathbf{do}\ P\ \mathbf{end}\{\varphi \wedge \neg B\}}$	**Правило оператора цикла**: если известен инвариант цикла (условие, удовлетворяющее верхней части правила), то оператор цикла корректен относительно инварианта и инварианта, усиленного условием выхода из цикла
(6)	$\dfrac{\varphi \to \varphi',\ \{\varphi'\}P\{\psi'\},\ \psi' \to \psi}{\{\varphi\}P\{\psi\}}$	**Правило консеквенции**: корректность программы не нарушается при усилении предусловия и ослаблении постусловия

[16] Для определенности считаем, что такое вычисление одно.

В аксиоме оператора присваивания используется обозначение $\varphi[x \coloneqq t]$ — результат подстановки в формулу φ терма t вместо каждого вхождения переменной x[17].

Пример 2.5

Рассмотрим формулу $a = b \cdot q + r$. Последовательно применим к ней две подстановки:

$$[r \coloneqq r - b] \quad \text{и} \quad [q \coloneqq q + 1]:$$

1. $\big((a = b \cdot q + r)[r \coloneqq r - b]\big)[q \coloneqq q + 1] =$
2. $\big(a = b \cdot q + (r - b)\big)[q \coloneqq q + 1] =$
3. $a = b \cdot (q + 1) + (r - b) \equiv$
4. $a = b \cdot q + r.$

Обратите внимание: в результате применения подстановок мы получили ту же самую формулу $a = b \cdot q + r$ — формула *инвариантна* относительно заданного преобразования.

□

Известно, что логика Хоара является *значимой* (из доказуемости условия корректности вытекает его истинность) и *полной* (из истинности условия корректности вытекает его доказуемость)[18] [Ben12]. Доказуемость условия корректности $\{\varphi\}P\{\psi\}$ (обозначается: $\vdash \{\varphi\}P\{\psi\}$) есть ничто иное, как возможность его получения из аксиом путем применения правил вывода. Таким образом, значимость логики Хоара вместе с ее полнотой означают следующее:

$$\vdash \{\varphi\}P\{\psi\} \text{ тогда и только тогда, когда } \vDash \{\varphi\}P\{\psi\}.$$

Проиллюстрируем значимость системы на примере оператора присваивания $x \coloneqq t$. Исполнение этого оператора описывается переходом $\langle x \coloneqq t, s\rangle \to \langle \varepsilon, s[x \coloneqq s(t)]\rangle$. Для произвольного условия φ имеем: $\vdash \{\varphi[x \coloneqq t]\}x \coloneqq t\{\varphi\}$ (аксиома, очевидно, доказуема). Зафиксируем произвольное состояние s, такое что $s \vDash \varphi[x \coloneqq t]$, и покажем, что

$$M[\![x \coloneqq t]\!](s) = s[x \coloneqq s(t)] \vDash \varphi.$$

---

[17] В математической логике часто используют обозначение $\varphi\{t/x\}$.

[18] Здесь уместнее говорить об *относительной* полноте дедуктивной системы в том смысле, что свойства предметной области в ней не доказываются: истинные формулы рассматриваются в качестве аксиом [Ben12].

Это и будет означать, что $\vDash \{\varphi[x \coloneqq t]\}x \coloneqq t\{\varphi\}$. Не ограничивая общности рассуждений, будем считать, что условие φ имеет вид $p(x)$, где p — некоторый одноместный предикатный символ (обозначим интерпретирующее его отношение символом $[\![p]\!]$). Приведенная ниже цепочка эквивалентностей доказывает требуемое утверждение:

$$s \vDash p(x)[x \coloneqq t] \iff s \vDash p(t) \iff s(t) \in [\![p]\!] \iff s[x \coloneqq s(t)] \vDash p(x).$$

Остановимся подробнее на правиле вывода для оператора цикла:

$$\frac{\{\varphi \wedge B\}P\{\varphi\}}{\{\varphi\}\textbf{while } B \textbf{ do } P \textbf{ end}\{\varphi \wedge \neg B\}}.$$

Условие φ, указанное в этом правиле, называется *инвариантом цикла* (исполнение тела цикла не нарушает истинность этого условия). Для проведения дедуктивной верификации нужно знать «информативные», как можно более «точные» инварианты циклов, однако поиск таких инвариантов, вообще говоря, не поддается автоматизации. Сказанное станет понятным из лекций 3 и 5; пока лишь отметим, что тривиальные условия, $true$ и $false$, являются инвариантами любого цикла, однако они бесполезны для верификации.

Пример 2.6

Условие $a = b \cdot q + r$ является инвариантом цикла, присутствующего в программе DIV, что было доказано в предыдущем примере:

$$\vDash \{(a = b \cdot q + r)\}\, q \coloneqq q + 1; r \coloneqq r - b\, \{a = b \cdot q + r\}.$$

Заметим: в приведенном предусловии нет условия цикла, как того требует определение инварианта, однако его можно добавить, не нарушив истинность условия корректности[19].

□

Программный контракт, безусловно, является описанием семантики программы, но это описание не вполне точно. Правило консеквенции позволяет усиливать предусловие программы и ослаблять постусловие без ущерба для корректности. Таким образом, для любой программы можно построить семейство контрактов разной степени детализации. В каче-

[19] Как правило, инварианты φ, такие что $\vDash \{\varphi\}P\{\varphi\}$, не являются «информативными»: они не принимают в расчет условие цикла.

стве семантики программы следует рассматривать наиболее «сильный» из них. Формализация этой идеи была предложена Эдсгером Дейкстрой (Edsger Dijkstra, 1930-2002) в 1976 г. и базируется на понятии *слабейшего предусловия* (*wp*, *weakest precondition*) [Dij76][20].

Говорят, что формула φ' *слабее* формулы φ, если истинна импликация $\varphi \to \varphi'$. Например, формула $x > 0$ слабее формулы $(x = 1) \vee (x = 2) \vee (x = 3)$. Очевидно, что чем слабее формула, тем большее число состояний программы она характеризует:

- $false$ — пустое множество состояний;
- $true$ — множество всех возможных состояний.

Слабейшим предусловием программы P относительно постусловия ψ (обозначается: $wp(P, \psi)$) называется слабейшая формула φ, для которой $\vDash \{\varphi\}P\{\psi\}$[21].

Заметим, что для заданной программы отображение wp ставит в соответствие одному предикату (постусловию) другой предикат (слабейшее предусловие). По этой причине это отображение называют *предикатным преобразователем*.

Свойства предикатного преобразователя wp показаны в таблице 2.4.

Таблица 2.4. Свойства слабейшего предусловия

(1)	$\vDash \big(wp(P,\psi) \wedge wp(P,\chi)\big) \leftrightarrow wp(P, \psi \wedge \chi)$	Свойство дистрибутивности
(2)	$\vDash \neg wp(P,\neg\psi) \to wp(P,\psi)$	Свойство двойственности
(3)	$\vDash (\psi \to \chi) \to \big(wp(P,\psi) \to wp(P,\chi)\big)$	Свойство монотонности

Обратите внимание: в свойстве двойственности wp имеет место импликация, а не эквивалентность: в состояниях s, в которых программа P зацикливается ($M[\![P]\!](s) = \omega$), если такие есть, истинно и предусловие $wp(P, \psi)$ и предусловие $wp(P, \neg\psi)$.

В таблице 2.5 приведены правила вычисления слабейшего предусловия для языка while [Ben12]. Они в целом соответствуют аксиомам и правилам вывода аксиоматической семантики; единственное исключение — здесь нет консеквенции.

---

[20] Другой вариант — *сильнейшее постусловие* (*sp*, *strongest postcondition*).

[21] Строго говоря, это определения *слабейшего свободного предусловия* (*wlp*, *weakest liberal precondition*); оригинальное определение слабейшего предусловия требует истинности условия полной корректности.

Таблица 2.5. Правила вычисления слабейшего предусловия

(1)	$wp(\mathbf{skip}, \psi) = \psi$
(2)	$wp(x \coloneqq t, \psi) = \psi[x \coloneqq t]$
(3)	$wp(P_1; P_2, \psi) = wp\big(P_1, wp(P_2, \psi)\big)$
(4)	$wp(\mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2\ \mathbf{end}, \psi) = \big(B \wedge wp(P_1, \psi)\big) \vee \big(\neg B \wedge wp(P_2, \psi)\big)$
(5)	$wp(\mathbf{while}\ B\ \mathbf{do}\ P\ \mathbf{end}, \psi) = (\neg B \wedge \psi) \vee \big(B \wedge wp(P; \mathbf{while}\ B\ \mathbf{do}\ P\ \mathbf{end}, \psi)\big)$

Ранее мы договорились считать, что для языка программирования задана формальная семантика, если определено отображение M, ставящее в соответствие каждой программе P на этом языке и каждому начальному состоянию s множество возможных конечных состояний $M[\![P]\!](s)$. В рамках аксиоматического подхода мы немного отойдем от принятой договоренности — отображение M определяется для пары программа-постусловие[22]:

$$M[\![P, \psi]\!](s) = \begin{cases} \{\bot\}, & \text{если } s \not\models wp(P, \psi); \\ \{\, s' \mid s' \models \psi \,\} \cup \{\omega\}, & \text{иначе.} \end{cases}$$

Пример 2.7

Вычислим слабейшее предусловие программы $x \coloneqq x + y; y \coloneqq x - y; x \coloneqq x - y$ относительно постусловия $(x = a) \wedge (y = b)$.

1. $wp\big(x \coloneqq x + y; y \coloneqq x - y; x \coloneqq x - y, (x = a) \wedge (y = b)\big) =$
2. $wp\Big(x \coloneqq x + y; y \coloneqq x - y, wp\big(x \coloneqq x - y, (x = a) \wedge (y = b)\big)\Big) =$
3. $wp\big(x \coloneqq x + y; y \coloneqq x - y, \big((x = a) \wedge (y = b)\big)[x \coloneqq x - y]\big) =$
4. $wp\Big(x \coloneqq x + y; y \coloneqq x - y, \big((x - y) = a\big) \wedge (y = b)\Big) =$
5. $wp\Big(x \coloneqq x + y, wp\Big(y \coloneqq x - y, \big((x - y) = a\big) \wedge (y = b)\Big)\Big) =$
6. $wp\Big(x \coloneqq x + y, \Big(\big((x - y) = a\big) \wedge (y = b)\Big)[y \coloneqq x - y]\Big) =$
7. $wp\Big(x \coloneqq x + y, \Big(\big(x - (x - y)\big) = a\Big) \wedge \big((x - y) = b\big)\Big) =$
8. $\Big(\Big(\big(x - (x - y)\big) = a\Big) \wedge \big((x - y) = b\big)\Big)[x \coloneqq x + y] =$
9. $\Big(\Big((x + y) - \big((x + y) - y\big)\Big) = a\Big) \wedge \Big(\big((x + y) - y\big) = b\Big) =$
10. $(y = a) \wedge (x = b)$.

□

[22] Значение $\bot$ («дно») интерпретируется как неопределенность поведения программы вследствие нарушения контракта, ω — как зацикливание (условие частичной корректности допускает зацикливание).

Для того чтобы доказать условие корректности $\{\varphi\}P\{\psi\}$, нужно вычислить слабейшее предусловие $wp(P,\psi)$ и проверить общезначимость (тождественную истинность) импликации $\varphi \to wp(P,\psi)$. Задача проверки общезначимости логических формул является вычислительно трудной, если не неразрешимой, но для некоторых предметных областей известны эффективные разрешающие процедуры (см. лекцию 8).

2.2.3. Доказательное программирование

Последняя тема, затрагиваемая в этой лекции, — метод Дейкстры построения корректных программ, так называемый *метод доказательного программирования* [Dij76]. Основная идея подхода состоит в том, что разработка программы осуществляется совместно с доказательством ее корректности. Ключевым элементом метода является техника построения циклов. Предположим, что требуется найти программу P, такую что $\vDash \{\varphi\}P\{\psi\}$ и имеющую следующий вид [Apt09]:

$$Q;\ \textbf{while}\ B\ \textbf{do}\ R\ \textbf{end}.$$

Если бы программа P была известна, то для доказательства условия корректности достаточно было бы найти формулу χ, удовлетворяющую следующим свойствам:

1. формула χ истинна перед входом в цикл: $\vDash \{\varphi\}Q\{\chi\}$;
2. формула χ является инвариантом цикла: $\vDash \{\chi \wedge B\}R\{\chi\}$;
3. формула ψ истинна после выхода из цикла: $\vDash (\chi \wedge \neg B) \to \psi$.

Вышесказанное можно написать более наглядно в форме *наброска доказательства*:

```
{φ} /* предусловие */
Q;
{inv: χ} /* инвариант цикла */
while B do
  {χ ∧ B}
  R
  {χ}
end
{χ ∧ ¬B}
{ψ} /* постусловие */
```

Основная сложность состоит в нахождении χ — инварианта несуществующего цикла — для заданных φ и ψ. Одна из стратегий поиска состоит в обобщении постусловия, например,

путем замены некоторой константы на переменную (тема синтеза инвариантов рассматривается в лекции 7).

Подчеркнем: в методе Дейкстры инвариант ищется до построения цикла; кроме того, он является неотъемлемой частью программы.

Исследования по семантике программ начались в 1960-ые гг. и продолжаются по сей день. Подробное описание методов формализации языков программирования, включая *денотационную семантику*[23], не рассмотренную в лекции, можно найти в специализированной литературе, например, в книге [Gun92].

2.3. Вопросы и упражнения

1. Чем операционный подход к определению семантики языков программирования отличается от аксиоматического?
2. Верно ли, что две функционально эквивалентные программы имеют одинаковую семантику?
3. Выпишите сигнатуру программы P, реализующей алгоритм Евклида (см. лекцию 1). Расширьте эту сигнатуру предикатным символом $\text{НОД}^{(2)}$, определите аксиомы и правила вывода предметной области.
4. Докажите утверждение о единственности вычисления любой while-программы в произвольном начальном состоянии.
5. Постройте вычисление программы целочисленного деления DIV для $a = 7$ и $b = 3$ (начальные значения переменных q и r могут быть любыми).
6. Постройте вычисления программ P и Q, реализующих алгоритм Евклида (см. лекцию 1), для $a = 14$ и $b = 21$.

---

[23] В денотационной семантике каждой конструкции языка программирования сопоставляется его математический аналог, так называемый *денотат*. Денотаты сложных конструкций определяются индуктивно через денотаты составляющих их элементов. Программе P сопоставляется функция (в математическом смысле) $M[\![P]\!]$. Подход разработан в конце 1960-ых — начале 1970-ых гг. в работах Дана Скотта (Dana Scott, род. в 1932 г.) и Кристофера Стрейчи (Christopher Strachey, 1916-1975). Денотационная семантика языка программирования while рассматривается, например, в книге [Mit96].

7. Опишите семантику оператора недетерминированного выбора **choice**, используя операционный и аксиоматический подходы.
8. Опишите семантику оператора цикла **repeat** − **until**, используя операционный и аксиоматический подходы.
9. Определите операционную семантику следующего языка. Программа — последовательность $\{op_i\}_{i=0}^{n-1}$, в которой каждый элемент op_i — один из следующих операторов: **skip**, $x := t$ или **if** B **then goto** j (переход на j-ый оператор, если истинно условие B). Можно ли это сделать, не выходя за рамки структурной операционной семантики?
10. Докажите условие корректности $\{true\}\, x := a; y := b\, \{(x = a) \wedge (y = b)\}$, используя определенные в лекции аксиомы и правила вывода.
11. Предложите инвариант цикла для программы целочисленного деления DIV.
12. Предложите инварианты циклов для программ P и Q, реализующих алгоритм Евклида (см. лекцию 1). Совпадают ли эти инварианты?
13. Докажите свойства дистрибутивности, двойственности и монотонности слабейшего предусловия.
14. Какие состояния программы P характеризует условие $wp(P, false)$?
15. Докажите свойство дистрибутивности слабейшего предусловия относительно дизъюнкции: $wp(P, \psi) \vee wp(P, \chi) \rightarrow wp(P, \psi \vee \chi)$.
16. Вычислите $wp(x := x + a; y := y - 1, x = (b - y) \cdot a)$.
17. Постройте с помощью метода Дейкстры программу, вычисляющую сумму элементов целочисленного массива заданной длины.
18. Предложите способ вычисления сильнейшего постусловия программы (*sp*, *strongest postcondition*). Подсказка — считайте, что программа представлена в SSA-форме[24].
19. Сформулируйте свойства предикатного преобразователя sp.

---

[24] SSA-форма (от англ. Static Single Assignment) — это промежуточное представление программы, в котором каждой переменной значение присваивается лишь единожды [Aho06]. Чтобы обеспечить это требование, вводятся так называемые версии переменных — номер версии инкрементируется при каждом присваивании в переменную; при использовании переменной берется ее последняя версия. Итак, $sp(P, \varphi) = \big(\varphi[\text{оригиналы} := \text{версии}_\text{н}] \wedge ssa(P)\big)[\text{версии}_\text{к} := \text{оригиналы}]$, где $\text{версии}_\text{н}$ и $\text{версии}_\text{к}$ — начальные и конечные версии переменных соответственно. В задаче требуется найти способ построения формулы $ssa(P)$.

20. Какие состояния программы P характеризует условие $sp(P, true)$?

21. Вычислите $sp(x \coloneqq x + a; y \coloneqq y - 1, x = (b - y) \cdot a)$.

Лекция 3. Дедуктивная верификация последовательных программ

Некоторые автомашины дребезжат на ходу. Моя автомашина – это некоторая автомашина. Неудивительно, что моя автомашина дребезжит!

Р.М. Смаллиан.
Как же называется эта книга?

Рассматриваются основы дедуктивной верификации последовательных программ: приводятся необходимые сведения о дедуктивных системах и теории доказательств; дается представление о верификации программы как поиске доказательства условия корректности в дедуктивной системе, формализующей язык программирования (платформу). Показывается, что свойства циклов, так называемые инварианты, не выводимы из структуры программы. Задание инвариантов дает набросок доказательства корректности, проверку которого можно автоматизировать.

3.1. Дедуктивные системы и доказательства

Задача верификации — доказать корректность программы[25], т.е. соответствие программы требованиям. В классическом подходе, рассматриваемом нами, требования задаются с помощью пред- и постусловий, а верификация состоит в доказательстве частичной или полной корректности программы относительно этой пары условий. Когда говорится о доказательстве, предполагается, что есть дедуктивная система (исчисление), в рамках которой строится логический вывод. Такая дедуктивная система определяется для конкретного языка программирования, точнее, для конкретной платформы (нужно учитывать разрядность типов, представление данных в памяти и прочие «мелочи»). Эта система есть не что иное, как аксиоматическая семантика языка программирования (см. лекцию 2), называемая в честь Тони Хоара (Charles Antony Richard Hoare, род. в 1934 г.) *логикой Хоара*.

[25] На практике уместнее говорить о доказательстве или опровержении корректности программы, т.е. верификации/фальсификации. В противном случае постановка задачи становится абсурдной (см. первую аксиому М.Р. Шуры-Буры, упомянутую в лекции 1). Пример абсурдного высказывания [And79, рус. пер.]: «доказательство правильности весьма полезно и способствует обнаружению тех ошибок, которые могли быть пропущены, если бы доказательство не проводилось». Если доказана правильность, какие могут быть ошибки?

Итак, мы имеет классическую задачу математической логики: имеются *дедуктивная система* (множество аксиом и правил вывода) и *утверждение* (логическая формула, которую нужно доказать); требуется построить *доказательство* (логический вывод) заданного утверждения в заданной дедуктивной системе.

Доказательством называется последовательность формул $\{\varphi_i\}_{i=1}^{n}$, в которой каждая формула либо является аксиомой, либо может быть выведена из некоторых предыдущих формул путем применения одного из правил вывода. Последняя формула последовательности называется *теоремой*, а сама последовательность — ее доказательством. Формула φ, для которой существует доказательство, называется *доказуемой* (обозначается: $\vdash \varphi$) [Ben12]. В дальнейшем для краткости символ доказуемости мы будем опускать.

3.1.1. Примеры дедуктивных систем

Пожалуй, самой известной дедуктивной системой всех времен и народов является геометрия Евклида, описанная в «Началах» [Евк50] примерно в 300 г. до н.э. Представление о ней дается в средней школе. Сейчас же мы обратимся к системе Генцена (Gerhard Gentzen, 1909-1945), предложенной в 1930-ые гг. [Ben12]. Этот пример не связан с программированием; его цель — проиллюстрировать понятие логического вывода с формальной стороны, без смыслового наполнения (взгляд со стороны компьютера).

Дедуктивная система Генцена содержит один тип аксиом и два типа правил вывода, так называемые α- и β-правила:

- *аксиомы* — множества литералов (высказываний и их отрицаний), содержащие хотя бы одну контрарную пару (некоторое высказывание и его отрицание);
- *α-правило* (см. таблицу 3.1):

  $\frac{U \cup \{\alpha_1, \alpha_2\}}{U \cup \{\alpha\}}$, где U — множество формул, не содержащее α_1 и α_2;
- *β-правило* (см. таблицу 3.1):

  $\frac{U_1 \cup \{\beta_1\}, U_2 \cup \{\beta_2\}}{U_1 \cup U_2 \cup \{\beta\}}$, где U_1 не содержит β_1, а U_2 не содержит β_2.

Таблица 3.1. α- и β-формулы в дедуктивной системе Генцена

α	α_1	α_2	β	β_1	β_2
$\neg\neg A$	A		—	—	—
$\neg(A_1 \wedge A_2)$	$\neg A_1$	$\neg A_2$	$(B_1 \wedge B_2)$	B_1	B_2
$(A_1 \vee A_2)$	A_1	A_2	$\neg(B_1 \vee B_2)$	$\neg B_1$	$\neg B_2$
…	…	…	…	…	…

Пример 3.1

Построим доказательство утверждения $\{\neg(p \wedge q) \vee (q \wedge p)\}$ в системе Генцена:

1. $\{\neg p, \neg q, p\}$ аксиома (контрарная пара $\{p, \neg p\}$);
2. $\{\neg p, \neg q, q\}$ аксиома (контрарная пара $\{q, \neg q\}$);
3. $\{\neg(p \wedge q), p\}$ α-правило (1, ∧);
4. $\{\neg(p \wedge q), q\}$ α-правило (2, ∧);
5. $\{\neg(p \wedge q), (q \wedge p)\}$ β-правило (3 и 4, ∧);
6. $\{\neg(p \wedge q) \vee (q \wedge p)\}$ α-правило (5, ∨).

Обратите внимание на два момента: первое — процедура логического вывода исключительно формальная (проверка доказательства легко может быть выполнена компьютером); второе — подобрать нужные аксиомы и правила вывода не так-то просто (поиск доказательства трудно автоматизировать).

□

3.1.2. Значимость и полнота дедуктивных систем

Определение не накладывает никаких ограничений на дедуктивные системы. Разумно использовать такие системы, в которых *доказуемость* утверждений соотносится с их *истинностью*. Дедуктивная система называется *значимой* или *семантически непротиворечивой*, если все утверждения, доказуемые в этой системе, являются истинными. Дедуктивная система называется *полной*, если все истинные утверждения могут быть в ней доказаны. Упомянутые выше дедуктивные системы Евклида и Генцена являются как значимыми, так и полными [Ben12].

Работать имеет смысл только со значимыми системами — в незначимой системе могут быть доказаны ложные утверждения (представьте себе: программа, для которой доказана корректность, содержит ошибки). Полнота дедуктивной системы желательна, но не обязательна — для неполной системы существуют истинные утверждения, которые невозможно

в ней доказать (скажем, существуют корректные программы, которые невозможно верифицировать). Неполные дедуктивные системы часто встречаются в математике, например, аксиоматика Пеано для арифметики натуральных чисел[26] [Pea89].

Значимость и полноту можно определить и для методов верификации, даже если они не основаны на дедукции. Результатом применения любого метода верификации к программе является *вердикт*: либо положительный (в программе нет ошибок или ошибки не найдены), либо отрицательный (при верификации обнаружены ошибки)[27]. Метод называется *значимым*, если из того, что вердикт отрицательный, следует, что программа ошибочна; *полным* — если из того, что вердикт положительный, следует, что программа корректна [Tre08].

3.2. Дедуктивная верификация программ

Под *дедуктивной верификацией* понимается поиск доказательства условия корректности программы в дедуктивной системе (логике Хоара) соответствующего языка (платформы). Поиск доказательства является творческим процессом, который можно алгоритмизировать лишь в частных случаях. Например, проверка выполнимости (непротиворечивости) или общезначимости (тождественной истинности) формулы логики высказываний является NP-полной задачей; для теорий первого порядка эти задачи и вовсе могут быть алгоритмически неразрешимыми (см. лекцию 8).

Итак, если программа корректна, доказать это хотя и возможно (считаем, что дедуктивная система полна), но, вообще говоря, непросто. А что если программа ошибочна? Очевидно, в этом случае доказательства ее корректности не существует (считаем, что дедуктивная система значима). Как отличить эту ситуацию от ситуации, когда доказательство существует, но мы не может его найти? Общего рецепта тут нет, ясно лишь, что одной дедукции недостаточно — помогают опыт и интуиция.

---

[26] Неполнота аксиоматики Пеано является следствием теоремы Генцена о ее непротиворечивости и теоремы Геделя о неполноте.

[27] Для простоты здесь считается, что неопределенного вердикта нет.

3.2.1. Верификация программ на языке while

Рассмотрим язык while и соответствующую ему логику Хоара (см. лекцию 2). Известно, что эта дедуктивная система является *значимой* и *относительно полной* [Ben12]. Слово «относительно» говорит о том, что полнота имеет место в предположении, что все истинные формулы предметной области включены в множество аксиом (доказательство свойств предметной области не входит в компетенцию логики Хоара).

Пример 3.2

Докажем корректность программы целочисленного деления.

```
{a ≥ 0 ∧ b > 0}
q := 0;
r := a;
while r ≥ b do
  q := q + 1;
  r := r - b
end
{a = q·b + r ∧ 0 ≤ r < b}
```

Одно из возможных доказательств приведено в таблице 3.2.

Таблица 3.2. Доказательство корректности программы целочисленного деления

Условие корректности (тройка Хоара)	Аксиома или правило
$\{(a \geq 0) \wedge (b > 0)\}\, q \coloneqq 0\, \{(a \geq 0) \wedge (b > 0) \wedge (q = 0)\}$	аксиома 2
$\{(a \geq 0) \wedge (b > 0) \wedge (q = 0)\}\, r \coloneqq a\, \{(a \geq 0) \wedge (b > 0) \wedge (q = 0) \wedge (r = a)\}$	аксиома 2
$\{(a \geq 0) \wedge (b > 0)\}\, q \coloneqq 0; r \coloneqq a\, \{(a \geq 0) \wedge (b > 0) \wedge (q = 0) \wedge (r = a)\}$	правило 3
$\{(a = q \cdot b + r) \wedge (r \geq b)\}\, q \coloneqq q + 1\, \{(a = (q \cdot b\ + r) - b) \wedge (r \geq b)\}$	аксиома 2
$\{(a = (q \cdot b + r) - b) \wedge (r \geq b)\}\, r \coloneqq r - b\, \{(a = q \cdot b + r) \wedge (r \geq 0)\}$	аксиома 2
$\{(a = q \cdot b + r) \wedge (r \geq b)\}\, q \coloneqq q + 1; r \coloneqq r - b\, \{(a = q \cdot b + r) \wedge (r \geq 0)\}$	правило 3
$\{(a = q \cdot b + r) \wedge (r \geq 0) \wedge (r \geq b)\}\, q \coloneqq q + 1; r \coloneqq r - b\, \{(a = q \cdot b + r) \wedge (r \geq 0)\}$	правило 6
$\{(a = q \cdot b + r) \wedge (r \geq 0)\}$ **while** $r \geq b$ **do** ... **end** $\{(a = q \cdot b + r) \wedge (0 \leq r <\ b)\}$	правило 5
$\{(a \geq 0) \wedge (b > 0)\}\, q \coloneqq 0; r \coloneqq a\, \{(a = q \cdot b + r) \wedge (r \geq 0)\}$	правило 6
$\{(a \geq 0) \wedge (b >\ 0)\}\, q \coloneqq 0; r \coloneqq a;$ **while** $r \geq b$ **do** ... **end** $\{(a = q \cdot b + r) \wedge (0 \leq r < b)\}$	правило 3

□

Как правило, доказательства корректности программ являются громоздкими. Для их декомпозиции на простые фрагменты используются так называемые *аннотированные программы* — программы, в которые добавлены *утверждения* [Gri81]. Утверждение — это логическая формула, которая должна быть истинной каждый раз, когда исполнение программы проходит через соответствующую точку. Как пред- и постусловия, утверждения заключаются в фигурные скобки. Отдельный тип утверждений составляют предполагаемые *инварианты циклов* — они помещаются перед операторами циклов и помечаются ключевым словом **inv**. Инварианты должны быть заданы для всех циклов.

Пример 3.3

Программу целочисленного деления можно аннотировать следующим образом.

```
{a ≥ 0 ∧ b > 0} /* предусловие */
q := 0;
r := a;
{a = q·b + r ∧ 0 ≤ r ≤ a}       /* утверждение */
{inv: a = q·b + r ∧ 0 ≤ r ≤ a} /* инвариант цикла */
while r ≥ b do
  q := q + 1;
  r := r - b
end
{a = q·b + r ∧ 0 ≤ r < b} /* постусловие */
```

□

Аннотированная программа дает *набросок доказательства*: для верификации программы достаточно убедиться в корректности каждого ее фрагмента относительно ограничивающих его утверждений (в частности, в инвариантности «инвариантов циклов»). Эта работа может быть автоматизирована (при условии разрешимости предметной области).

Пусть нам нужно доказать условие корректности $\{\varphi\}P\{\psi\}$, где P — программный фрагмент, не содержащий циклов. Как это сделать? Построить слабейшее предусловие $wp(P,\psi)$ и проверить общезначимость импликации $\varphi \to wp(P,\psi)$. Построение слабейшего предусловия ациклической программы легко программируется, а проверка общезначимости формулы может быть решена средствами SMT-решателей (см. лекцию 8). На построении слабейшего предусловия основана, например, работа модуля Jessie платформы статического анализа Frama-C (см. лекцию 4).

Пример 3.4

Аннотации, записанные в предыдущем примере, дают набросок доказательства, состоящий в проверке истинности следующих формул:

1. $\{(a \geq 0) \wedge (b > 0)\} q \coloneqq 0; r \coloneqq a \{(a = q \cdot b + r) \wedge (0 \leq r \leq a)\}$;
2. $\{(a = q \cdot b + r) \wedge (0 \leq r \leq a) \wedge (r \geq b)\} q \coloneqq q + 1; r \coloneqq r - b \{(a = q \cdot b + r) \wedge (0 \leq r \leq a)\}$;
3. $\big((a = q \cdot b + r) \wedge (0 \leq r \leq a) \wedge (r < b)\big) \rightarrow \big((a = q \cdot b + r) \wedge (0 \leq r < b)\big)$.

□

Если в программе есть циклы, нужно найти их инварианты, а это краеугольный камень всей дедуктивной верификации. Поясним в чем трудность. Синтез содержательного инварианта цикла $W \equiv \mathbf{while}\ B\ \mathbf{do}\ P\ \boldsymbol{end}$ есть процедура близкая к построению слабейшего предусловия для заданного постусловия: если $wp(W, \varphi \wedge \neg B) = \varphi$, то φ — инвариант цикла[28]. Слабейшее предусловие определяется рекурсивно:

1. $wp(W, \psi) =$
2. $(\neg B \wedge \psi) \vee \big(B \wedge wp(P; W, \psi)\big) = (\neg B \wedge \psi) \vee \Big(B \wedge wp\big(P, wp(W, \psi)\big)\Big) =$
3. $(\neg B \wedge \psi) \vee \Big(B \wedge wp\Big(P, (\neg B \wedge \psi) \vee \Big(B \wedge wp\big(P, wp(W, \psi)\big)\Big)\Big)\Big) =$
4. …

Таким образом, построение слабейшего предусловия (а вместе с ним и инварианта цикла) приводит, вообще говоря, к бесконечному вычислению, что не конструктивно[29]. Важно отметить, что итерационный процесс построения слабейшего предусловия обладает в определенном смысле свойством *монотонности*: если отбросить дизъюнктивный член, зависящий от исходной программы, то в конце каждого шага получается более слабая формула по сравнению с той, что была вначале:

1. $wp(W, \psi) = \underline{false} \vee wp(W, \psi) =$
2. $\underline{(\neg B \wedge \psi)} \vee \big(B \wedge wp(P; W, \psi)\big) = (\neg B \wedge \psi) \vee \Big(B \wedge wp\big(P, wp(W, \psi)\big)\Big) =$

---

[28] В самом деле, $\varphi = wp(W, \varphi \wedge \neg B) = (\neg B \wedge \varphi) \vee \big(B \wedge wp(P; W, \varphi \wedge \neg B)\big) = (\neg B \wedge \varphi) \vee \Big(B \wedge wp\big(P, wp(W, \varphi \wedge \neg B)\big)\Big) = (\neg B \wedge \varphi) \vee \big(B \wedge wp(P, \varphi)\big)$. Откуда следует, что $(B \wedge \varphi) = \big(B \wedge wp(P, \varphi)\big)$, $(B \wedge \varphi) \rightarrow wp(P, \varphi)$ и $\{B \wedge \varphi\} P \{\varphi\}$.

[29] Конструктивизм — направление в математике, основанное на представлении о том, что объекты можно объявлять существующими в том и только в том случае, если можно указать способ их построения («существовать — значит быть построенным»).

3. $(\neg B \wedge \psi) \vee \Big(B \wedge wp\Big(P, (\neg B \wedge \psi) \vee \Big(B \wedge wp\big(P, wp(W, \psi)\big)\Big)\Big)\Big) \leftarrow$

   (свойство дистрибутивности слабейшего предусловия относительно дизъюнкции — см. упражнения к лекции 2)

   $\underline{(\neg B \wedge \psi) \vee \Big(B \wedge wp\big(P, (\neg B \wedge \psi)\big)\Big)} \vee \Big(B \wedge wp\Big(B \wedge wp\big(P, wp(W, \psi)\big)\Big)\Big) =$

4. …

Это наталкивает на мысль о возможности вычисления инварианта цикла как *неподвижной точки* заданного таким образом преобразователя формул (к этой теме мы еще вернемся в лекциях 7 и 14).

Пример 3.5

Рассмотрим простейшую циклическую программу на языке while [Ben12]:

```
while x > 0 do
  x := x - 1
end
```

Обозначим ее символом W и построим слабейшее предусловие для постусловия $x = 0$:

1. $wp(W, x = 0) =$
2. $\big(\neg(x > 0) \wedge (x = 0)\big) \vee \big((x > 0) \wedge wp(x \coloneqq x - 1; W, x = 0)\big) =$
3. $\underline{(x = 0)} \vee \Big((x > 0) \wedge wp\big(x \coloneqq x - 1, wp(W, x = 0)\big)\Big) =$
4. $(x = 0) \vee \big((x > 0) \wedge wp(W, x = 0)[x \coloneqq x - 1]\big) =$
5. $(x = 0) \vee \big((x > 0) \wedge wp(W, x = 1)\big) =$
6. $(x = 0) \vee \big((x = 1) \vee (x > 1) \wedge wp(x \coloneqq x - 1;\ W, x = 1)\big) =$
7. $\underline{(x = 0) \vee (x = 1)} \vee \big((x > 1) \wedge wp(W, x = 1)[x \coloneqq x - 1]\big) =$
8. …
9. $\underline{(x = 0) \vee (x = 1) \vee (x = 2) \vee \ldots} =$ бесконечное число шагов
10. $x \in \mathbb{N}_0$ обобщение

□

Рассмотрим стратегии верификации некоторых while-программ некоторых типов:

1. *линейных программ* — программ, представляющих собой линейные последовательности присваиваний;
2. *ациклических программ с ветвлениями* — программ, содержащих условные операторы, но без операторов цикла;
3. *циклических программ* — программ, включающих циклы.

3.2.2. Верификация линейных программ

Линейная программа (в компиляторной терминологии — *базовый блок*) имеет вид

$$P \equiv x_1 := t_1; x_2 := t_2; \dots; x_n := t_n.$$

Стратегия доказательства условия корректности $\{\varphi\}P\{\psi\}$ может быть следующей:

1. вычислить $wp(P, \psi)$, используя *метод обратных подстановок*:

   $wp(P, \psi) \equiv \psi[x_n := t_n] \dots [x_1 := t_1]$;
2. доказать, что исходное предусловие φ не слабее вычисленного:

   $\vDash \varphi \to wp(P, \psi)$.

Пример 3.6

Рассмотрим программу, осуществляющую перестановку значений двух переменных (фрагмент программы сортировки массива):

```
{x = a ∧ y = b}
t := x;
x := y;
y := t
{x = b ∧ y = a}
```

Применим к постусловию $(x = b) \wedge (y = a)$ обратную последовательность подстановок:

1. $\big((x = b) \wedge (y = a)\big)[y := t][x := y][t := x] =$
2. $\big((x = b) \wedge (t = a)\big)[x := y][t := x] =$
3. $\big((y = b) \wedge (t = a)\big)[t := x] =$
4. $(y = b) \wedge (x = a)$.

Получаем формулу, эквивалентную предусловию $(x = a) \wedge (y = b)$.

□

3.2.3. Верификация ациклических программ

Для наглядности рассмотрим программу, содержащую один условный оператор:

$P \equiv Q$; **if** B **then** P_1 **else** P_2 **end**, где Q, P_1 и P_2 — линейные программы.

Доказательство условия корректности $\{\varphi\}P\{\psi\}$ может быть разбито на две части, соответствующие ветвям **then** и **else** условного оператора:

1. вычислить условие прохода по ветви **then** после исполнения Q: $wp(Q, B)$;
2. доказать условие корректности $\{\varphi_1\}Q; P_1\{\psi\}$, где $\varphi_1 \equiv \varphi \wedge wp(Q, B)$;
3. вычислить условие прохода по ветви **else** после исполнения Q: $wp(Q, \neg B)$;
4. доказать условие корректности $\{\varphi_2\}Q; P_2\{\psi\}$, где $\varphi_2 \equiv \varphi \wedge wp(Q, \neg B)$.

Если программа содержит несколько условных операторов, то для доказательства ее корректности можно выписать все возможные пути между начальной и конечной точками (их конечное число, поскольку в программе нет циклов), построить условия прохождения по этим путям и доказать условие корректности для каждого из них. Можно поступить проще: не перебирать все возможные пути, а вычислить слабейшее предусловие для программы в целом и работать с ним.

Пример 3.7

Рассмотрим более крупный фрагмент программы сортировки массива, аннотированный пред- и постусловием:

```
{true}
x := a;
y := b;
if x > y then
  t := x;
  x := y;
  y := t
end
{(x ≤ y) ∧ ((x = a ∧ y = b) ∨ (x = b ∧ y = a))}
```

Рассмотрим путь, соответствующий проходу по ветви **then**:

1. вычислим условие прохода по пути:

   $wp(x \coloneqq a; y \coloneqq b, x > y) = (x > y)[y \coloneqq b][x \coloneqq a] = (a > b)$;

2. докажем условие корректности, соответствующее пути:

$\{true \wedge (a > b)\}$

$x \coloneqq a; y \coloneqq b; t \coloneqq x; x \coloneqq y; y \coloneqq t$

$\{(x \leq y) \wedge \big((x = a) \wedge (y = b)\big) \vee \big((x = b) \wedge (y = a)\big)\}$:

a. вычислим слабейшее предусловие:

$$wp\Big(x \coloneqq a; y \coloneqq b; t \coloneqq x; x \coloneqq y; y \coloneqq t, (x \leq y) \wedge \big((x = a) \wedge (y = b)\big) \vee \big((x = b) \wedge (y = a)\big)\Big) =$$

$$(x \leq y) \wedge \Big(\big((x = a) \wedge (y = b)\big) \vee \big((x = b) \wedge (y = a)\big)\Big)[y \coloneqq t][x \coloneqq y][t \coloneqq x][y \coloneqq b][x \coloneqq a] =$$

$$(b \leq a) \wedge \Big(\big((b = a) \wedge (a = b)\big) \vee \big((b = b) \wedge (a = a)\big)\Big) =$$

$$(b \leq a);$$

b. докажем утверждение $(a > b) \rightarrow (b \leq a)$.

□

3.2.4. Верификация циклических программ

Для наглядности рассмотрим программу, содержащую один оператор цикла:

$P \equiv Q;$ **while** B **do** R **end**, где Q и R — линейные программы.

В этом случае стратегия доказательства условия корректности $\{\varphi\}P\{\psi\}$ следующая:

1. найти инвариант χ: $\{\chi \wedge B\}R\{\chi\}$;
2. доказать условие корректности $\{\varphi\}Q\{\chi\}$;
3. доказать утверждение $\{\chi \wedge \neg B\} \rightarrow \psi$.

Пример 3.8

Рассмотрим программу вычисления суммы элементов числового массива размера N (считаем, что индексация начинается с нуля, как в языке программирования C).

```
{N ≥ 0}
s := 0;
n := 0;
while n < N do
  s := s + x[n];
  n := n + 1
end
{s = sum(x, N)}
```

Формальный анализ программ с массивами базируется на *теории массивов*, предложенной Джоном Маккарти (John McCarthy, 1927-2011) в 1962 г. [Mcc62]. Сигнатура теории содержит два функциональных символа: $read$: $A \times I \rightarrow E$ (соответствует операции [] в правой

части присваивания: $x \coloneqq a[i]$) и $write: A \times I \times E \to A$ (соответствует операции [] в левой части присваивания: $a[i] \coloneqq x$), где I, E и A— это типы индекса, элемента и массива соответственно. Семантика задается следующими аксиомами[30]:

- $(i = j) \to read(write(a, i, e), j) = e$;
- $(i \neq j) \to read(write(a, i, e), j) = read(a, j)$.

Функция sum, используемая в постусловии, определяется индуктивно:

1. если $n = 0$, то $sum(x, n) = 0$;
2. если $n > 0$, то $sum(x, n) = sum(x, n-1) + x[n-1]$.

В качестве инварианта рассмотрим условие $\big(s = sum(x, n)\big) \wedge (n \leq N)$:

1. проверим, что это действительно инвариант:

   $\{(n < N) \wedge \big(s = sum(x, n)\big) \wedge (n \leq N)\} s \coloneqq s + x[n]; n \coloneqq n + 1 \{\big(s = sum(x, n)\big) \wedge (n \leq N)\}$

   a. $wp\Big(s \coloneqq s + x[n]; n \coloneqq n + 1, \big(s = sum(x, n)\big) \wedge (n \leq N)\Big) =$

   $\Big(\big(s = sum(x, n)\big) \wedge (n \leq N)\Big)[n \coloneqq n + 1]\big[s \coloneqq s + x[n]\big] =$

   $\big(s + x[n] = sum(x, n + 1)\big) \wedge (0 \leq n + 1 \leq N) =$

   $\big(s = sum(x, n)\big) \wedge (n + 1 \leq N)$;

   b. $\Big(\big(s = sum(x, n)\big) \wedge (n + 1 \leq N)\Big) \to \Big(\big(s = sum(x, n)\big) \wedge (n \leq N)\Big)$;

2. докажем $\{N \geq 0\} s \coloneqq 0; n \coloneqq 0 \{\big(s = sum(x, n)\big) \wedge (n \leq N)\}$:

   a. $wp\Big(s \coloneqq 0; n \coloneqq 0, \big(s = sum(x, n)\big) \wedge (n \leq N)\Big) =$

   $\Big(\big(s = sum(x, n)\big) \wedge (n \leq N)\Big)[n \coloneqq 0][s \coloneqq 0] =$

   $\big(0 = sum(x, 0)\big) \wedge (0 \leq N) =$

   $N \geq 0$;

3. докажем $\Big(\big(s = sum(x, n)\big) \wedge (n \leq N) \wedge \neg(n < N)\Big) \to \big(s = sum(x, N)\big)$:

[30] В указанных аксиомах предполагается, что все используемые переменные связаны кванторами всеобщности.

a. $\Big(\big(s = sum(x,n)\big) \wedge (n \leq N) \wedge \neg(n < N)\Big) =$

$\Big(\big(s = sum(x,n)\big) \wedge (n = N)\Big) =$

$\Big(\big(s = sum(x,N)\big) \wedge (n = N)\Big) \rightarrow$

$(s = sum(x,N))$.

□

Более подробно дедуктивная верификация программ с циклами рассматривается в лекциях 5 и 6, посвященных методу Флойда [Flo67].

3.3. Вопросы и упражнения

1. Что такое дедуктивная система? Приведите примеры дедуктивных систем.
2. Дайте определения значимости и полноты дедуктивной системы. Проанализируйте известные вам дедуктивные системы на значимость и полноту.
3. Приведите примеры дедуктивных систем следующих типов:
   a. значимая и полная;
   b. значимая и неполная;
   c. незначимая и полная;
   d. незначимая и неполная.
4. Расширьте дедуктивную систему Генцена α- и β-правилами для логической связки $\rightarrow$ (импликация). Постройте доказательство формулы $(p \vee q) \rightarrow (q \vee p)$.
5. Дайте определения значимости и полноты метода верификации. Приведите примеры методов верификации для всех комбинаций значимости/незначимости, полноты/неполноты.
6. Оцените возможность автоматизации дедуктивной верификации программ. Какие проблемы возникают на пути к автоматизации?
7. Рассмотрите возможность дедуктивной «фальсификации» программ — построения дедуктивных систем, предназначенных для доказательства некорректности (опровержения корректности).

8. Докажите следующее условие корректности, вычислив слабейшее предусловие:

```
{true}
y := 0;
z := a;
y := y + x;
z := z * x;
y := y * z;
z := z / a;
z := z * b;
z := z + c;
y := y + z
{y = a·x² + b·x + c}
```

9. Докажите следующее условие корректности:

```
{a > 0 ∧ b > 0 ∧ a ≠ b}
x := a;
y := b;
if x > y then
  x := x − y
else
  y := y − x
end
{a > 0 ∧ b > 0 ∧ НОД(x, y) = НОД(a, b)}
```

10. Предложите инварианты циклов для двух реализаций алгоритма Евклида из лекции 1 (программы P и программы Q).

11. Докажите частичную корректность двух реализаций алгоритма Евклида из лекции 1 (программы P и программы Q).

12. Напишите пред- и постусловие для программ сортировки числовых массивов. Докажите частичную корректность программы, реализующей «метод пузырька».

13. Выполните верификацию следующей программы для заданных пред- и постусловий:

```
{a ≥ 0}
x := a;
n := 1;
y := 0;
while x ≠ 0 do
  y := y + n;
  n := n + 2;
  x := x - 1
end
{y = a²}
```

14. Выполните верификацию следующей программы для заданных пред- и постусловий:

```
{true}
x := a;
n := 0;
while x ≠ 0 do
  x := x & (x - 1);
  n := n + 1
end
{n = count(a)}
```

Здесь & — операция побитового И, а $count(x)$ — функция, возвращающая число единиц в двоичном представлении числа x. Определите формально предметную область (для определенности можете считать, что переменные принимают целочисленные значения из отрезка $[0,2^{32}-1]$).

15. Выполните верификацию следующей программы для заданных пред- и постусловий:

```
{a > 1}
i := a - 1;
x := 1;
while i > 0 do
  if a % i = 0 then
    k := 0;
    j := i - 1;
    while j > 1 do
      if i % j = 0 then
        k := k + 1
      end;
      j := j - 1
    end;
    if k = 0 then
      x := i;
      i := 1
    end
  end;
  i := i - 1
end
{x = maxPrimeFactor(a)}
```

Здесь $maxPrimeFactor(a)$ — максимальный простой делитель целого числа $a > 1$.

Лекция 4. Инструменты дедуктивной верификации программ

Если ваш единственный инструмент — молоток, то каждая проблема становится похожей на гвоздь.

А. Маслоу

Рассматриваются элементы языка ACSL (ANSI/ISO C Specification Language), предназначенного для формальной спецификации (аннотирования) программ на языке C. Делается обзор платформы статического анализа Frama-C (Framework for Modular Analysis of C programs) и модуля Jessie, автоматизирующего дедуктивную верификацию C-программ на соответствие ACSL-спецификациям. Изложение базируется на примерах (подробное описание языка ACSL и основанных на нем инструментов доступно в соответствующей документации). Описанные средства могут использоваться в повседневной программистской практике.

4.1. Введение в язык ACSL

До настоящего момента мы касались только теоретических аспектов дедуктивной верификации. Настало время перейти к реально применяемым средствам спецификации и анализа программ. Эта лекция — преамбула к практическим занятиям; в ней рассматриваются язык ACSL (ANSI/ISO C Specification Language) [Bau10] и инструмент Frama-C (Framework for Modular Analysis of C programs) [FraC].

Язык ACSL предназначен для формальной спецификации (*аннотирования*) программ, написанных на языке C [Ker88]. Язык включает средства для определения пред- и постусловий функций, утверждений, инвариантов циклов и других элементов формальных спецификаций — так называемых *аннотаций*. Аннотации ACSL делятся на два типа: *внешние* (*global annotations*) и *внутренние* (*statement annotations*). К первому типу относятся:

- *контракты функций* — пред- и постусловия функций программы или библиотеки;
- *глобальные инварианты* — ограничения целостности данных программы;
- *инварианты типов данных* — ограничения на типы данных (так называемые *ограничения подтипов*);

- *логические спецификации* — предметные области (функциональные символы и описывающие их аксиомы).

К второму типу относятся:

- *утверждения* — условия, истинность которых должна обеспечиваться в определенных точках программы;
- *аннотации циклов* — свойства циклов, включая инварианты;
- *контракты операторов* — пред- и постусловия операторов программы или блоков операторов;
- *вспомогательный код* — действия, модифицирующие внутренние переменные спецификации.

4.1.1. Базовые сведения о языке

ACSL-спецификации пишутся непосредственно в исходном коде программы в виде комментариев, начинающихся с символов `/*@` или `//@`[31]. Лексика языка в основном заимствована из C. Наиболее важные отличия состоят в следующем. Во-первых, идентификаторы можно начинать с обратного слеша — `\`. Во-вторых, в языке расширен состав операций: добавлены логические связки импликации и эквивалентности (`==>` и `<==>`), кванторы существования и всеобщности (`\exists` и `\forall`) и некоторые другие конструкции. В-третьих, ACSL имеет большое число встроенных функций: `\min`, `\max`, `\abs` и т.д.

Операции языка ACSL резюмированы в таблице 4.1.

Таблица 4.1. Основные операции языка ACSL

Конструкция языка		Описание конструкции
Логические константы (значения истинности)		
`\true`		истина
`\false`		ложь
Логические связки и побитовые операции (в порядке уменьшения приоритета)		
!	~	отрицание
&&	&	конъюнкция
^^	^	исключающее ИЛИ

[31] Первым символом комментария-аннотации обязательно должен быть `@`: при наличии пробела между `/*` или `//` и `@` текст комментария будет игнорироваться (как это происходит с обычным комментарием).

`||`	`|`	дизъюнкция
`==>`	`-->`	импликация
`<==>`	`<-->`	эквивалентность
Кванторы существования и всеобщности		
`\exists` $T_1\ x_1, \ldots, T_n\ x_n;\ B$		квантор существования
`\forall` $T_1\ x_1, \ldots, T_n\ x_n;\ B$		квантор всеобщности
Здесь T_i — типы данных, x_i — связанные переменные, B — логическая формула		
Конструкции связывания переменных и именования выражений		
`\let` $x\ =\ e;\ E$		связывание переменных
$id: E$		именование выражений
Здесь x — переменная, e и E — выражения (термы или логические формулы), id — имя (идентификатор)		
Операции модификации структур и массивов		
$\{s\ \mathbf{for}\ .field\ =\ e\}$		модификация структуры
$\{a\ \mathbf{for}\ [i]\ =\ e\}$		модификация массива
Здесь s и a — переменные соответствующих типов, $field$ — имя поля, i — индекс, e — выражение		

В ACSL возможна сокращенная форма записи («синтаксический сахар») для нескольких подряд идущих операций сравнения. Например, язык позволяет вместо $0 <= r \ \&\& \ r < b$ писать $0 <= r < b$. Общее правило такое: запись

$$e_1 \gtrless_1 e_2 \gtrless_2 e_3 \gtrless_3 \ \ldots \gtrless_{n-1} e_n,$$

где $\gtrless_i$ — символы операций сравнения, имеющие одинаковую «ориентацию» (либо они все из множества $\{==, <=, <\}$, либо — из множества $\{==, >=, >\}$), эквивалентна записи

$$(e_1 \gtrless_1 e_2) \ \&\& \ (e_2 \gtrless_2 e_3) \ \&\& \ \ldots \ \&\& \ (e_{n-1} \gtrless_{n-1} e_n).$$

Приоритеты операций ACSL согласованы с приоритетами, принятыми в C. Побитовые операции имеют больший приоритет по сравнению с логическими. Приоритет кванторов существования и всеобщности, также как и конструкций связывания переменных и именования выражений, ниже приоритета условного оператора (тернарного оператора ?:), при этом конструкции `\forall`, `\exists` и `\let` имеют одинаковый приоритет, больший приоритета конструкции $id: E$.

4.1.2. Типы данных

Система типов данных ACSL включает как встроенные типы C, так называемые *машинные типы*, так и *математические типы*: **boolean**, **integer** и **real**. Считается, что машинные типы являются подтипами математических типов: типы **char**, **short**, **int**, **long** (знаковые и

беззнаковые варианты) — подтипы **integer**, а типы **float** и **double** — подтипы **real**[32]. Значения типа **integer** неявно приводятся к значениям типа **boolean**, однако для преобразования из **boolean** в **integer** нужно использовать оператор приведения типов. Допускается преобразование значений математических типов в значения машинных типов: результат — значение, равное исходному по модулю $2^{8 \cdot \mathbf{sizeof}(T)}$, где T — целевой тип. Побитовые операции определены и для значений типа **integer** (их можно рассматривать как бесконечные двоичные последовательности в дополнительном коде).

Из базовых типов данных в ACSL можно строить составные типы: *структуры* и *массивы*. Значения составных типов можно сравнивать, использовать в качестве параметров функций и т.д. При декларации массива указывается его длина; узнать длину массива можно с помощью встроенной функции $\backslash\mathrm{length}$. В остальном все похоже на C.

Пример 4.1

Рассмотрим примеры определения и использования составных типов данных [Bau10]. Обратите внимание: внутри аннотаций можно использовать комментарии; они начинаются с двойного слеша — `//`.

```
//@ // определение составных типов данных
//@ type point = struct { real x; real y; };
//@ type triangle = point[3];

/*@ // логические спецификации (фрагмент описания предметной области)
  @ logic point origin = { .x = 0.0, .y = 0.0 };
  @*/
```

□

4.1.3. Контракты функций

Пред- и постусловия функций задаются с помощью *клауз* **requires** и **ensures** соответственно. Для одной функции можно определить несколько (ноль и более) клауз одного типа: предусловие образуется путем конъюнкции всех условий, указанных в **requires**; постусловие — путем конъюнкции всех условий, указанных в **ensures**. В постусловии, т.е. в клаузах **ensures**, можно использовать следующие конструкции:

- $\backslash\mathrm{old}(e)$ — значение выражения *e* до вызова специфицируемой функции;

[32] Последнее справедливо с оговоркой: специальные значения «*не число*» (NaN, Not a Number) и $\pm\infty$ в типе **real** не представимы.

- \result — результат функции, если тип возвращаемого значения отличен от **void**.

Если функция меняет состояние памяти (например, присваивает что-то в глобальные переменные), модифицируемые *блоки памяти* должны быть явно указаны в клаузах **assigns**. Блок памяти можно задать как с помощью L-выражений языка C, так и используя специальные конструкции ACSL, например, $x[0..n-1]$ или $*\big(x+(0..n-1)\big)$ – обе записи обозначают область, образованную n элементами массива x. Если функция не модифицирует память, в **assigns** следует указать \nothing. В языке есть встроенный предикат \valid, проверяющий доступность (выделенность) блока памяти для заданного адреса или диапазона адресов, например, $\valid(x)$ или $\valid\big(x+(0..n-1)\big)$.

В простейшем случае контракт функции выглядит следующим образом:

```
/*@ requires φ1; (requires φ2; ...) // φ1 && φ2 && ...
  @ assigns  L1; (assigns  L2; ...) // L1,    L2,    ...
  @ ensures  ψ1; (ensures  ψ2; ...) // ψ1 && ψ2 && ...
  @*/
```

Пример 4.2

Ниже приведена сигнатура функции целочисленного деления $idiv$ на языке C и спецификация этой функции на языке ACSL. Функция имеет три параметра: a — делимое, b — делитель, r — указатель на переменную, в которую записывается остаток от деления; результатом функции является частное от деления.

```
/*@ // предусловие функции
  @ requires a >= 0;     // условие на делимое
  @ requires b > 0;      // условие на делитель
  @ requires \valid(r); // доступность блока памяти
  @ // модифицируемый блок памяти (запись остатка от деления)
  @ assigns *r;
  @ // постусловие функции
  @ ensures \let q = \result; a == q * b + *r; // необходимое условие
  @ ensures 0 <= *r < b;                       // условие на остаток
  @*/
int idiv(int a, int b, int *r);
```

□

Язык ACSL поддерживает декомпозицию контрактов функций на так называемые *ветви функциональности* (*named behaviors*). Каждая ветвь функциональности описывает определенный вариант поведения функции: некорректные входные данные, внутренняя ошибка, нормальное исполнение и т.п. В случае если у функции есть разные варианты поведения,

имеет смысл представить ее спецификацию в следующей форме (для наглядности здесь приведены две ветви).

```
/*@ requires φ;      // общее предусловие функции
  @ assigns  L;      // память, модифицируемая во всех ветвях
  @ ensures  ψ;      // общее постусловие функции
  @
  @ behavior b1:     // ветвь функциональности b1
  @   assumes  A1;   // условие, идентифицирующее ветвь
  @   requires φ1;   // предусловие ветви
  @   assigns  L1;   // память, модифицируемая в ветви
  @   ensures  ψ1;   // постусловие ветви
  @
  @ behavior b2:     // ветвь функциональности b2
  @   assumes  A2;
  @   requires φ2;
  @   assigns  L2;
  @   ensures  ψ2;
  @*/
```

Приведенный контракт эквивалентен следующему (если пренебречь деталями, связанными с модификацией разных блоков памяти в разных ветвях функциональности).

```
/*@ requires φ && (A1 ==> φ1) && (A2 ==> φ2);
  @ assigns  L, L1, L2;
  @ ensures  ψ && (\old(A1) ==> ψ1) && (\old(A2) ==> ψ2);
  @*/
```

Пример 4.3

Специфицируем функцию дихотомического поиска в упорядоченном числовом массиве с помощью ветвей функциональности [Bau10]. У функции три параметра: x — массив, в котором осуществляется поиск; n — длина массива; v — искомое значение. Результатом функции является индекс найденного элемента или -1, если элемент не найден.

```
/*@ requires n >= 0 && \valid(x + (0..n-1));
  @ assigns  \nothing;
  @ ensures  -1 <= \result <= n-1;
  @
  @ behavior success:
  @   ensures \result >= 0 ==> x[\result] == v;
  @
  @ behavior failure:
  @   assumes \forall integer i, integer j;
  @     0 <= i < j <= n-1 ==> x[i] <= x[j];
  @   ensures \result == -1 ==>
  @     \forall integer i; 0 <= i <= n-1 ==> x[i] != v;
  @*/
int bsearch(double x[], int n, double v);
```

Обратите внимание: упорядоченность массива требуется не в предусловии, а в условии ветви $failure$: если функция возвратила допустимый индекс, гарантируется, что по этому индексу находится искомый элемент (неважно, отсортирован массив или нет); если же массив не отсортирован, не факт, что элемент будет найден, даже если он там есть.

□

ACSL допускает *неполные спецификации* (в которых дизъюнкция условий всех ветвей не покрывает полностью предусловие функции) и *спецификации с перекрывающимися ветвями* (в которых условия разных ветвей могут быть одновременно истинными). В примере выше спецификация полная, но с перекрывающимися ветвями.

Для дополнительного контроля полноты и разделимости ветвей функциональности можно использовать специальные клаузы: **complete behaviors** и **disjoint behaviors**.

4.1.4. Утверждения и аннотации циклов

Утверждения (assertions) задают ограничения на состояние, которые должны быть истинными в определенных точках программы. В ACSL утверждения определяются с помощью ключевого слова **assert**, после которого указывается ограничение:

```
/*@ assert B; */
```

Можно задать утверждение, актуальное только для некоторых ветвей функциональности:

```
/*@ for b1, ..., bn: assert B; */
```

Напомним: *инвариантом цикла* называется условие, истинность которого не нарушается при исполнении тела цикла. Для определения инвариантов в языке ACSL применяется клауза **loop invariant**:

```
/*@ loop invariant φ; // инвариант цикла
  @ loop assigns   L; // модифицируемая в цикле память
  @*/
```

Есть возможность определить инвариант для избранных ветвей функциональности:

```
/*@ for b1, ..., bn: loop invariant φ; */
```

Язык программирования C, очевидно, сложнее языка while, используемого нами для разъяснения основных концепций дедуктивной верификации: в C есть несколько операторов

циклов (**while**, **do** – **while**, **for**) и допустимы условия с побочным эффектом. Клаузы **loop invariant** φ и **loop assigns** L интерпретируются следующим образом:

1. условие φ истинно перед входом в цикл (для цикла **for** — после инициализации);
2. истинность φ сохраняется после исполнения тела цикла:
    a. для **while**(B) P истинность φ сохраняется после исполнения $B; P$;
    b. для **for**$(I; B;\ U)$ P — после исполнения $B; P; U$;
    c. для **do** P **while**(B) — после исполнения $P; B$;
3. память вне областей L остается неизменной.

Пример 4.4

Приведем примеры функций на языке C и их аннотаций на ACSL. Первый пример — функция целочисленного деления (ее контракт приводился выше).

```
int idiv(int a, int b, int *r) {
  int q = 0;
  int p = a;
  /*@ loop invariant (a == q * b + p) && (0 <= p <= a);
    @ loop assigns q, p;
    @*/
  while(p <= b) {
    q++;
    p -= b;
  }
  /*@ assert (a == q * b + p) && (0 <= p < b); */
  *r = p;
  return q;
}
```

□

Пример 4.5

Второй пример — функция дихотомического поиска в упорядоченном массиве [Bau10].

```
int bsearch(double x[], int n, double v) {
  int l = 0, u = n - 1;
  /*@ loop invariant 0 <= l && u <= n - 1;
    @ for failure: loop invariant
    @   \forall integer k;
    @     (0 <= k <= n - 1 && x[k] == v) ==> l <= k <= u;
    @*/
  while(l <= u) {
    int m = (l + u)/2;
    /*@ assert 0 <= l <= m <= u <= n - 1; */
        if(x[m] < v) l = m + 1;
    else if(x[m] > v) u = m - 1;
```

```
        else return m;
      }
      return -1;
    }
```

□

4.2. Обзор платформы Frama-C

Ниже делается краткий обзор средств автоматизации проверки С-программ на соответствие ACSL-спецификациям: модульной платформы статического анализа Frama-C [FraC] и модуля (plugin) Jessie [Jess], [Krak], [Mar10], отвечающего за дедуктивную верификацию. Указанные инструменты потребуются для выполнения практических заданий, однако они могут использоваться вами и в повседневной программистской практике.

4.2.1. Платформа Frama-C

Платформа Frama-C предназначена для статического анализа программ на языке С. Под *статическим анализом* понимается вычисление характеристик программ без их исполнения на компьютере, т.е. путем вывода из исходного или бинарного кода. Платформа разработана языке программирования OCaml [Min13] и распространяется с открытым исходным кодом по лицензии GNU LGPL v2. К созданию Frama-C причастны две французские организации: CEA-LIST (Laboratory for Integration of Systems and Technology) и INRIA (Institut National de Recherche en Informatique et en Automatique).

Платформа Frama-C состоит из ядра и внешних модулей. Ядро управляет анализом программ и хранит информацию, полученную в ходе анализа. Есть несколько предопределенных модулей, но что важно, можно разрабатывать новые анализаторы на базе имеющихся (взаимодействие между модулями осуществляется через интерфейсы ядра). Тремя основополагающими модулями Frama-C являются:

- *Value analysis* – вычисление множеств возможных значений переменных программы с помощью техник *абстрактной интерпретации* (см. лекцию 7);
- *Jessie* и *Wp* – дедуктивная верификации функций на соответствие контрактам с использованием техник на основе слабейшего предусловия (см. лекции 2 и 3).

Для выполнения заданий первого практикума нами будет использован модуль Jessie.

4.2.2. Модуль Jessie и платформа Why3

Модуль Jessie предназначен для автоматизации верификации С-программ, снабженных ACSL-аннотациями. Модуль является препроцессором (front-end) для специализированной платформы дедуктивной верификации Why3 [Why3], совместно разрабатываемой институтом INRIA, лабораторией LRI (Laboratoire de Recherche en Informatique) и исследовательским центром CNRS (Centre National de la Recherche Scientifique). Отметим следующую техническую особенность – Jessie реализован с помощью Why версии 2, но подготавливает данные для Why версии 3 — для установки Jessie требуются обе версии Why [Krak][33].

Задачами связки Jessie-Why3 являются:

1. построение внутреннего представления аннотированной программы;
2. генерация условий верификации (см. лекцию 5);
3. преобразование условий верификации к формату внешних инструментов автоматизированного доказательства теорем (см. лекцию 8).

Модуль Jessie решает только первую задачу; за оставшиеся две отвечает Why3.

Платформа Why3 базируется на специальном языке WhyML, который может использоваться сам по себе (для разработки программ), а может применяться для промежуточного представления программ на других языках (C, Java, Ada). В платформе имеется библиотека предметных областей: алгебра логики, целочисленная и вещественная арифметика; поддерживаются основные типы данных и связанные с ними теории: массивы, очереди, отображения и т.п. Why3 использует внешние инструменты автоматизированного доказательства теорем (provers): Alt-Ergo, CVC4, Yices, Z3, Coq, PVS (последние два средства являются интерактивными, остальные — автоматическими).

Запуск Frama-C с модулем Jessie осуществляется следующей командой:

```
frama-c -jessie имя-файла
```

[33] Скорее всего, указанная особенность Jessie будет устранена в новых версиях.

В результате появится графический интерфейс, отображающий результат преобразования программы в WhyML, проверяемые функциональные свойства и свойства безопасности (завершимость программы, доступность используемой памяти, отсутствие переполнений и делений на ноль), установленные инструменты доказательства теорем и настройки.

Для того чтобы система Why3 «подцепила» имеющиеся пруверы, нужно выполнить следующую команду:

```
why3 config --detect-provers
```

Более подробную информацию об инструментах Frama-C, Jessie и Why3 можно найти по указанным выше ссылкам.

4.3. Вопросы и упражнения

1. Укажите основные конструкции языка ACSL.
2. С помощью каких средств языка ACSL определяются программные контракты и инварианты циклов?
3. Что такое ветвь функциональности? Дайте определения полного множества ветвей и разделимого множества ветвей.
4. Определите на языке ACSL контракты следующих функций:
   a. нахождение НОД двух натуральных чисел;
   b. сортировка числового массива.
5. Реализуйте на языке C следующие функции:
   a. нахождение суммы элементов числового массива;
   b. сортировка числового массива «методом пузырька».

   Для указанных функций определите на языке ACSL инварианты циклов.
6. Установите следующий инструментарий: платформу Frama-C, модуль Jessie (включая Why2 и Why3) и SMT-решатели (например, Z3 и CVC4).
7. Докажите с использованием Frama-C корректность следующих функций:
   a. целочисленное деление (функция $idiv$);

b. дихотомический поиск в упорядоченном массиве (функция $bsearch$);

8. Проверифицируйте с помощью Frama-C ваши реализации следующих функций:

   a. нахождение суммы элементов числового массива;

   b. сортировка числового массива «методом пузырька».

9. Специфицируйте на языке ACSL и проверифицируйте с помощью Frama-C следующие функции:

   a. нахождение максимального делителя натурального числа, отличного от самого числа:

```
int maxDivisor(int n) {
  int m = 1;
  for (int i = n - 1; i > 0; i--) {
    if (n % i == 0) {
      if (i > m) {
        m = i;
      }
    }
  }
  return m;
}
```

   b. нахождение максимального простого делителя натурального числа:

```
int maxPrimeFactor(int n) {
  int min = 1;
  do {
    n /= min;
    min = n;
    for (int i = n - 1; i > 1; i--) {
      if (i * i <= n && n % i == 0) {
        min = i;
      }
    }
  } while (min < n);
  return n;
}
```

   c. вычисление целой части квадратного корня целого неотрицательного числа:

```
int isqrt(int n) {
  int a = 0;
  int b = 1;
  int c = 1;
  for(; b <= n; a++) {
    c += 2;
    b += c;
  }
  return a;
```

```
}
```

d. поиск наиболее дешевого варианта с пользой не менее заданной:

```
int minCostForGivenValue(int n, int *cost, int *value, int k) {
  int r = -1;
  for (int i = 0; i < n; i++) {
    if (value[i] >= k) {
      if (r == -1) {
        r = i;
      } else if (cost[i] < cost[r]) {
        r = i;
      }
    }
  }
  return r;
}
```

Лекция 5. Метод индуктивных утверждений

Индукция и дедукция связаны между собой столь же необходимым образом, как синтез и анализ. Вместо того чтобы односторонне превозносить одну из них до небес за счет другой, надо стараться применять каждую из них на своем месте.

Ф. Энгельс.
Диалектика природы

Рассматривается один из первых подходов к формальной верификации программ — метод Флойда. Метод состоит из двух частей: метод индуктивных утверждений, предназначенный для доказательства частичной корректности, т.е. корректности без учета завершимости, и метод фундированных множеств, используемый для доказательства завершимости и полной корректности (вторая часть метода разбирается в лекции 6). Метод Флойда имеет дело с блок-схемами, что позволяет анализировать программы с общим видом потока управления, в том числе, не отвечающие принципам структурного программирования[34]. Метод состоит в рассечении циклов, сопоставлении точкам сечения индуктивных утверждений (инвариантов), формулировке условий верификации и их доказательстве.

5.1. Синтаксис блок-схем

Метод, рассматриваемый в этой лекции, был предложен Робертом Флойдом (Robert W Floyd, 1936-2001) в далеком 1967 г. [Flo67]. Это классический метод, который, несмотря на свою «древность», используется в современных инструментах дедуктивной верификации, например, в модуле Jessie платформы Frama-C (см. лекцию 4).

Метод Флойда оперирует с блок-схемами. Как и при определении языка while будем считать, что задана сигнатура Σ, т.е. множество функциональных и предикатных символов вместе с их арностью, и множество переменных V. Для удобства разобьем множество V на три непересекающиеся подмножества [Буз14]:

- X — *входные* переменные;
- Y — *внутренние* переменные;
- Z — *выходные* переменные.

[34] Методология структурного программирования была предложена в 1970-х гг. [Dah72], а Роберт Флойд (Robert Floyd, 1936-2001) разработал свой метод в конце 1960-х [Flo67].

Входные переменные содержат исходные данные и не меняются во время исполнения программы; внутренние переменные используются для хранения промежуточных результатов; выходные переменные предназначены для выдачи полученного решения.

Введем обозначение: запись $(y_1, \dots, y_n) \coloneqq (t_1, \dots, t_n)$, или в векторной форме $\vec{y} \coloneqq \vec{t}$, где $y_1, \dots, y_n$ — разные переменные, а $t_1, \dots, t_n$ — термы, равносильна традиционной последовательной записи:

$$y_1' \coloneqq t_1;\ \dots\ ; y_n' \coloneqq t_n;$$

$$y_1 \coloneqq y_1';\ \dots\ ; y_n \coloneqq y_n'.$$

Сначала вычисляются значения всех термов, и лишь потом исполняются присваивания. Эту запись можно трактовать как *параллельное присваивание*: значений термов вычисляются параллельно и независимо друг от друга и параллельно присваиваются переменным (важно лишь, чтобы присваивания стартовали после вычисления значений всех термов).

Блок-схемы состоят из операторов следующих типов:

1. *начальный оператор* $\boxed{\textbf{start}: \vec{y} \coloneqq \vec{t}}$, где $\vec{y}$ — кортеж из переменных множества Y, а $\vec{t}$ — кортеж из термов над множеством X;
2. *оператор присваивания* $\boxed{\textbf{assign}: \vec{y} \coloneqq \vec{t}}$, где $\vec{y}$ — кортеж из переменных множества Y, а $\vec{t}$ — кортеж из термов над множеством $X \cup Y$;
3. *условный оператор* $\boxed{\textbf{test}: B}$, где B — логическая формула над множеством $X \cup Y$;
4. *оператор соединения* $\boxed{\textbf{join}}$;
5. *завершающий оператор* $\boxed{\textbf{halt}: \vec{z} \coloneqq \vec{t}}$, где $\vec{z}$ — кортеж из переменных множества Z, а $\vec{t}$ — кортеж из термов над множеством $X \cup Y$.

Графическое изображение операторов (точнее, узлов блок-схем, помеченных операторами) показано на рис. 5.1.

Обозначим множество всех возможных операторов над сигнатурой Σ и множеством переменных $V = X \cup Y \cup Z$ символом Op. *Блок-схемой* называется размеченный ориентированный граф $\langle N, E, L \rangle$, где N — множество узлов, $E \subseteq N \times \{true, false, \epsilon\} \times N$ — множество дуг с пометками, $L: N \to Op$ — функция разметки узлов операторами.

Для краткости мы будем называть узлы блок-схемы операторами. Например, фраза «в блок-схеме $\langle N, E, L\rangle$ присутствует ровно один начальный оператор» на самом деле означает, что существует единственный узел $n \in N$, такой что оператор $L(n)$ имеет тип **start**.

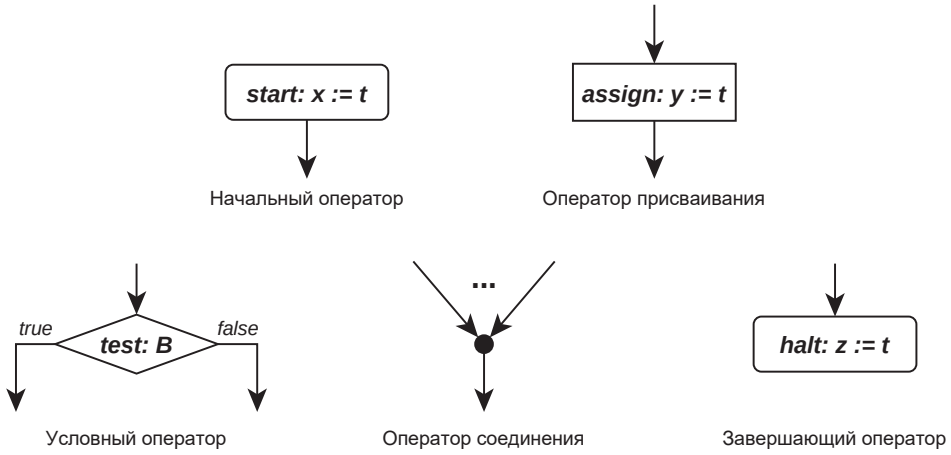


Рисунок 5.1. Графическое представление операторов блок-схем

Блок-схема называется *корректно-определенной*, если она удовлетворяет следующим ограничениям [Буз14]:

1. в блок-схеме присутствует ровно один начальный оператор и по крайней мере один завершающий;
2. любой оператор блок-схемы находится на пути от начального оператора к некоторому завершающему;
3. число дуг, исходящих из каждого оператора блок-схемы, и их разметка соответствуют типу оператора:
    - из начального оператора, оператора присваивания и оператора соединения исходит одна дуга: она имеет пометку ϵ;
    - из условного оператора исходят две дуги: одна из них имеет пометку $true$, другая — $false$;
    - из завершающего оператора не исходит ни одной дуги;
4. число дуг, входящих в каждый оператор блок-схемы, соответствует типу оператора:
    - в начальный оператор не входит ни одной дуги;

- в оператор присваивания, условный оператор и завершающий оператор входит одна дуга;
- в оператор соединения входит не менее одной дуги.

В дальнейшем мы будем рассматривать только корректно-определенные блок-схемы.

Заметим, что для каждого узла $n \in N$ и пометки $l \in \{true, false, \epsilon\}$ в блок-схеме существует не более одной дуги $(n, l, n') \in E$. Если такая дуга существует (оператор $L(n)$ не является завершающим, а пометка l является допустимой для этого типа операторов), узел n' будем называть *последователем* узла n по пометке l и обозначать как $succ(n, l)$; запись $succ(n)$ является сокращением $succ(n, \epsilon)$.

Несколько слов о графическом представлении блок-схем. При изображении блок-схем будем опускать «пустые» пометки ϵ, как и ключевые слова, задающие типы операторов (тип однозначно определяется телом оператора, а также числом входящих и исходящих дуг).

Пример 5.1

Рассмотрим блок-схему, реализующую целочисленное деление. Множество ее переменных устроено следующим образом: $X = \{a, b\}$, $Y = \{q', r'\}$ и $Z = \{q, r\}$. Графическое изображение блок-схемы представлено на рис. 5.2.

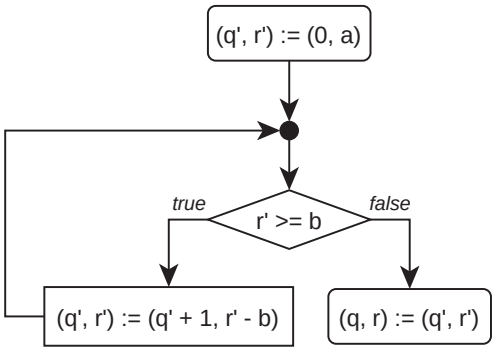


Рисунок 5.2. Блок-схема целочисленного деления

□

5.2. Семантика блок-схем

Обозначим *домен* (множество возможных значений) переменной x символом D_x. Как и прежде, *состоянием* программы (блок-схемы) называется отображение, которое каждой переменной x ставит в соответствие ее значение — элемент домена D_x. Множество всех возможных состояний обозначим символом S.

Формализуем семантику блок-схем, используя операционный подход (см. лекцию 2). *Конфигурацией* блок-схемы $\langle N, E, L\rangle$ называется пара $\langle n, s\rangle \in (N \cup \{n_\omega\}) \times S$, включающая узел блок-схемы (или псевдо-узел n_ω, соответствующий завершению работы) и состояние. Определим отношение переходов на множестве конфигураций, как показано в таблице 5.1.

Таблица 5.1. Отношение переходов на множестве конфигураций блок-схемы

(1)	$L(n) = \boxed{\textbf{start}: \vec{y} := \vec{t}}$	$\langle n, s\rangle \to \langle succ(n), s\big[\vec{y} := s(\vec{t})\big]\rangle$
(2)	$L(n) = \boxed{\textbf{assign}: \vec{y} := \vec{t}}$	$\langle n, s\rangle \to \langle succ(n), s\big[\vec{y} := s(\vec{t})\big]\rangle$
(3)	$L(n) = \boxed{\textbf{test}: B}$	$\langle n, s\rangle \to \langle succ(n, true), s\rangle$, если $s \vDash B$
(4)	$L(n) = \boxed{\textbf{test}: B}$	$\langle n, s\rangle \to \langle succ(n, false), s\rangle$, если $s \vDash \neg B$
(5)	$L(n) = \boxed{\textbf{join}}$	$\langle n, s\rangle \to \langle succ(n), s\rangle$
(6)	$L(n) = \boxed{\textbf{halt}: \vec{z} := \vec{t}}$	$\langle n, s\rangle \to \langle n_\omega, s\big[\vec{z} := s(\vec{t})\big]\rangle$

Напомним: $s(t)$ — значение терма (кортежа термов) t в состоянии s; $s[y := v]$ — состояние, полученное из состояния s, сопоставлением переменной (кортежа переменных) y значения (кортежа значений) v без изменения означивания других переменных.

Для блок-схем, как и для while-программ, можно определить понятие вычисления (см. лекцию 2). *Последовательностью переходов* блок-схемы P в состоянии s называется конечная или бесконечная последовательность конфигураций $\{\langle n_i, s_i\rangle\}_{i\geq 0}$, такая что:

1. n_0 — начальный узел P;
2. $s_0 = s$;
3. $\langle n_i, s_i\rangle \to \langle n_{i+1}, s_{i+1}\rangle$, для всех $i \geq 0$.

Вычислением блок-схемы P в состоянии s называется последовательность переходов, которая не может быть продолжена. Говорят, что вычисление блок-схемы завершается в состоянии s', если оно конечно и его последняя конфигурация имеет вид $\langle n_\omega, s'\rangle$. Говорят, что

блок-схема P *может зациклиться* в состоянии s, если существует бесконечное вычисление P в s.

Семантика блок-схем определяется следующим образом (как и раньше, $\to^*$ обозначает транзитивное и рефлексивное замыкание отношения переходов $\to$):

$$M[\![P]\!](s) = \{\, s' \mid \langle n_0, s\rangle \to^* \langle n_\omega, s'\rangle \,\} \cup \{\, \omega \mid P \text{ может зациклиться в } s \,\}.$$

Для каждой блок-схемы P функция $M[\![P]\!]$ является однозначной, поэтому для краткости вместо $M[\![P]\!](s) = \{s'\}$ будем писать $M[\![P]\!](s) = s'$.

5.3. Метод индуктивных утверждений

Метод индуктивных утверждений предназначен для доказательства частичной корректности блок-схем. Напомним: блок-схема P называется *частично корректной* относительно предусловия φ и постусловия ψ (обозначается: $\vDash \{\varphi\}P\{\psi\}$), если для любого состояния s, такого что $s \vDash \varphi$ и $M[\![P]\!](s) \neq \omega$, имеет место $M[\![P]\!](s) \vDash \psi$.

Пусть P — блок-схема, $\pi = \{(n_i, l_i, n_{i+1})\}_{i=0}^{k-1}$ — путь в P, $N_\pi = \{n_i\}_{i=0}^{k}$ — последовательность узлов, лежащих вдоль пути π (для удобства N_π также будем называть путем). Заметим, что путь не обязан начинаться со **start** и оканчиваться **halt** — это может быть произвольная последовательность смежных дуг.

Определим формулу φ_π и подстановку θ_π:

- формула φ_π определяет, каким должно быть состояние блок-схемы, чтобы, начиная с узла n_0, вычисление пошло по пути π;
- подстановка θ_π определяет, как изменится состояние блок-схемы при исполнении всех операторов, лежащих на пути π.

Естественным способом вычисления φ_π и θ_π является *метод обратных подстановок*.

5.3.1. Метод обратных подстановок

Предположим, что последний оператор $L(n_k)$ не является условным оператором. Для каждого $0 \le m \le k+1$ определим $\varphi_\pi^{(m)}$ и $\theta_\pi^{(m)}$ — соответственно формулу и подстановку для пути $N_\pi^{(m)} = \{n_i\}_{i=m}^{k}$. Заметим, что $\varphi_\pi \equiv \varphi_\pi^{(0)}$ и $\theta_\pi \equiv \theta_\pi^{(0)}$. Воспользуемся индукцией по m.

Базис индукции

- $N_\pi^{(k+1)} = \{\}$ — пустой путь;
- $\varphi_\pi^{(k+1)} = true$ — тождественная истина (пустой путь будет пройден при любом начальном состоянии блок-схемы);
- $\theta_\pi^{(k+1)} = []$ — пустая подстановка (прохождение по пустому пути не меняет состояния блок-схемы).

Индуктивный переход

Предположим, что $\varphi_\pi^{(m+1)}$ и $\theta_\pi^{(m+1)}$ определены для некоторого $0 \le m \le k$.

Определим $\varphi_\pi^{(m)}$ и $\theta_\pi^{(m)}$ в зависимости от оператора $L(n_m)$ и пометки l_m (если n_m — последний узел пути, т.е. $m = k$, считаем $l_m = \epsilon$), как показано в таблице 5.2.

Таблица 5.2. Вычисление формулы $\varphi_\pi^{(m)}$ и подстановки $\theta_\pi^{(m)}$

(1)	$L(n_m) = \boxed{\textbf{start}: \vec{y} := \vec{t}}$, если $m = 0$	$\varphi_\pi^{(m)} = \varphi_\pi^{(m+1)}\left[\vec{y} := \vec{t}\right]$	$\theta_\pi^{(m)} = \theta_\pi^{(m+1)} \cdot \left[\vec{y} := \vec{t}\right]$
(2)	$L(n_m) = \boxed{\textbf{assign}: \vec{y} := \vec{t}}$	$\varphi_\pi^{(m)} = \varphi_\pi^{(m+1)}\left[\vec{y} := \vec{t}\right]$	$\theta_\pi^{(m)} = \theta_\pi^{(m+1)} \cdot \left[\vec{y} := \vec{t}\right]$
(3)	$L(n_m) = \boxed{\textbf{test}: B}$ и $l_m = true$	$\varphi_\pi^{(m)} = \varphi_\pi^{(m+1)} \wedge B$	$\theta_\pi^{(m)} = \theta_\pi^{(m+1)}$
(4)	$L(n_m) = \boxed{\textbf{test}: B}$ и $l_m = false$	$\varphi_\pi^{(m)} = \varphi_\pi^{(m+1)} \wedge \neg B$	$\theta_\pi^{(m)} = \theta_\pi^{(m+1)}$
(5)	$L(n_m) = \boxed{\textbf{join}}$	$\varphi_\pi^{(m)} = \varphi_\pi^{(m+1)}$	$\theta_\pi^{(m)} = \theta_\pi^{(m+1)}$
(6)	$L(n_m) = \boxed{\textbf{halt}: \vec{z} := \vec{t}}$	$\varphi_\pi^{(m)} = \varphi_\pi^{(m+1)}\left[\vec{z} := \vec{t}\right]$	$\theta_\pi^{(m)} = \theta_\pi^{(m+1)} \cdot \left[\vec{z} := \vec{t}\right]$

Обратите внимание: путь просматривается от конца к началу, поэтому подстановки применяются в порядке, обратном порядку исполнения операторов; этим обуславливается название метода.

5.3.2. Точки сечения блок-схемы, индуктивные утверждения и условия верификации

Для удобства последующего описания метода Флойда совершим следующие преобразования блок-схемы: расширим множество узлов блок-схемы двумя псевдо-узлами n_α и n_ω; добавим псевдо-дугу $(n_\alpha, \epsilon, n_s)$, где n_s — начальный узел; для каждого завершающего узла n_h добавим псевдо-дугу $(n_h, \epsilon, n_\omega)$. Псевдо-дугу $(n_\alpha, \epsilon, n_s)$ будем называть *начальной*, а псевдо-дуги $(n_h, \epsilon, n_\omega)$ — *завершающими*.

Составим множество C дуг блок-схемы так, чтобы выполнялись следующие условия:

1. множество C содержит начальную и все завершающие псевдо-дуги;
2. каждый цикл в блок-схеме содержит по крайней мере одну дугу из множества C.

Элементы множества C называются *точками сечения*. Точки сечения, отличные от начальных и завершающих псевдо-дуг, называются *внутренними*. Пути между точками сечения[35], не содержащие других точек сечения, называются *базовыми путями*. Базовые пути, исходящие из начальной точки, называются *начальными*; базовые пути, входящие в завершающие точки, — *конечными*; остальные базовые пути называются *внутренними*.

Сопоставим каждой внутренней точке сечения i логическую формулу χ_i, задающую соотношение между значениями переменных при прохождении этой точки. Формула χ_i называется *индуктивным утверждением*. Для начальной точки α положим $\chi_\alpha \equiv \varphi$, где φ — предусловие; для каждой завершающей точки h положим $\chi_h \equiv \psi$, где ψ — постусловие.

Построим для каждого базового пути π (пусть он начинается в точке сечения i и завершается в точке сечения j) так называемое *условие верификации* (*VC*, *Verification Condition*):

$$VC_{ij} \equiv (\chi_i \wedge \varphi_\pi) \to \chi_j \theta_\pi.$$

Это условие выражает следующую мысль: если истинно индуктивное утверждение χ_i и, начиная с точки сечения i, исполнение пойдет по пути π, то в точке сечения j должно быть истинно индуктивное утверждение χ_j.

Пусть нам удалось доказать истинность всех условий верификации. Рассмотрим вычисление блок-схемы в произвольном начальном состоянии, удовлетворяющем предусловию φ. Какая бы точка сечения j не встретилась нам на пути, в текущем состоянии будет истинно индуктивное утверждение χ_j (это следствие условия верификации VC_{ij}, где i — предыдущая пройденная точка сечения). Аналогично, после исполнения одного из завершающих операторов (если это произойдет), будет истинно постусловие ψ. Это и означает частичную корректность блок-схемы.

---

[35] Когда говорится, что путь начинается (оканчивается) в точке сечения (n, l, n'), имеется в виду, что он начинается (оканчивается) с узла n' (узлом n).

Почему утверждения, сопоставленные точкам сечения, называются *индуктивными*? Название происходит от метода *математической индукции*. Согласно этому методу, для доказательства некоторого утверждения в общем виде, например, $\forall n \in \mathbb{N}_0\{\varphi(n)\}$, достаточно сделать следующее:

- проверить истинность утверждения в простейшем случае: $\varphi(0)$;
- показать, что из истинности утверждения в некотором произвольном случае следует его истинность в «следующем» по сложности случае: $\varphi(n) \to \varphi(n+1)$.

Так же и в методе Флойда: утверждения, сопоставленные точкам сечения, должны обеспечивать возможность индуктивного перехода; например, из истинности утверждения перед исполнением тела цикла должна вытекать его истинность после исполнения.

5.3.3. Общая схема метода индуктивных утверждений

Пусть дана блок-схема P и ее спецификация: предусловие φ и постусловие ψ. Метод индуктивных утверждений Флойда состоит из следующих шагов:

1. выбор точек сечения;
2. формулировка индуктивных утверждений для точек сечения;
3. построение условий верификации для базовых путей;
4. доказательство условий верификации.

Если удается доказать истинность всех построенных условий верификации, то блок-схема частично корректна относительно спецификации: $\vDash \{\varphi\}P\{\psi\}$.

Все шаги метода Флойда, за исключением шага 2 и отчасти шага 4, могут быть выполнены автоматически, а вот выбор подходящих индуктивных утверждений требует *понимания* программы и с трудом поддается автоматизации (см. лекцию 7).

Пример 5.2

Докажем частичную корректность блок-схемы, показанной на рис. 5.2, относительно предусловия $\varphi \equiv (a \geq 0) \wedge (b \geq 0)$ и постусловия $\psi \equiv (a = q \cdot b + r) \wedge (0 \leq r < b)$.

Следуя описанному методу, добавим начальную и завершающую псевдо-дуги (A и C) и сопоставим им соответственно φ и ψ. «Рассечем» единственный цикл блок-схемы точкой B, т.е. дугой между оператором соединения и условным оператором (см. рис. 5.3). Для этой точки рассмотрим следующее индуктивное утверждение:

$$\chi \equiv (a = q' \cdot b + r') \wedge (0 \leq r' \leq a) \wedge \varphi.$$

При таком выборе точек сечения определены три базовых пути:

1. AB — путь между точками A и B (начальный базовый путь);
2. BB — путь между точками B и B (внутренний базовый путь);
3. BC — путь между точками B и C (конечный базовый путь).

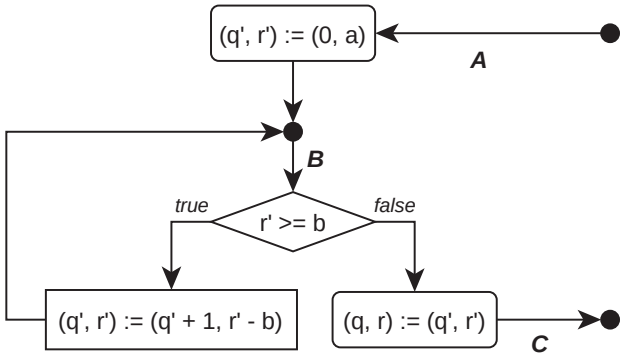


Рисунок 5.3. Точка сечения блок-схемы целочисленного деления

Рассмотрим путь AB:

- $\varphi_{AB} = true;$
- $\theta_{AB} = [q' \coloneqq 0, r' \coloneqq a].$

Имеем:

$$(\varphi \wedge \varphi_{AB}) \equiv \varphi,$$

$$\chi\theta_{AB} \equiv \big((a = q' \cdot b + r') \wedge (0 \leq r' \leq a) \wedge \varphi\big)[q' \coloneqq 0, r' \coloneqq a] \equiv$$
$$(a = 0 \cdot b + a) \wedge (0 \leq a \leq a) \wedge \varphi \equiv \varphi.$$

Очевидно, что $\vDash (\varphi \wedge \varphi_{AB}) \rightarrow \chi\theta_{AB}$.

Рассмотрим путь BB — путь между условным оператором и оператором соединения, проходящий через дугу с пометкой $true$:

- $\varphi_{BB} = (r' \geq b);$
- $\theta_{BB} = [q' \coloneqq q' + 1, r' \coloneqq r' - b].$

Имеем:

$$(\chi \wedge \varphi_{BB}) \equiv (a = q' \cdot b + r') \wedge (0 \leq r' \leq a) \wedge \varphi \wedge (r' \geq b) \equiv$$
$$(a = q' \cdot b + r') \wedge (b \leq r' \leq a) \wedge \varphi,$$

$$\chi\theta_{BB} \equiv \big((a = q' \cdot b + r') \wedge (0 \leq r' \leq a) \wedge \varphi\big)[q' \coloneqq q' + 1, r' \coloneqq r' - b] \equiv$$
$$\big(a = (q' + 1) \cdot b + (r' - b)\big) \wedge (0 \leq (r' - b) \leq a) \wedge \varphi \equiv$$
$$(a = q' \cdot b + r') \wedge \big(b \leq r' \leq (a + b)\big) \wedge \varphi.$$

Истинность импликации $(\chi \wedge \varphi_{BB}) \to \chi\theta_{BB}$ устанавливается легко. В самом деле,

$\vDash \varphi \to (b \geq 0)$, следовательно, $\vDash \big((b \leq r' \leq a) \wedge \varphi\big) \to \Big(\big(b \leq r' \leq (a + b)\big) \wedge \varphi\Big)$.

Наконец, рассмотрим путь BC — путь между условным оператором и оператором соединения, проходящий через дугу с пометкой $false$:

- $\varphi_{BC} = \neg(r' \geq b) \equiv (r' < b)$;
- $\theta_{BC} = [q \coloneqq q', r \coloneqq r']$.

Имеем

$$(\chi \wedge \varphi_{BC}) \equiv (a = q' \cdot b + r') \wedge (0 \leq r' \leq a) \wedge \varphi \wedge (r' < b) \equiv$$
$$(a = q' \cdot b + r') \wedge (0 \leq r' < b) \wedge (r' \leq a) \wedge \varphi,$$
$$\psi\theta_{BC} \equiv \big((a = q \cdot b + r) \wedge (0 \leq r < b)\big)[q \coloneqq q', r \coloneqq r'] \equiv$$
$$(a = q' \cdot b + r') \wedge (0 \leq r' < b).$$

Очевидно, что $\vDash (\chi \wedge \varphi_{BC}) \to \psi\theta_{BC}$.

Таким образом, блок-схема является частично корректной относительно φ и ψ.

□

5.4. Вопросы и упражнения

1. Дайте определение корректно определенной блок-схемы. Определите формально операционную семантику блок-схем.
2. Каким образом выбирается множество точек сечения блок-схемы в методе Флойда? Что понимается под базовыми путями? На какие типы они подразделяются?

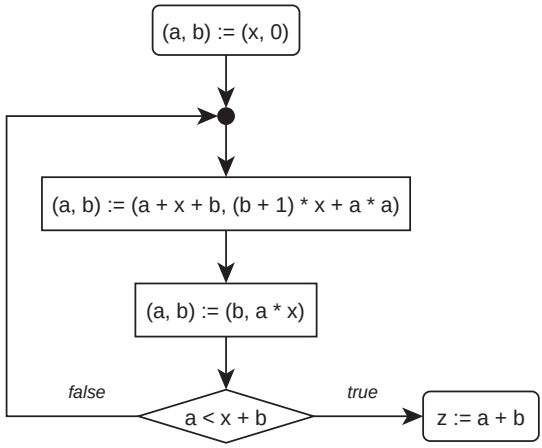
3. Что такое индуктивное утверждение? Соотнесите понятия индуктивного утверждения и инварианта цикла.

4. Перечислите основные шаги метода индуктивных утверждений Флойда. В чем состоит трудность его автоматизации?

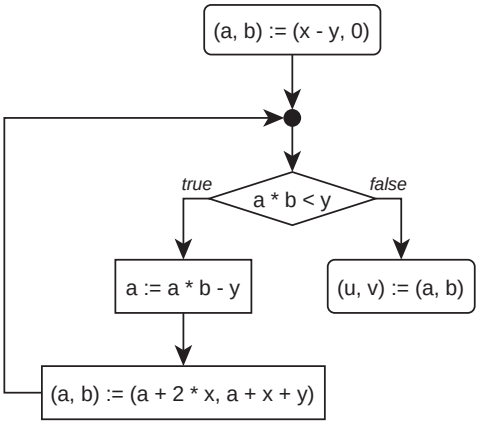
5. Постройте блок-схему, реализующую возведения числа в целую неотрицательную степень. Докажите, используя метод индуктивных утверждений Флойда, частичную корректность построенной блок-схемы.

6. Постройте блок-схему, реализующую алгоритм Евклида. Докажите, используя метод индуктивных утверждений Флойда, частичную корректность построенной блок-схемы.

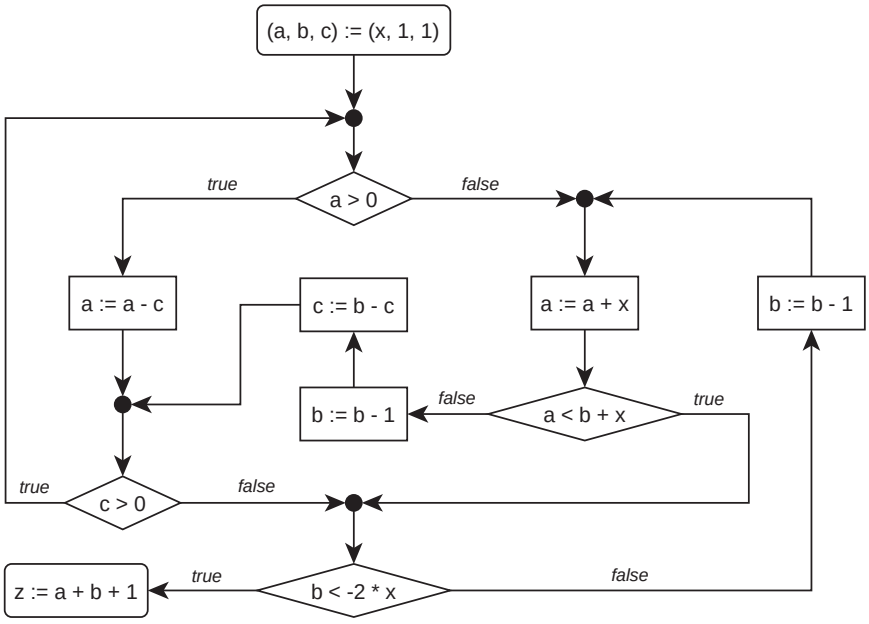
7. Докажите частичную корректность приведенной ниже блок-схемы относительно предусловия $\varphi \equiv (x \geq 0)$ и постусловия $\psi \equiv (z \geq 0)$ [Буз14].



8. Докажите частичную корректность приведенной ниже блок-схемы относительно предусловия $\varphi \equiv (x > y)$ и постусловия $\psi \equiv \big((u - x) = (v - y)\big)$ [Буз14].



9. Докажите частичную корректность приведенной ниже блок-схемы относительно предусловия $\varphi \equiv (x \geq 0)$ и постусловия $\psi \equiv (z = x^2)$ [Буз14].



10. Напишите алгоритм, который для заданного пути строит условие (логическую формулу), гарантирующее проход по этому пути.

11. Напишите алгоритм, который для заданных пути и переменной строит выражение (терм), задающее значение переменной после исполнения операторов пути.

12. Докажите значимость метода индуктивных утверждений Флойда.

Лекция 6. Метод фундированных множеств

The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked.

"Begin at the beginning," the King said gravely, "and go on till you come to the end; then stop."

L. Carroll.
Alice's Adventures in Wonderland

Рассматривается метод фундированных множеств, применяемый в паре с методом индуктивных утверждений для доказательства завершимости и полной корректности программ. Идея метода заключается в сопоставлении каждой точке сечения оценки — функции состояния, значение которой убывает при каждом прохождении через эту точку, но в то же время не может убывать бесконечно долго в силу характера изменений и ограничений, накладываемых соответствующим индуктивным утверждением. Если удается найти такие оценочные функции и доказать условия верификации, программа завершается и, более того, является полностью корректной.

6.1. Завершимость программ и полная корректность

Одной из важнейших составляющих понятия корректности вычислительной программы является *завершимость*, т.е. гарантированный останов за конечное число шагов в любом допустимом начальном состоянии. Основной причиной, по которой программа может не завершиться, является *зацикливание* — вхождение вычислений в так называемый *бесконечный цикл* — цикл, условие выхода из которого никогда не становится истинным. Анализ завершимости программы представляет весьма сложную задачу; чтобы это осознать, достаточно вспомнить, что проблема останова алгоритмически неразрешима [Tur36].

Сразу отметим, что существует широкий класс программ, для которых завершение работы является редким событием, сигнализирующем, скорее, об ошибке, — это так называемые *реагирующие системы* [Кар10]. Они, оправдывая название, реагируют на воздействия окружения, меняя свое состояние и вырабатывая ответные действия. К ним относятся операционные системы, драйверы устройств, реализации телекоммуникационных протоколов, системы управления. Методам формальной спецификации и верификации реагирующих систем посвящена вторая часть пособия (она начинается с лекции 9).

Вспомним определение: программа (блок-схема) *P* называется *полностью корректной* относительно предусловия φ и постусловия ψ (обозначается: $\vDash \langle\varphi\rangle P\langle\psi\rangle$), если для любого состояния s, такого что $s \vDash \varphi$, справедливо следующее:

$$M[\![P]\!](s) \neq \omega \quad \text{и} \quad M[\![P]\!](s) \vDash \psi.$$

Очевидно, что полная корректность программы P относительно предусловия φ и тривиального постусловия $true$ равносильна завершимости P во всех начальных состояниях, удовлетворяющих φ.

Справедливо следующее утверждение: если $\vDash \{\varphi\}P\{\psi\}$ и $\vDash \langle\varphi\rangle P\langle true\rangle$, то $\vDash \langle\varphi\rangle P\langle\psi\rangle$. Иными словами, для доказательства полной корректности программы достаточно доказать ее частичную корректность и завершимость (см. таблицу 6.1).

Таблица 6.1. Правило декомпозиции для доказательства полной корректности

$\dfrac{\{\varphi\}P\{\psi\}, \langle\varphi\rangle P\langle true\rangle}{\langle\varphi\rangle P\langle\psi\rangle}$	**Правило декомпозиции**: из частичной корректности и завершимости программы выводится ее полная корректность

Для доказательства завершимости программ, содержащих циклы, используются два основных метода: *метод фундированных множеств* [Flo67] и *метод счетчиков* [Неп88].

6.2. Метод фундированных множеств

Для изложения первого метода нам потребуются вспомогательные понятия.

Частичным порядком на множестве M называется бинарное отношение $\preccurlyeq \subseteq M \times M$, удовлетворяющее следующим свойствам:

- *рефлексивность*:
  для всех $a \in M$ справедливо $a \preccurlyeq a$;
- *антисимметричность*:
  для всех $a, b \in M$ из того, что $a \preccurlyeq b$ и $b \preccurlyeq a$, следует, что $a = b$;
- *транзитивность*:
  для всех $a, b,$ с $\in M$ из $a \preccurlyeq b$ и $b \preccurlyeq c$ вытекает $a \preccurlyeq c$.

Если $a \preccurlyeq b$, говорят, что a *предшествует* b.

Частично упорядоченным множеством называется пара $\langle M, \preccurlyeq\rangle$, где M — некоторое множество, а $\preccurlyeq$ — отношение частичного порядка на M.

Отношение, удовлетворяющее свойствам рефлексивности, антисимметричности и транзитивности, также называют *нестрогим частичным порядком*. Зафиксируем множество M. Бинарное отношение $\prec \subseteq M \times M$ называется *строгим частичным порядком*, если оно удовлетворяет следующим свойствам:

- *антирефлексивность*:
  для всех $a \in M$ справедливо $\neg(a \prec a)$;
- *асимметричность*:
  для всех $a, b \in M$ из того, что $a \prec b$, следует, что $\neg(b \prec a)$;
- *транзитивность*:
  для всех $a, b, c \in M$ из $a \prec b$ и $b \prec c$ вытекает $a \prec c$.

Очевидно, что если $\preccurlyeq$ — нестрогий частичный порядок, то $\preccurlyeq \setminus \{(a, a) \mid a \in M\}$ — строгий. Обратно: если $\prec$ — строгий частичный порядок, то $\prec \cup \{(a, a) \mid a \in M\}$ — нестрогий. Таким образом, нет большой разницы, какой порядок рассматривать, нестрогий или строгий. Мы в этом разделе будем работать со строгими порядками.

Частично упорядоченное множество $\langle M, \prec \rangle$ называется *фундированным*, если в нем отсутствуют бесконечно убывающие последовательности элементов: $a_0 \succ a_1 \succ \cdots$ [36] (это условие известно как *условие обрыва убывающих цепей*). Другими словами, для любого непустого подмножества $W \subseteq M$ частично упорядоченное множество $\langle W, \prec \cap W^2 \rangle$ имеет *минимальный элемент*, т.е. элемент $a \in W$, такой что для любого $b \in W$ не верно, что $b \prec a$: либо $a \prec b$, либо $a = b$, либо a и b несравнимы[37] [Ерш87].

Пример 6.1

Рассмотрим примеры фундированных множеств.

- $\langle \mathbb{N}_0, < \rangle$ — расширенное множество натуральных чисел $\mathbb{N}_0 = \{0, 1, \dots\}$ с отношением «строго меньше».
- $\langle \mathbb{Z}, \prec \rangle$ — множество целых чисел $\mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$ со следующим отношением: $a \prec b$ тогда и только тогда, когда $a \neq b$ и $b \vdots a$ (b делится на a без остатка).

---

[36] Запись $a \succ b$ эквивалентна записи $b \prec a$.

[37] Строго говоря, эти определения не эквивалентны: первое определение эквивалентно второму, если принять *аксиому выбора*.

- $\langle \Sigma^*, \subset \rangle$ — множество конечных слов в алфавите Σ со следующим отношением: $a \subset b$ тогда и только тогда, когда слово a является строгим подсловом слова b.

□

6.2.1. Оценочные функции и условия завершимости

Пусть мы верифицируем некоторую блок-схему P. Рассмотрим точки сечения, т.е. дуги, выбранные таким образом, чтобы каждый цикл блок-схемы содержал по крайней мере одну из дуг этого множества. Для каждой точки сечения i сформулируем индуктивное утверждение χ_i (см. лекцию 5). Для каждого базового пути π (пусть он начинается в точке сечения i и завершается в точке сечения j) запишем условие верификации:

$$VC_{ij} \equiv (\chi_i \wedge \varphi_\pi) \rightarrow \chi_j \theta_\pi,$$

где φ_π — формула пути, а θ_π — подстановка, определяющая, как изменится состояние блок-схемы при последовательном исполнении лежащих на пути операторов.

Выберем фундированное множество $\langle M, \prec \rangle$ и для каждой точки сечения i определим терм u_i, задающий *оценочную функцию* — функцию, ставящую в соответствие каждому состоянию, удовлетворяющему условию χ_i, элемент множества M. Сформулируем для каждого внутреннего базового пути π *условие завершимости* (*TC*, *Termination Condition*):

$$TC_{ij} \equiv (\chi_i \wedge \varphi_\pi) \rightarrow (u_j \theta_\pi \prec u_i).$$

Это условие выражает простую мысль: если утверждение χ_i истинно в точке сечения i и, начиная с этой точки, вычисление пойдет по пути π, то значение оценочной функции u_j после исполнения операторов, лежащих на пути, будет «строго меньше» значения оценочной функции u_i в исходном состоянии. Условие обрыва убывающих цепей гарантирует конечность вычисления.

6.2.2. Общая схема метода Флойда

Пусть дана блок-схема и ее спецификация. Метод Флойда, включающий в полном варианте метод индуктивных утверждений и метод фундированных множеств, состоит из следующих шагов:

1. выбор точек сечения;
2. формулировка индуктивных утверждений для точек сечения;
3. формулировка оценочных функций для внутренних точек сечения;

4. построение условий верификации для базовых путей;
5. построение условий завершимости для внутренних базовых путей;
6. доказательство условий верификации и условий завершимости.

Если удается доказать истинность всех условий верификации и условий завершимости, то блок-схема полностью корректна относительно заданной спецификации. Для доказательства завершимости вместо содержательного постусловия достаточно рассмотреть $true$.

Как и выбор индуктивных утверждений, построение оценочных функций с трудом поддается автоматизации.

Пример 6.2

Применим метод фундированных множеств для доказательства завершимости блок-схемы целочисленного деления (см. лекцию 5) во всех состояниях, удовлетворяющих предусловию $\varphi \equiv (a \geq 0) \wedge (b > 0)$. Обратите внимание: ограничение на b теперь строгое. Рассмотрим точку сечения B (см. рис. 5.3). Этой точке сопоставлено индуктивное утверждение

$$\chi_B \equiv (a = q' \cdot b + r') \wedge (0 \leq r' \leq a) \wedge \varphi.$$

Задачи, которые нужно решить:

1. предложить для точки B оценочную функцию;
2. записать условие завершимости для единственного внутреннего базового пути BB;
3. доказать условие завершимости.

В качестве фундированного множества возьмем $\langle \mathbb{N}_0, < \rangle$. Рассмотрим оценочную функцию $u_B \equiv (a - q')$. Покажем, что u_B определена во всех состояниях, удовлетворяющих χ_B, и ее значение принадлежит выбранному множеству.

Покажем, что $u_B \in \mathbb{N}_0$ есть следствие χ_B:

- $(a = q' \cdot b + r') \wedge (0 \leq r' \leq a) \wedge (a \geq 0) \wedge (b > 0) \equiv$
- $\left(q' = \frac{a - r'}{b}\right) \wedge (0 \leq r' \leq a) \wedge (a \geq 0) \wedge (b > 0) \rightarrow$
- $\left(q' \leq \frac{a}{b}\right) \wedge (a \geq 0) \wedge (b > 0) \rightarrow$
- $(q' \leq a) \equiv$

- $(a - q') \geq 0 \equiv$
- $u_B \in \mathbb{N}_0$.

Доказательства условий верификации для усиленных вариантов предусловия φ и индуктивного утверждения χ_B практически не отличаются от приведенных в лекции 5. Докажем условие завершимости:

$$(\chi_B \wedge \varphi_{BB}) \rightarrow (u_B\theta_{BB} < u_B),$$

где $\varphi_{BB} \equiv (r' \geq b)$ и $\theta_{BB} \equiv [q' \coloneqq q' + 1, r' \coloneqq r' - b]$:

- $(\chi_B \wedge \varphi_{BB}) \rightarrow \big((a - q')[q' \coloneqq q' + 1, r' \coloneqq r' - b] < (a - q')\big) \equiv$
- $(\chi_B \wedge \varphi_{BB}) \rightarrow \Big(\big((a - (q' + 1)\big) < (a - q')\Big) \equiv$
- $(\chi_B \wedge \varphi_{BB}) \rightarrow (0 < 1) \equiv$
- $(\chi_B \wedge \varphi_{BB}) \rightarrow true \equiv$
- $true$.

Таким образом, для предусловия $(a \geq 0) \wedge (b > 0)$ блок-схема завершается.

□

6.3. Метод счетчиков

Еще один способ доказательства полной корректности программ дает метод счетчиков [Неп88]. В верифицируемую программу вводятся *переменные-счетчики* — по одной для каждого цикла. Счетчик инициализируется перед входом в цикл и увеличивает свое значение при каждом исполнении тела цикла. В инварианты циклов включаются условия, ограничивающие значения счетчиков функциями, зависящими только от входных переменных (такие функции называются *ограничивающими*). Для доказательства полной корректности исходной программы достаточно доказать частичную корректность «инструментированной» программы, т.е. программы с добавленными счетчиками.

Пример 6.3

В программу целочисленного деления дополнительный счетчик можно не вводить — его роль играет переменная q, аккумулирующая неполное частное от деления a на b.

```
⟨a ≥ 0 ∧ b > 0⟩
q := 0;          /* инициализация счетчика */
```

```
r := a;
while r ≥ b do
  q := q + 1;  /* инкремент счетчика */
  r := r - b
end
⟨a = q·b + r ∧ 0 ≤ r < b⟩
```

Требуется сформулировать инвариант цикла таким образом, чтобы в него вошло условие ограниченности q. Рассмотрим ограничение $q \leq a$ и следующее условие, претендующее на роль инварианта цикла:

$$\chi \equiv (a = q \cdot b + r) \wedge (0 \leq r \leq a) \wedge (a \geq 0) \wedge (b > 0) \wedge (q \leq a).$$

Ранее мы доказали, что $(a = q \cdot b + r) \wedge (0 \leq r \leq a)$ — инвариант (см. лекцию 3). Эта же формула, усиленная предусловием,

$$(a = q \cdot b + r) \wedge (0 \leq r \leq a) \wedge (a \geq 0) \wedge (b > 0),$$

также является инвариантом. В предыдущем примере мы показали, что $q \leq a$ является следствием $(a = q \cdot b + r) \wedge (0 \leq r \leq a) \wedge (a \geq 0) \wedge (b > 0)$. Таким образом, χ — инвариант, откуда (с учетом того, что χ содержит ограничение на значение счетчика q) следует завершимость и полная корректность программы.

□

Пример 6.4

Докажем завершимость программы $SQRT$, вычисляющей целую часть квадратного корня неотрицательного целого числа: $r = \lfloor \sqrt{a} \rfloor$, где $a \geq 0$. Для этого введем вспомогательную переменную-счетчик i (см. таблицу 6.2).

Таблица 6.2. Добавление счетчика в цикл программы $SQRT$

Исходная программа ($SQRT$)	Программа после добавления счетчика

d := 1;
r := 0;
x := a;
 	
while x > 0 **do**
 	
  x := x - d;
  d := d + 2;
  if x ≥ 0 **then**
  r := r + 1
  end
end
	d := 1;
	r := 0;
	x := a;
	i := 0; /* инициализация */
	while x > 0 **do**
	  i := i + 1; /* инкремент */
	  x := x - d;
	  d := d + 2;
	  if x >= 0 **then**
	  r := r + 1
	  end
	end

Обратите внимание: в теле цикла значение переменной x, изначально равное a, уменьшается на очередное нечетное число $d = 1, 3, \ldots$ Таким образом, условие

$$x = a - \sum_{k=1}^{i} (2k - 1)$$

является инвариантом.

Если $x > 0$, то перед исполнением цикла справедливо неравенство $\sum_{k=1}^{i}(2k-1) = i^2 < a$, что равносильно $i < \sqrt{a}$. Если допустить, что условие цикла ложно ($x = a = 0$ или значение x стало неположительным в результате исполнения предыдущей итерации), имеет место оценка $i < \sqrt{a} + 1$. Существование ограничивающей функции $u_i \equiv \sqrt{a} + 1$ для счетчика i доказывает завершимость программы.

□

6.4. Дедуктивная система для доказательства полной корректности

Дедуктивная система, рассмотренная в лекциях 2 и 3, предназначена для доказательства частичной корректности while-программ. Как ее модифицировать, чтобы она позволяла доказывать полную корректность? Метод фундированных множеств дает ответ на этот вопрос (см. таблицу 6.3) [Apt09].

Таблица 6.3. Аксиоматическая семантика языка программирования while (полная корректность)

(1)	$\langle\varphi\rangle\mathbf{skip}\langle\varphi\rangle$	*Без изменений*
(2)	$\langle\varphi[x \coloneqq t]\rangle x \coloneqq t\langle\varphi\rangle$	*Без изменений*
(3)	$\dfrac{\langle\varphi\rangle P_1\langle\chi\rangle, \langle\chi\rangle P_2\langle\psi\rangle}{\langle\varphi\rangle P_1; P_2\langle\psi\rangle}$	*Без изменений*
(4)	$\dfrac{\langle\varphi \wedge B\rangle P_1\langle\psi\rangle, \langle\varphi \wedge \neg B\rangle P_2\langle\psi\rangle}{\langle\varphi\rangle\mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2\langle\psi\rangle}$	*Без изменений*
(5)	$\dfrac{\langle\varphi \wedge B\rangle P\langle\varphi\rangle, \langle(\varphi \wedge B) \wedge (u = z)\rangle P\langle u \prec z\rangle, \varphi \rightarrow (u \in M)}{\langle\varphi\rangle\mathbf{while}\ B\ \mathbf{do}\ P\ \mathbf{end}\langle\varphi \wedge \neg B\rangle}$	Здесь u — терм оценочной функции; $\langle M, \prec\rangle$ — фундированное множество; z — уникальная переменная
(6)	$\dfrac{\varphi \rightarrow \varphi', \langle\varphi'\rangle P\langle\psi'\rangle, \psi' \rightarrow \psi}{\langle\varphi\rangle P\langle\psi\rangle}$	*Без изменений*

Изменяется только правило вывода для оператора цикла: в нем появляется условие «строгого уменьшения» значения оценочной функции при исполнении тела цикла.

Пример 6.5

Еще раз обратимся к программе целочисленного деления. На этот раз рассмотрим оценочную функцию $u \equiv r$. Докажем условие

$$\langle\chi \wedge (r \geq b) \wedge (r = z)\rangle\, q \coloneqq q + 1; r \coloneqq r - b\, \langle r < z\rangle,$$

где $\chi \equiv (a = q \cdot b + r) \wedge (r \geq 0) \wedge (b > 0)$. Используем известный подход — вычисление слабейшего предусловия (см. лекцию 3):

$$wp(q \coloneqq q + 1, r \coloneqq r - b, r < z) = (r - b) < z.$$

Покажем, что имеет место импликация $\big(\chi \wedge (r \geq b) \wedge (r = z)\big) \rightarrow \big((r - b) < z\big)$, т.е. что условие $\chi \wedge (r \geq b) \wedge (r = z)$ сильнее слабейшего предусловия, гарантирующего истинность постусловия $r < z$:

- $\big((a = q \cdot b + r) \wedge (r \geq 0) \wedge (b > 0) \wedge (r \geq b) \wedge (r = z)\big) \rightarrow \big((r - b) < z\big) \leftarrow$
- $\big((b > 0) \wedge (r = z)\big) \rightarrow \big((r - b) < z\big) \leftarrow$
- $\big((b > 0) \wedge (r = z)\big) \rightarrow (b > 0) \leftarrow$
- $(b > 0) \rightarrow (b > 0) \leftarrow$
- $true$.

Отсюда можно сделать вывод, что программа завершается.

□

6.5. Аннотирование программ для доказательства завершимости

В лекции 3 было введено понятие *аннотированной программы* — программы, в которую добавлены специальные утверждения (аннотации), описывающие свойства ее фрагментов. До сих пор мы имели дело с двумя типами аннотаций: ограничениями на состояние программы и инвариантами циклов. Для доказательства завершимости и полной корректности программ применяются аннотации, задающие оценочные (ограничивающие) функции. Такие аннотации записываются следующим образом:

$$\{\mathbf{bd}: u[, \langle M, \prec \rangle]\}.$$

Квадратные скобки говорят о том, что заключенный в них текст (задание фундированного множества) является необязательным: вариант по умолчанию — $\langle \mathbb{N}_0, < \rangle$.

Пример 6.6

Ниже приведена программа целочисленного деления, включающая аннотации для инварианта и оценивающей функции.

```
⟨a ≥ 0 ∧ b > 0⟩ /* предусловие */
q := 0;
r := a;
{inv: a = q·b + r ∧ r ≥ 0 ∧ b > 0} /* инвариант */
{bd: r}                            /* оценочная функция */
while r ≥ b do
  q := q + 1;
  r := r - b
end
⟨a = q·b + r ∧ 0 ≤ r < b⟩ /* постусловие */
```

В качестве упражнения выпишите условия верификации и условия завершимости.

□

Рассмотрим, как пишутся аннотации на языке ACSL для C-программ (см. лекцию 4). Оценочная функция, т.е. величина, уменьшаемая при каждом исполнении тела цикла, задается с помощью клаузы **loop variant** [Bau10]:

```
/*@ loop variant u [for R]; */
```

Здесь u — терм, задающий целочисленную оценку, а R — необязательный параметр, задающий фундированное множество (точнее, отношение строгого частичного порядка). По умолчанию используется фундированное множество $\langle \mathbb{N}_0, < \rangle$.

Пример 6.7

Приведенная ниже ACSL-аннотация оценочной функции равнозначна аннотации $\{\mathrm{bd}: r\}$ из предыдущего примера.

```
int idiv(int a, int b, int *r) {
  int q = 0;
  int p = a;
  /*@ ...
    @ loop variant p;
    @*/
  while(p <= b) {
    q++;
    p -= b;
  }
  *r = p;
  return q;
}
```

□

Чтобы определить специфический частичный порядок можно использовать конструкцию **predicate**, позволяющую определять предикаты:

```
/*@ predicate R(T x, T y) = ...; */
```

Пример 6.8

Определим фундированное множество $\langle \mathbb{Z}, \prec \rangle$ со следующим отношением частичного порядка: $a \prec b$ тогда и только тогда, когда $a \neq b$ и $b \vdots a$.

```
/*@ predicate RemIsZero(integer a, integer b) =
  @   a != 0 && a != b && b % a == 0;
  @*/
```

□

6.6. Вопросы и упражнения

1. Каким свойствам удовлетворяет отношение строгого частичного порядка?
2. Дайте определение фундированного множества. Приведите примеры.
3. Является ли фундированными следующие частично упорядоченные множества:

a. $\langle \mathbb{Q}_+, < \rangle$, где $\mathbb{Q}_+$ — множество положительных рациональных чисел;

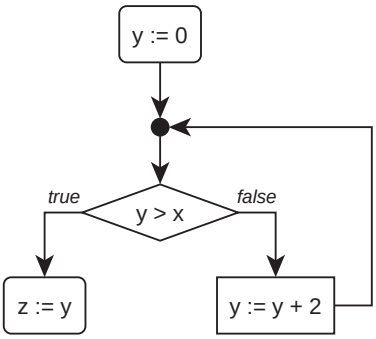
b. $\langle \mathbb{N}_0 \times \mathbb{N}_0, \prec \rangle$, где отношение $\prec$ определяется следующим образом:
$(a, b) \prec (c, d)$ тогда и только тогда, когда $a < c$ и $b < d$;

c. $\langle \mathbb{N}_0 \times \mathbb{N}_0, \prec \rangle$, где отношение $\prec$ определяется следующим образом:
$(a, b) \prec (c, d)$ тогда и только тогда, когда $a < c$ и $b > d$;

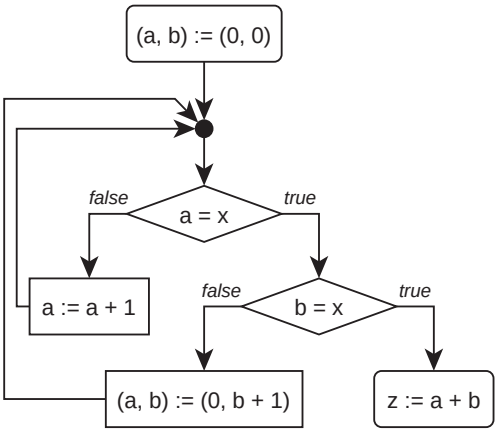
d. $\langle \Sigma^*, < \rangle$, где Σ — конечный алфавит, а $<$ — отношение лексикографического порядка?

4. Верно ли, что метод фундированных множеств является более общим по сравнению с методом счетчиков, т.е. всегда, когда удается построить ограничивающую функцию для счетчика цикла, для этого цикла можно построить оценочную функцию?

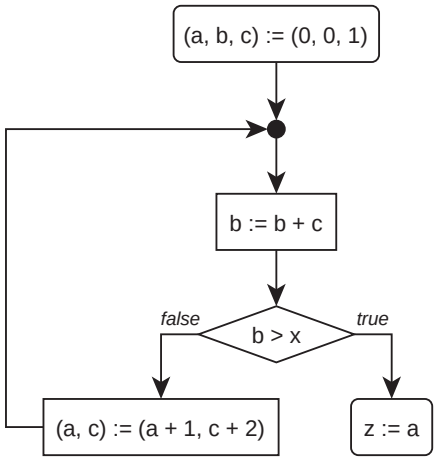
5. Докажите завершимость приведенной ниже блок-схемы для любых значений входной переменной x [Буз14].



6. Докажите завершимость приведенной ниже блок-схемы для любых неотрицательных значений входной переменной x [Буз14].



7. Докажите полную корректность приведенной ниже блок-схемы относительно предусловия $\varphi \equiv (x \geq 0)$ и постусловия $\psi \equiv (z^2 \leq x < (z+1)^2)$ [Буз14].



8. Предложите фундированные множества и оценочные функции циклов для двух реализаций алгоритма Евклида из лекции 1 (программы P и программы Q). Докажите полную корректность этих программ.

9. Предложите ограничивающие функции циклов для программы сортировки массива «методом пузырька». Докажите завершимость программы, используя метод счетчиков.

10. Аннотируйте на языке ACSL функцию дихотомического поиска в упорядоченном числовом массиве (см. лекцию 4). Докажите полную корректность этой функции.

11. Докажите значимость метода фундированных множеств Флойда.

Лекция 7. Автоматический синтез инвариантов циклов

Только постоянное изменяется; изменчивое подвергается не изменению, а только смене.

И. Кант.
Критика чистого разума

Рассматриваются вопросы автоматизации дедуктивной верификации программ, в частности вопросы автоматического синтеза инвариантов циклов. Задача синтеза оценочных функций отдельно не рассматривается — она может быть сведена к построению инвариантов у программ, расширенных счетчиками. Описываются два класса методов: эвристические методы и методы, основанные на абстрактной интерпретации, т.е. символическом исполнении в упрощенных предметных областях — абстрактных моделях. Определяются основные понятия абстрактной интерпретации; приводятся примеры интерпретации в таких областях, как интервальная арифметика и теория выпуклых многогранников.

7.1. Автоматизация дедуктивной верификации программ

В лекциях 5 и 6 мы разобрались с методом Флойда дедуктивной верификации программ. Представьте: наша цель — разработать инструмент, который берет на вход программу (код на языке программирования) и формальную спецификацию (программный контракт, записанный, например, в форме аннотаций) и выдает вердикт: положительный, если программа корректна; отрицательный, если доказать корректность не удалось[38]. С учетом изложенного ранее работа такого инструмента могла бы состоять из следующих шагов:

1. анализ исходного кода (лексический, синтаксический, семантический [Сер12]) и конструирование графа потока управления программы;
2. построение множества точек сечения (множества дуг, обладающего тем свойством, что в каждом цикле графа потока управления имеется по крайней мере одна дуга из этого множества) и перечисление базовых путей (путей, соединяющих точки сечения и не содержащих точки сечения внутри себя) (см. лекцию 5);

[38] Заметим, что такая постановка задачи допускает тривиальное решение, когда инструмент всегда возвращает отрицательный вердикт. Если же потребовать, чтобы для любой корректной программы инструмент возвращал положительный вердикт, а для любой некорректной —отрицательный, то решения, по понятным причинам, не существует.

3. конструирование формул и подстановок для базовых путей: формула пути определяет условие, гарантирующее проход по этому пути; подстановка задает преобразование состояния программы, совершаемое при исполнении операторов, расположенных вдоль пути (см. лекцию 5);
4. синтез индуктивных утверждений и оценочных функций для внутренних точек сечения (этот вопрос рассматривается в этой лекции);
5. построение условий верификации для всех базовых путей и условий завершимости для внутренних базовых путей (см. лекции 5 и 6);
6. доказательство построенных условий верификации и завершимости (см. лекцию 8).

Возможность автоматизации шага 1 не вызывает сомнений — теория трансляции и компиляции программ хорошо проработана [Сер12]. Шаг 2 в простейшем случае сводится к обходу графа с запоминанием его вершин: если очередная дуга ведет в уже пройденную вершину, она добавляется в множество внутренних точек сечения. Конечно, постановку задачи можно усложнить, потребовав минимальности этого множества, однако вопросы оптимизации сейчас не принципиальны. Для реализации шага 3 может использоваться метод обратных подстановок, подробно описанный в лекции 5. Шаг 5 тривиален. Основные трудности автоматизации дедуктивной верификации связаны с шагами 4 и 6, т.е. с синтезом индуктивных утверждений и оценочных функций, а также с доказательством условий верификации и завершимости.

Хотя шаг 4, вообще говоря, не поддается автоматизации[39], для ряда предметных областей существуют подходы, позволяющие строить индуктивные утверждения (их рассмотрению посвящена оставшаяся часть лекции). Мы не будем отдельно рассматривать синтез оценивающих функций — эта задача может быть сведена к построению инвариантов циклов у программ, расширенных счетчиками [Неп88].

Напомним: *инвариантом цикла* $\mathbf{while}\ B\ \mathbf{do}\ P\ \mathbf{end}$ называется условие χ, такое что $\{\chi \wedge B\}P\{\chi\}$, т.е. условие, которое истинно перед каждым шагом цикла и после него. Предположим, что у цикла известны предусловие φ и постусловие ψ:

[39] Наличие алгоритма синтеза оценочных функций означало бы алгоритмическую разрешимость проблемы останова.

$$\{\varphi\}\ \textbf{while}\ B\ \textbf{do}\ P\ \textbf{end}\ \{\psi\}.$$

На основе какой информации можно построить инвариант χ, обеспечивающий истинность импликации $(\chi \wedge \neg B) \rightarrow \psi$? На лицо четыре возможности (а также их комбинации):

1. предусловие цикла φ;
2. условие цикла B;
3. тело цикла P;
4. постусловие цикла ψ.

В лекции описываются два класса методов:

- *эвристические методы* [Gri81];
- методы на основе *абстрактной интерпретации* [Cou78].

7.2. Эвристические методы синтеза инвариантов

Эвристические методы основаны на использовании формальных спецификаций. Из двух составляющих контракта, предусловия φ и постусловия ψ, наиболее важной для построения инварианта считается постусловие: оно описывает цель совершаемых программой вычислений, в то время как предусловие задает область определения. В соответствии с этим, основное внимание уделяется постусловию.

7.2.1. Ослабление постусловия

Итак, нам известно постусловие ψ:

$$\textbf{while}\ B\ \textbf{do}\ P\ \textbf{end}\ \{\psi\}.$$

В качестве кандидата на роль инварианта рассмотрим следующее утверждение [Неп88]:

$$\chi_w \ \equiv\ \neg B \rightarrow \psi \ \equiv\ B \vee \psi.$$

Формально утверждение χ_w является инвариантом. Действительно, если после исполнения тела цикла истинно условие B, то истинно и χ_w; если же условие B ложно (выход из цикла), то χ_w все равно истинно, поскольку истинно постусловие ψ. Как правило, такой инвариант оказывается слишком слабым.

Другим кандидатом на роль инварианта является само постусловие: $\chi_s \equiv \psi$ [Gri81]. Вообще говоря, χ_s не является инвариантом: ψ истинно после выхода из цикла, но для остальных

итераций это не так. Если утверждение χ_w слишком слабое (нуждается в усилении), то χ_s, напротив, слишком сильное (нуждается в ослаблении):

$$\chi_s \to \chi \to \chi_w.$$

Знание того, что χ «лежит» между χ_w и χ_s не помогает в синтезе инвариантов — «отрезок» $[\chi_w, \chi_s]$ содержит бесконечное множество формул (если, конечно, B не есть $false$).

Будем строить инвариант путем ослабления постусловия. Дэвид Грис (David Gries, род. в 1939 г.) использует метафору воздушного шарика [Gri81]. Постусловие — это воздушный шарик, из которого выпущен воздух, а инвариант — надутый шарик[40]; при каждом исполнении тела цикла из шарика выпускается немного воздуха (цель приближается — множество возможных состояний уточняется). «Вопрос ⟨… ⟩ состоит в том, чтобы знать, как надуть шарик так, чтобы выполнение цикла могло его опустошить полностью».

Имеются следующие основные эвристики для ослабления утверждений (для усиления можно использовать двойственные эвристики) [Gri81]:

1. *устранение конъюнктивного члена*:
   замена формулы $(\varphi_1 \wedge \ldots \wedge \varphi_n)$ на $(\varphi_1 \wedge \ldots \wedge \varphi_{i-1} \wedge \varphi_{i+1} \wedge \ldots \wedge \varphi_n)$;
2. *расширение области значений переменной*:
   замена условия вида $a \le x \le b$ на $a \le x$ или $x \le b$ и т.п.;
3. *замена константы* (*или входной переменной*) *переменной* (*или термом*):
   название этой эвристики говорит само за себя.

Пример 7.1

Попытаемся, используя перечисленные выше эвристики, построить инвариант цикла программы целочисленного деления.

```
{a ≥ 0 ∧ b > 0}
q := 0;
r := a;
while r ≥ b do
  q := q + 1;
  r := r - b
```

[40] Будучи слабее, инвариент характеризует более «объемное» множество состояний по сравнению с постусловием.

```
end
{a = q·b + r ∧ 0 ≤ r < b}
```

В качестве аппроксимации инварианта рассмотрим постусловие

$$(a = q \cdot b + r) \wedge (0 \leq r < b).$$

Очевидно, что инвариантом оно не является. Применяя эвристику 1, устраним конъюнктивный член $(0 \leq r < b)$. Получим формулу $(a = q \cdot b + r)$, которая, как легко убедиться, есть инвариант, но слишком слабый. Вернемся к постусловию и применим к нему эвристику 2, т.е. расширим область значений переменной r с полуинтервала $[0, b)$ до полупрямой $[0, \infty)$. Получим адекватный инвариант $(a = q \cdot b + r) \wedge (r \geq 0)$.

□

Пример 7.2

Для иллюстрации эвристики 3 (замены константы переменной) рассмотрим программу суммирования элементов числового массива.

```
{N ≥ 0}
s := 0;
n := 0;
while n < N do
  s := s + x[n];
  n := n + 1
end
{s = sum(x, N)}
```

В качестве аппроксимации инварианта возьмем постусловие, т.е. $s = sum(x, N)$ (определение функции sum дано в лекции 3). Замена входной переменной N на внутреннюю переменную n дает подходящий инвариант $s = sum(x, n)$.

□

7.2.2. Комбинирование пред- и постусловий

В некоторых случаях при построении инварианта цикла следует рассматривать как постусловие, так и предусловие (понятно, что инвариант должен быть слабее и того и другого) [Gri81]. Основная идея состоит в построении формулы, которая в некотором смысле объединяла бы в себе оба условия. Ограничимся примером.

Пример 7.3

Рассмотрим немного измененный пример из книги [Gri81]. Программа в цикле «пробегает» по элементам числового массива и прибавляет к i-ому элементу число i. Обратите внимание: в предусловии вводятся символические имена C_i, обозначающие начальные значения элементов массива.

```
{N ≥ 0 ∧ ∀i((0 ≤ i < N) → (x[i] = Cᵢ))}
n := 0;
while n ≠ N do
  x[n] := x[n] + n;
  n := n + 1
end
{∀i((0 ≤ i < N) → (x[i] = Cᵢ + i))}
```

Как и раньше, возьмем за основу инварианта постусловие. Применив эвристику 3 и заменив входную переменную N на внутреннюю переменную n, получим условие

$$\forall i\big((0 \le i < n) \to (x[i] = C_i + i)\big),$$

являющееся инвариантом. Условие это, хотя и позволяет доказать частичную корректность программы, не является *полным* в том смысле, что в нем нет ограничений на значения элементов $x[n], \dots, x[N-1]$[41]. Усилив его ограничением, взятым из предусловия, получим следующую формулу:

$$\forall i\Big(\big((0 \le i < n) \wedge (x[i] = C_i + i)\big) \vee \big((n \le i < N) \wedge (x[i] = C_i)\big)\Big).$$

□

7.3. Методы синтеза инвариантов на основе абстрактной интерпретации

Если эвристические методы синтеза инвариантов циклов базировались на *спецификации*, то в методах следующего типа задействуется *реализация*. Методы, о которых идет речь, основаны на *абстрактной интерпретации*. Ключевые концепции этого подхода были сформулированы Патриком Кюзо (Patrick Cousot, род. в 1948 г.) в 1977 г. [Cou77].

[41] В некоторых случаях, например, когда речь идет о синтезе программы по спецификации, полнота имеет большое значение [Gri81].

Под *абстрактной интерпретацией* понимается *символическое* исполнение программы на *упрощенной* (*абстрактной*) *модели* (*abstract domain*). Абстрактная интерпретация осуществляется пошагово, но, в отличие от обычного исполнения, результатом каждого шага является не конкретное состояние, а символическое представление множества возможных состояний. Важно отметить: такое представление является *приближенным* (что отличает абстрактную интерпретацию от традиционного символического исполнения), но обязательно *аппроксимацией сверху* — представление охватывает все возможные состояния и, возможно, некоторые другие.

Упрощенные модели, т.е. способы приближенного представления множеств состояний, могут быть разными и определяются предметной областью.

7.3.1. Абстрактная интерпретация и примеры упрощенных моделей

Одним из самых известных примеров упрощенных моделей, используемых для абстрактной интерпретации, является *интервальная арифметика* [Шок81]. В этой модели множество возможных значений числовой переменной программы аппроксимируется интервалом[42], а множество состояний — множеством интервалов для всех переменных. Над интервалами определены обычные арифметические операции:

- *сложение*: $[a, b] + [c, d] = [a + c, b + d]$;
- *вычитание*: $[a, b] - [c, d] = [a - d, b - c]$;
- *умножение*: $[a, b] \cdot [c, d] = [\min(a \cdot c, a \cdot d, b \cdot c, b \cdot d), \max(a \cdot c, a \cdot d, b \cdot c, b \cdot d)]$;
- *деление*: $\frac{[a,b]}{[c,d]} = \left[\min\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right), \max\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right)\right]$,
  если интервал-делитель не содержит нуля.

Числовые константы трактуются как одноэлементные интервалы.

[42] Следуя англоязычной терминологии, под *интервалом* здесь понимается произвольный, а не только открытый, промежуток числовой прямой. Более того, в контексте интервальной арифметики речь идет об *отрезках* (замкнутых интервалах).

Пример 7.4

Рассмотрим простую программу и проинтерпретируем ее, используя целочисленную интервальную арифметику. Программа и шаги ее интерпретации показаны в таблице 7.1.

Таблица 7.1. Абстрактная интерпретация в целочисленной интервальной арифметике

Программа	Аппроксимация множества возможных состояний
`{0 ≤ x ≤ 2 ∧ 0 ≤ y ≤ 3}`	$\{x \mapsto [0,2], y \mapsto [0,3]\}$
if `x > y` **then**	**Уточнение интервалов**: $x > \min(y) = 0 \Rightarrow x \geq 1 \Rightarrow x \in [1,2]$
`/* ветвь then */`	$y < \max(x) = 2 \Rightarrow y \leq 1 \Rightarrow y \in [0,1]$
`x := x + y;`	$\{x \mapsto [1,3], y \mapsto [0,1]\}$
`y := x - y;`	$\{x \mapsto [1,3], y \mapsto [0,3]\}$
`x := x - y`	$\{x \mapsto [-2,3], y \mapsto [0,3]\}$
else	**Уточнение интервалов**: $x \leq \max(y) = 3 \Rightarrow x \in [0,2]$ (уточнения нет)
`/* ветвь else */`	$y \geq \min(x) = 0 \Rightarrow y \in [0,3]$ (уточнения нет)
`x := 2 * x;`	$\{x \mapsto [0,4], y \mapsto [0,3]\}$
`y := y + 1`	$\{x \mapsto [0,4], y \mapsto [1,4]\}$
end	**Объединение интервалов** (соединение ветвей **then** и **else**):
`{0 ≤ x ≤ 4 ∧ 0 ≤ y ≤ 4}`	$\{x \mapsto [-2,4], y \mapsto [0,4]\}$

Заметим, что результат абстрактной интерпретации, отображение $\{x \mapsto [-2,4], y \mapsto [0,4]\}$, получился «шире» точной оценки (в классе интервальных ограничений), приведенной в постусловии: $\{x \mapsto [0,4], y \mapsto [0,4]\}$. Это связано с упрощенным характером используемой модели — в ней не учитываются взаимосвязи между переменными.

□

В лекции 2 мы говорили, что для языка программирования задана *формальная семантика*, если определена функция M, ставящее в соответствие каждой программе P и каждому начальному состоянию s множество возможных конечных состояний $M[\![P]\!](s)$. Для удобства можно рассматривать не одиночные состояния, а их множества, т.е. считать, что функция $M[\![P]\!]$ — *функция переходов программы* P — отображает множество начальных состояний во множество возможных конечных состояний:

$$M[\![P]\!]: 2^S \to 2^S.$$

Заметим, что пара $\langle 2^S, \subseteq \rangle$ является частично упорядоченным множеством (см. лекцию 6), а функция $M[\![P]\!]$ *монотонна*, т.е. $M[\![P]\!](S') \subseteq M[\![P]\!](S)$ для всех $S' \subseteq S$.

Для заданной упрощенной модели AD (от англ. *Abstract Domain*) и заданной программы P функция переходов «трансформируется» в *обобщенную функцию переходов*[43]:

$$M_{AD}[\![P]\!]: 2^S \to 2^S.$$

Теперь $M_{AD}[\![P]\!](S)$ — это не множество всех возможных состояний, в которые можно попасть из состояний множества S при исполнении программы P, а аппроксимация этого множества — надмножество $M[\![P]\!](S)$, представимое в заданной модели.

В дальнейшем для краткости мы будем опускать индекс AD у обобщенных функций переходов. Дадим несколько вспомогательных определений.

Пусть $\langle M, \preccurlyeq \rangle$ — частично упорядоченное множество. *Верхней гранью* множества $W \subseteq M$ называется элемент $a \in M$, такой что $b \preccurlyeq a$ для всех $b \in W$. *Точная верхняя грань* множества W (обозначается: $\sup W$) — это такая его верхняя грань a, что $a \preccurlyeq a'$ для любой другой верхней грани a'[44] [Bir67]. Частично упорядоченное множество $\langle M, \preccurlyeq \rangle$ называется *полным*, если любое множество $W \subseteq M$ имеет в нем точную верхнюю грань.

Обратите внимание: если $W = \emptyset$, то $\sup W$ — минимальный элемент M. Таким образом, полнота предполагает наличие минимального элемента.

Имеет место следующий факт (*теорема о неподвижной точке*), установленный Альфредом Тарским (Alfred Tarski, 1901-1983) в 1942 г.: если $\langle M, \preccurlyeq \rangle$ — полное частично упорядоченное множество и $f: M \to M$ — монотонная функция, т.е. $f(x) \preccurlyeq f(y)$, если $x \preccurlyeq y$, то $f(a) = a$ для некоторого $a \in M$[45] [Bir67]. Элемент $a \in M$, такой что $f(a) = a$, называется *неподвижной точкой* функции f.

Заметим, что $\langle 2^S, \subseteq \rangle$ — полное частично упорядоченное множество. В самом деле, для любого множества W подмножеств S точная верхняя грань $\sup W$ есть не что иное, как теоре-

[43] В общем случае в абстрактной интерпретации речь идет о преобразованиях одной характеристики (элемента частично упорядоченного множества) в другую (элемент того же самого множества); вид таких характеристик может быть самым разным – важно, чтобы преобразование было монотонным.

[44] Из антисимметричности частичного порядка следует, что если точная верхняя грань существует, то она единственна.

[45] Нетрудно убедиться, что $a = \sup\{x \in M \mid x \preccurlyeq f(x)\}$ — неподвижная точка функции f.

тико-множественное объединение $\bigcup_{S' \in W} S'$ (если $W = \emptyset$, объединение полагается пустым). Таким образом, по теореме Тарского, обобщенная функций переходов $M[\![P]\!]$ имеет неподвижную точку.

То же справедливо в отношении функции $M^*[\![P]\!](S) \equiv M[\![P]\!](S) \cup S$, которая отражает состояния до и после исполнения программы. Многократное применение функции $M^*[\![P]\!]$, $M^*[\![P]\!]\left(M^*[\![P]\!]\left(\dots(S)\right)\right)$, дает множество всех состояний (начальных, промежуточных, конечных), возможных при исполнении $P; \dots; P$.

Как вся эта математика связана с поиском инварианта цикла? Предположим, что требуется найти инвариант цикла **while** B **do** P **end**. Пусть зафиксирована некоторая упрощенная модель и $M[\![P]\!]$ — обобщенная функция тела цикла P. В качестве кандидата на роль инварианта берется символическое представление неподвижной точки функции $M^*[\![P]\!]$. Заметим: S' — неподвижная точка $M^*[\![P]\!]$ тогда и только тогда, когда $M[\![P]\!](S') \subseteq S'$, что интерпретируется как сохранение свойства, или *инвариантность*.

Подходящая неподвижная точка может быть выражена как предел последовательности $S^{(i+1)} = M^*[\![P]\!](S^{(i)})$[46], где $S^{(0)}$ — это аппроксимация множества состояний, в которых выполнено предусловие цикла. Чтобы итерационный процесс *стабилизировался* за конечное число шагов, применяют так называемые *операторы расширения*.

Оператор $\nabla\colon 2^S \times 2^S \to 2^S$ называется *оператором расширения*, если для любых двух множеств X и Y (допускающих представление в терминах соответствующей упрощенной модели) имеет место вложение $(X \cup Y) \subseteq (X \nabla Y)$ и, кроме того, для любой последовательности $\{X^{(i)}\}_{i=0}^{\infty}$ и любого множества $S^{(0)}$ последовательность $S^{(i+1)} = S^{(i)} \nabla X^{(i)}$ когда-нибудь стабилизируется [Mon09].

Инвариант цикла строится с помощью итерационного процесса $S^{(i+1)} = S^{(i)} \nabla M[\![P]\!]\left(S^{(i)}\right)$, критерием завершения которого является условие $M[\![P]\!]\left(S^{(i)}\right) \subseteq S^{(i)}$. Поясним на примере.

---

[46] Строго говоря, для этого нужно потребовать *непрерывность* функции $M^*[\![P]\!]$: для всех последовательностей $S_0 \subseteq S_1 \subseteq \cdots$ должно быть выполнено равенство $M^*[\![P]\!](\bigcup_{i=0}^{\infty} S_i) = \bigcup_{i=0}^{\infty} M^*[\![P]\!](S_i)$.

Пример 7.5

Найдем инвариант приведенной ниже циклической программы, используя абстрактную интерпретацию в теории выпуклых многогранников [Cou78]. Условия B_1 и B_2 (см. ниже) могут быть любыми (в примере они не учитываются, хотя могли бы).

```
{i = 2 ∧ j = 0}
while B1 do
  if B2 then
    i := i + 4
  else
    i := i + 2;
    j := j + 1
  end
end
```

Чтобы почувствовать разницу между абстрактной и конкретной интерпретацией, выпишем явно множества возможных состояний программы при исполнении не более чем n итераций цикла. Ниже приведены такие множества для $n \leq 3$ (каждое состояние описывается парой, в которой первый элемент — значение переменной i, второй — значение переменной j):

0. $\{(2,0)\}$;
1. $\{(2,0),(6,0),(4,1)\}$;
2. $\{(2,0),(6,0),(4,1),(10,0),(8,1),(6,2)\}$;
3. $\{(2,0),(6,0),(4,1),(10,0),(8,1),(6,2),(14,0),(12,1),(10,2),(8,3)\}$.

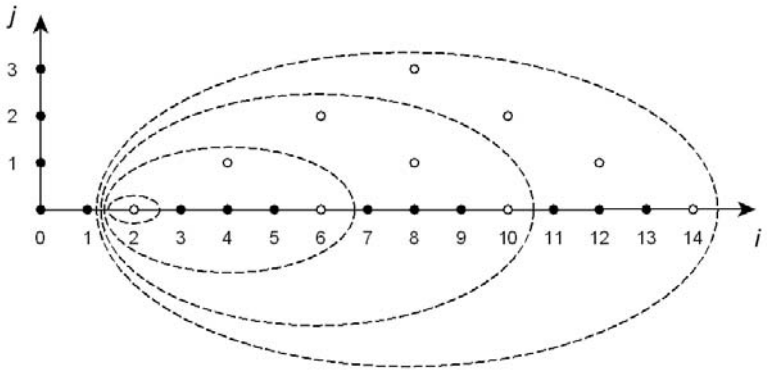


Рисунок 7.1. Множества возможных состояний при исполнении не более чем n итераций ($n = 0, 1, 2, 3$)

Рассмотрим теперь абстрактную интерпретацию этой программы. Множество возможных состояний программы аппроксимируется *выпуклым многогранником* (*CP*, *Convex Polyhedron*). Каждый раз, когда происходит соединение разных путей исполнения программы (оператор **join**) происходит вычисление *выпуклой оболочки* совокупности многогранников, вычисленных на разных ветвях программы. Таким образом, на выходе мы имеем выпуклый многогранник, содержащий многогранники, полученные на входе:

$$CP = Conv(\bigcup_{i=1}^{n} CP_i), \text{ где } Conv(X) \text{ — выпуклая оболочка множества } X.$$

Особым образом обрабатываются циклы. Занумерует дуги, ведущие в начало цикла, от 1 до n так, чтобы номер n имела дуга обратной связи, соответствующая переходу на очередную итерацию цикла. Как и прежде, вычисляется выпуклая оболочка $CP = Conv(\bigcup_{i=1}^{n} CP_i)$, однако в качестве результата берется, не многогранник CP, а

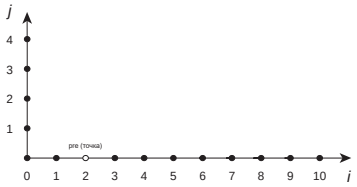
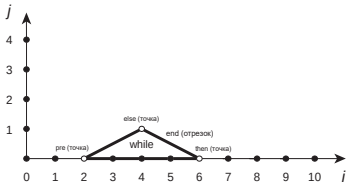
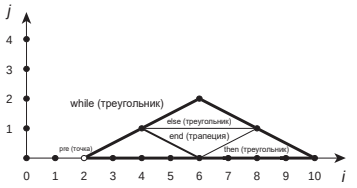
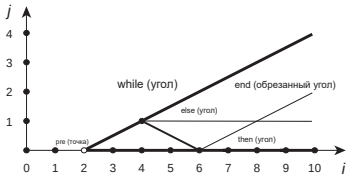
$$CP_n^{(i)} = CP_n^{(i-1)} \nabla CP,$$

где ∇ — оператор расширения, а i — номер итерации ($CP_n^{(0)} = Conv(\bigcup_{i=1}^{n-1} CP_i)$). В рассматриваемой модели $X \nabla Y$ — это многогранник, описываемый той же системой линейных неравенств, что и X, но из которой удалены ограничения, которым не удовлетворяет Y.

Рассмотрим пошагово абстрактную интерпретацию приведенной выше программы (см. таблицу 7.2). Введем следующие обозначения:

- φ_{pre} — многогранник (точнее, система ограничений, описывающая многогранник), соответствующая предусловию цикла: $\varphi_{pre} = \{i = 2, j = 0\}$ — вырожденный многогранник, состоящий из одной точки $(2, 0)$;
- $\varphi_{begin}^{(i)}$ — многогранник перед выполнением i-ой итерации цикла;
- $\varphi_{then}^{(i)}$ — многогранник, соответствующий исполнению ветви **then** условного оператора на i-ой итерации цикла;
- $\varphi_{else}^{(i)}$ — многогранник, соответствующий исполнению ветви **else** условного оператора на i-ой итерации цикла;
- $\varphi_{end}^{(i)}$ — многогранник, соответствующий соединению обеих ветвей условного оператора на i-ой итерации цикла;
- $\varphi_{while}^{(i)}$ – многогранник, соответствующий соединению многогранников φ_{pre} и $\varphi_{end}^{(i)}$.

Таблица 7.2. Абстрактная интерпретация в теории выпуклых многогранников

i	Абстрактное состояние	Иллюстрация
0	$\varphi_{pre} = \{i = 2, j = 0\}$	
1	$\varphi^{(1)}_{begin} = \varphi_{pre} = \{i = 2, j = 0\}^*$ $\varphi^{(1)}_{then} = \{i = 6, j = 0\}$ $\varphi^{(1)}_{else} = \{i = 4, j = 1\}$ $\varphi^{(1)}_{end} = Conv(\varphi^{(1)}_{then}, \varphi^{(1)}_{else}) = \{i + 2 \cdot j = 6, 4 \le i \le 6\}$ $\varphi^{(1)}_{while} = Conv(\varphi_{pre}, \varphi^{(1)}_{end}) = \{2 \cdot j + 2 \le i, i + 2 \cdot j \le 6, 0 \le j\}$ * Нет многогранника для дуги обратной связи.	
2	$\varphi^{(2)}_{begin} = \varphi^{(1)}_{while} = \{2 \cdot j + 2 \le i, i + 2 \cdot j \le 6, 0 \le j\}^*$ $\varphi^{(2)}_{then} = \{2 \cdot j + 6 \le i, i + 2 \cdot j \le 10, 0 \le j\}$ $\varphi^{(2)}_{else} = \{2 \cdot j + 2 \le i, i + 2 \cdot j \le 10, 1 \le j\}$ $\varphi^{(2)}_{end} = Conv(\varphi^{(2)}_{then}, \varphi^{(2)}_{else}) = \{2 \cdot j + 2 \le i, 6 \le i + 2 \cdot j \le 10, 0 \le j\}$ $\varphi^{(2)}_{while} = Conv(\varphi_{pre}, \varphi^{(2)}_{end}) = \{2 \cdot j + 2 \le i, i + 2 \cdot j \le 6, 0 \le j\}$ * Расширяющий оператор не применяется, так как собрано недостаточно информации и расширение $\varphi^{(1)}_{begin} \nabla \varphi^{(1)}_{while}$ даст $\mathbb{R}^2$.	
3	$\varphi^{(3)}_{begin} = \varphi^{(2)}_{begin} \nabla \varphi^{(2)}_{while} = \{2j + 2 \le i, 0 \le j\}^*$ $\varphi^{(3)}_{then} = \{2j + 6 \le i, 0 \le j\}$ $\varphi^{(3)}_{else} = \{2j + 2 \le i, 1 \le j\}$ $\varphi^{(3)}_{end} = Conv(\varphi^{(3)}_{then}, \varphi^{(3)}_{else}) = \{2j + 2 \le i, 6 \le i + 2j, 0 \le j\}$ $\varphi^{(3)}_{while} = Conv(\varphi_{pre}, \varphi^{(3)}_{end}) = \{2j + 2 \le i, 0 \le j\} \subseteq \varphi^{(3)}_{begin}$ ** * Начинает применяться расширяющий оператор. ** Происходит стабилизация упрощенного представления.	

Обратите внимание: присваивание вида $x := x + C$, где C — некоторая константа соответствует сдвигу многогранника вдоль оси x на величину C (вправо, если $C > 0$). Если присваивание более сложное, ограничения, описывающие многогранник, меняются следующим образом:

- старое значение переменной выражается через новое;
- осуществляется подстановка в ограничение.

Например, для присваивания $x \coloneqq x + C$ подстановка имеет вид $x \coloneqq x - C$ (слева — старое значение, справа — новое). Если старое значение нельзя выразить через новое, т.е. когда при присваивании теряется информация, переменная исключается, используя оператор проецирования.

Итак, для рассматриваемого цикла с помощью абстрактной интерпретации в теории выпуклых многогранников извлекается инвариант $(2 \cdot j + 2 \leq i) \wedge (0 \leq j)$.

□

В заключение заметим, что методы синтеза инвариантов циклов на основе абстрактной интерпретации могут комбинироваться с эвристическими методами: если извлеченный с помощью абстрактной интерпретации инвариант оказывается слишком слабым, он может быть усилен путем применения эвристик.

7.4. Вопросы и упражнения

1. Перечислите основные шаги работы инструмента дедуктивной верификации программ. Какие из этих шагов легко автоматизируются, а какие нет? Поясните, в чем состоят основные трудности автоматизации дедуктивной верификации.
2. Напишите программу (на удобном вам языке программирования), помечающую дуги в связном ориентированном графе таким образом, чтобы каждый цикл содержал по крайней мере одну пометку (программа, которая для любого графа помечает все его дуги, решением не является).
3. Напишите программу (на удобном вам языке программирования), которая для заданного пути в графе потока управления, соединяющего начальную вершину с конечной, вычисляет формулу пути — условие на значения входных переменных, гарантирующее проход по этому пути.
4. Используя эвристики постройте инвариант цикла программы приближенного вычисления квадратного корня:

```
{a ≥ 0}
r:= 0;
while (r + 1)² ≤ a do
  r := r + 1
end
{0 ≤ r² ≤ a < (r + 1)²}
```

5. Используя эвристики постройте инвариант цикла программы дихотомического поиска в упорядоченном массиве (см. лекцию 4).

6. Напишите программу, которая по данному целому числу $n > 0$ находит наибольшее целое, которое, во-первых, является степенью двойки, а во-вторых, не больше n [Gri81]. Используя эвристики постройте инвариант цикла этой программы.

7. Опишите общую схему использования абстрактной интерпретации программ для поиска инвариантов циклов. Рассмотрите примеры.

8. Предположим, что условия B_1 и B_2 в рассмотренной в лекции программы имеют вид линейных ограничений. Каким образом их можно задействовать для абстрактной интерпретации?

9. Определите оператор расширения для интервальной арифметики.

10. Осуществите абстрактную интерпретацию следующей программы в интервальной арифметике [Cou77]:

```
x := 1;
while x ≤ 100 do
  x := x + 1
end
```

11. Используя абстрактную интерпретацию найдите инвариант цикла рассмотренной в лекции программы для предусловия $(i \geq 0) \wedge (j \geq 0) \wedge (i + j \leq 2)$.

Лекция 8. Автоматическое доказательство условий верификации

Со времен греков говорить «математика» — значит говорить «доказательство».

Н. Бурбаки.
Элементы математики

Рассматриваются основные подходы к автоматическому доказательству теорем, включая метод резолюций для логики высказываний и логики предикатов первого порядка. Вводятся понятия разрешающей процедуры и разрешимой теории, приводятся примеры разрешимых и неразрешимых теорий. Большое внимание уделяется принципам работы SAT- и SMT-решателей — средств, составляющих ядро инструментов формальной верификации: описывается алгоритм DPLL проверки выполнимости КНФ; приводится разрешающая процедура для теории равенства неинтерпретируемых функций; дается представление о методе Нельсона-Оппена комбинирования разрешающих процедур для разных теорий.

8.1. Формальные теории и разрешающие процедуры

В процессе дедуктивной верификации программы составляется множество логических формул (условий верификации и условий завершимости), общезначимость (тождественная истинность) которых свидетельствует о корректности программы (см. лекции 5 и 6). Как правило, истинность условий не является очевидной и должна быть доказана (или опровергнута). Доказательство может быть выполнено как «вручную» (как это делается на страницах этого пособия), так и с помощью специальных программных средств: *систем доказательства теорем* (*provers*), *систем помощи в доказательстве* (*proof assistants*), *систем проверки доказательств* (*proof checkers*) и *систем разрешения ограничений*, так называемых *решателей* (*solvers*)[47] [Har09].

Далее мы будем апеллировать к понятию теории. Термины «теория» и «дедуктивная система» являются синонимами. *Формальной системой*, или *теорией*, называется четверка

[47] Между системами автоматического доказательства теорем (provers) и системами разрешения ограничений (solvers) четкой границы нет: первый термин используют в отношении средств проверки общезначимости, результатом которых является вердикт (да или нет); второй термин применяют для средств проверки выполнимости, которые в случае выполнимости формулы возвращают ее модель — интерпретацию, в которой формула истинна. Очевидно, пруверы могут быть реализованы с помощью солверов.

$T = \langle \Sigma, \Phi, A, R \rangle$, где Σ — конечный или счетный *алфавит* символов; $\Phi \subseteq \Sigma^*$ — множество *формул*; $A \subseteq \Phi$ — множество *аксиом*; $R = \left\{ R^i \subseteq (\Phi \times \ldots \times \Phi) \times \Phi \right\}_{i=0}^{n-1}$ — конечное множество *правил вывода*. Обычно множество формул задается индуктивно, например, с помощью формальной грамматики.

Мы уже говорили (см. лекцию 3), что *теоремами* называются такие формулы, которые могут быть *доказаны*, т.е. выведены из аксиом с помощью правил вывода. В *значимых* теориях из доказуемости формулы вытекает ее общезначимость, а построение доказательства есть способ проверки общезначимости. В *полных* теориях доказательство существует для любой общезначимой формулы, однако процесс его поиска может быть нетривиальным.

Как правило, системы доказательства теорем не работают напрямую с аксиомами и правилами вывода, а используют специализированные процедуры.

Разрешающей процедурой теории T называется алгоритм DP (от англ. *Decision Procedure*), который для любой формулы $\varphi \in \Phi$ завершается с выдачей вердикта ($verdict$): $true$, если формула общезначима[48]; $false$ в противном случае [Kro08]:

$$\langle \varphi \in \Phi \rangle \; DP \; \langle (verdict = true) \leftrightarrow (\vDash \varphi) \rangle.$$

Частичной разрешающей процедурой теории T называется алгоритм DP, который для любой формулы $\varphi \in \Phi$ в случае завершения работы определяет, является она общезначимой или нет:

$$\{ \varphi \in \Phi \} \; DP \; \{ (verdict = true) \leftrightarrow (\vDash \varphi) \}.$$

Теория называется *разрешимой*, если для нее существует разрешающая процедура; в противном случае теория называется *неразрешимой*. Теория называется *частично разрешимой*, если для нее существует частичная разрешающая процедура.

Известно, что логика предикатов первого порядка неразрешима, но разрешима частично; точнее, *полуразрешима*: для необщезначимых формул процедура проверки может не за-

[48] В общем случае разрешающие процедуры имеет дело с произвольными проблемами, допускающими ответ «да» или «нет».

вершиться. Частичной разрешающей процедурой для этой логики является, например, *метод резолюций*, предложенный Джоном Аланом Робинсоном (John Alan Robinson, род. в 1930 г.)[49] [Rob65].

8.1.1. Примеры разрешимых и неразрешимых теорий

Важным направлением исследований в области автоматического доказательства теорем является поиск разрешимых теорий (прежде всего, *теорий первого порядка* — специализаций логики предикатов первого порядка) и создание для них эффективных разрешающих процедур. Для более глубокого изучения вопросов разрешимости и неразрешимости теорий, можно порекомендовать работы [Har09] и [Kro08]. Мы же здесь ограничимся примерами (см. таблицы 8.1 и 8.2).

Таблица 8.1. Примеры разрешимых теорий

Разрешимая теория	Кем и когда доказана разрешимость
Булева алгебра	Альфред Тарский (Alfred Tarski), 1949 г.
Евклидова геометрия	Альфред Тарский, 1949 г.
Гиперболическая геометрия	Вольфрам Швабхаузер (Wolfram Schwabhäuser), 1959 г.
Теория первого порядка с равенством	Леопольд Лёвенгейм (Leopold Löwenheim), 1915 г.
Теория первого порядка с равенством и одним унарным функциональным символом	Анджей Эренфойхт (Andrzej Ehrenfeucht), 1959 г.
Арифметика Пресбургера (теория первого порядка, описывающая натуральные числа с равенством, сложением, но без умножения)	Мойжеш Пресбургер (Mojżesz Presburger), 1929 г.
Теория равенства неинтерпретируемых функциональных символов (бескванторная теория первого порядка с равенством и произвольным числом функциональных символов)	Вильгельм Фридрих Аккерман (Wilhelm Friedrich Ackermann), 1954 г.
Бескванторная теория массивов (теория первого порядка с равенством, операцией получения элемента по индексу и операцией модификации элемента)	Джон Маккарти (John McCarthy), 1962 г.

[49] Фамилию Робинсон имеют несколько известных математиков-логиков: Джон Алан Робинсон (автор метода резолюций), Рафаэль Митчел Робинсон (чьим именем назван неразрешимый фрагмент арифметики Пеано), Абрахам (Авраам) Робинсон (создатель нестандартного анализа), Джулия Холл Робинсон (жена Рафаэля Робинсона, внесшая большой вклад в решение десятой проблемы Гилберта) и Джордж Робинсон (автор некоторых техник оптимизации в процедурах автоматического доказательства теорем).

Таблица 8.2. Примеры неразрешимых теорий

Неразрешимая теория	Кем и когда доказана неразрешимость
Теория первого порядка с равенством и либо одним предикатным символом арности не меньше двух, либо одним функциональным символом арности не меньше двух, либо двумя унарными функциональными символами	Борис Абрамович Трахтенброт, 1953 г.
Арифметика натуральных чисел	Джон Баркли Россер (John Barkley Rosser), 1936 г.
Арифметика целых чисел	Альфред Тарский, Анджей Мостовский (Andrzej Mostowski), 1949 г.
Арифметика рациональных чисел	Джулия Холл Робинсон (Julia Hall Robinson), 1949 г.
Теории групп и колец	Альфред Тарский, 1949 г.
Теория полей	Джулия Холл Робинсон, 1949 г.
Арифметика Робинсона (фрагмент арифметики Пеано без схемы индукции)	Рафаэль Митчел Робинсон (Raphael Mitchel Robinson), 1950 г.
Общая теория массивов (теория первого порядка с равенством, операцией получения элемента по индексу и операцией модификации элемента)	Джон Маккарти, 1962 г.

8.1.2. Задача выполнимости, SAT- и SMT-решатели

Вспомним основополагающие определения математической логики. Формула называется *выполнимой*, если существует *интерпретация*, в которой она истинна (такие интерпретации называются *моделями*). Формула называется *общезначимой*, если она истинна в любой интерпретации. Формула называется *невыполнимой* или *противоречивой*, если она ложна во всех интерпретациях. Формула называется *необщезначимой* или *опровержимой*, если существует интерпретация, в которой она ложна. Связь между выполнимостью и общезначимостью описывается следующими утверждениями:

- формула φ выполнима тогда и только тогда, когда $\neg\varphi$ необщезначима;
- формула φ общезначима тогда и только тогда, когда $\neg\varphi$ невыполнима.

Для логики высказываний задача проверки выполнимости называется *SAT* (по трем первым буквам англ. слова *satisfiability* — выполнимость). Поскольку число интерпретаций любой формулы логики высказываний конечно, задача SAT разрешима, но, вообще говоря, является вычислительно трудной (NP-полной) [Pap82]. Задача проверки выполнимости для теорий первого порядка называется *SMT* (*Satisfiability Modulo Theories* — выполнимость с уче-

том теорий). Соответственно, инструменты, проверяющие выполнимость формул для логики высказываний и теорий первого порядка, называются *SAT-* и *SMT-решателями*. Слово «решатель» говорит о том, что для проверки выполнимости формулы инструмент пытается решить описываемое ей ограничение, т.е. построить возможную модель (означивание переменных).

Ниже мы рассмотрим базовые алгоритмы работы SAT- и SMT-решателей. Для более детального изучения этой темы можно обратиться к работам [Har09], [Kro08] и [Cha73].

8.2. Логика высказываний и SAT-решатели

В классической постановке задача выполнимости рассматривается для утверждений, представленных в *конъюнктивной нормальной форме* (*КНФ*) или, что равнозначно, в *клаузальной форме*. *Клаузой*, или *дизъюнктом*, называется множество *литералов* — элементарных высказываний (*атомов*) и их отрицаний. Формулой в *клаузальной форме* называется множество клауз: каждая клауза соответствует дизъюнкции литералов, а формула — конъюнкции дизъюнктов [Ben12]. Пустая клауза обозначается □, а пустая формула —∅.

Пример 8.1

Рассмотрим примеры клаузальных форм:

- $\emptyset \equiv true$ — пустая формула;
- $\{\square\} \equiv false$ — формула, состоящая из пустой клаузы;
- $\{pr, \bar{q}\bar{p}q, p\bar{p}q\}$[50] $\equiv (p \vee r) \wedge (\neg q \vee \neg p \vee q) \wedge (p \vee \neg p \vee q)$.

□

8.2.1. Метод резолюций для логики высказываний

Для опровержения (доказательства невыполнимости) клаузальных форм применяется процедура, называемая *резолюцией* [Ben12]. Процедура состоит в последовательном применении *правила резолюции*, сохраняющем свойство выполнимости: формула F', полученная из формулы F в результате применения этого правила, выполнима тогда и только тогда,

[50] Для удобства при записи клауз будем опускать скобки и запятые, а отрицание высказывания будем обозначать чертой над ним. Например, вместо $\{\neg q, \neg p, q\}$ будем писать $\bar{q}\bar{p}q$.

когда выполнима F; говорят что формулы F' и F *равносильны* (обозначение: $F' \approx F$)[51]. Если на некотором шаге выводится клауза □, то исходная формула невыполнима (соответственно, ее отрицание общезначимо).

Пусть клауза C_1 содержит литерал l, а клауза C_2 — *контрарный* литерал l^C:

- $l^C = \neg p$, если $l = p$;
- $l^C = p$, если $l = \neg p$.

Клаузы C_1 и C_2 называются *сталкивающимися* по *контрарной паре* l и l^C, а клауза

$$Res(C_1, C_2) = (C_1 \setminus \{l\}) \cup (C_2 \setminus \{l^C\})$$

называется *резольвентой* C_1 и C_2.

Правило резолюции состоит в добавлении во множество клауз, некоторые из которых сталкиваются, какой-нибудь резольвенты.

Пример 8.2

Клаузы $ab\bar{c}$ и $bc\bar{e}$ сталкиваются по контрарной паре $\overline{c}$ и c. Соответствующая резольвента определяется равенством $Res(ab\bar{c}, bc\bar{e}) = ab \cup b\bar{e} = ab\bar{e}$.

□

Процедура резолюции описывается следующим псевдокодом:

```
Вход:  клаузальная форма F, не содержащая клаузы □
Выход: true (F выполнима) или false (F невыполнима)

while F содержит сталкивающиеся клаузы do
  выбрать из F пару сталкивающихся клауз C1 и C2;
  C := Res(C1, C2);
  if C = □ then
    return false
  end;
  F := F ∪ {C}
end;

return true
```

[51] Не путайте равносильность, т.е. одновременную выполнимость или невыполнимость ($F' \approx F$) с эквивалентностью ($F' \leftrightarrow F$).

Резолютивный вывод клаузы C из формулы F есть конечная последовательность клауз $\{C_i\}_{i=1}^{k}$, такая что:

1. $C_k = C$;
2. для всех $i \in \{1, \dots, k\}$:
    a. либо $C_i \in F$;
    b. либо C_i — резольвента клауз, предшествующих C_i.

Вывод □ из F называется *опровержением* F [Cha73]. Процедура резолюции является значимой и полной системой опровержения формул [Ben12].

Пример 8.3

Опровергнем формулу $\{p, \bar{p}q, \bar{r}, \bar{p}\bar{q}r\}$, используя метод резолюций [Ben12]. Вот одно из возможных решений:

- $C_1 = p$;
- $C_2 = \bar{p}q$;
- $C_3 = \bar{r}$;
- $C_4 = \bar{p}\bar{q}r$;
- $C_5 = Res(C_3, C_4) = Res(\bar{r}, \bar{p}\bar{q}r) = \bar{p}\bar{q}$;
- $C_6 = Res(C_2, C_5) = Res(\bar{p}q, \bar{p}\bar{q}) = \bar{p}$;
- $C_7 = Res(C_1, C_6) = Res(p, \bar{p}) = \square$.

□

8.2.2. Алгоритм DPLL проверки выполнимости

Пожалуй, самым известным алгоритмом проверки выполнимости формул логики высказываний (простой перебор не в счет) является алгоритм DPLL (1962 г.), названный в честь его авторов [Dav62]:

- D — Мартин Дэвис (Martin Davis, род. в 1928 г.);
- P — Хилари Патнем (Hilary Putnam, 1926-2016);
- L — Джорж Логеман (George Logemann, 1938-2012);
- L — Дональд Лавленд (Donald Loveland, род. в 1934 г.).

DPLL представляет собой усовершенствованную версию еще более старого алгоритма Дэвиса-Патнема (1960 г.) [Dav60]. Алгоритм основан на очень простых правилах вывода, одно из которых — распространение единицы — является частным случаем правила резолюции.

Правило 0 — удаление тавтологий

Клауза, включающая в себя контрарные литералы, называется *тривиальной клаузой* или *тавтологией*.

Пусть формула F' получена из формулы F удалением всех тавтологий. Тогда $F' \approx F$.

Правило 1 — распространение единицы

Клауза, состоящая из одного единственного литерала, называется *единичной клаузой*.

Пусть $\{l\}$ — единичная клауза в формуле F, а формула F' получена из F удалением всех клауз, содержащих l, и удалением l^C из всех оставшихся клауз. Тогда $F' \approx F$.

Правило 2 — исключение чистых литералов

Чистым литералом в формуле F называется литерал l, такой что l входит хотя бы в одну клаузу F, а l^C не входит ни в одну клаузу F.

Пусть l — чистый литерал в формуле F, а формула F' получена из F удалением всех клауз, содержащих l. Тогда $F' \approx F$.

Ниже приведен псевдокод алгоритма DPLL. Функция $DPLL$ после удаления тривиальных клауз вызывает рекурсивную функцию $DPLL'$, имеющую два параметра:

- F — проверяемая формула;
- I — частичная интерпретация (означивание части элементарных высказываний).

Функция возвращает $true$, если формула выполнима (пример модели содержится в переменной I'), и $false$ в противном случае.

В псевдокоде используются следующие обозначения: $p^{true} \equiv p$ и $p^{false} \equiv \neg p$.

```
Функция DPLL
Вход:  клаузальная форма F
Выход: true (F выполнима) или false (F невыполнима)
  /* правило 0 */
  F' := удалить_тавтологии(F);
  DPLL'(F', ∅)
```

```
Функция DPLL'
Вход:  клаузальная форма F, частичная интерпретация I
Выход: true (F выполнима) или false (F невыполнима)

F' := F;
I' := I;

/* правило 1 */
while F' содержит единичные клаузы do
  выбрать единичную клаузу {p^x};
  F' := распространить_единицу(F', p^x);
  I' := I'[p := x]
end;

/* правило 2 */
while F' содержит чистые литералы do
  выбрать чистый литерал p^x;
  F' := исключить_чистый_литерал(F', p^x);
  I' := I'[p := x]
end;

/* конфликт */
if □ ∈ F' (F' содержит клаузу, невыполнимую в I') then
  return false
end;

/* решение */
if F' = ∅ (F' выполнима в I') then
  return true
end;

/* ветвление */
выбрать элементарное высказывание p, входящее в F';
выбрать значение истинности x ∈ {true, false};

/* вычисления исполняются по правилам «короткой» логики */
return DPLL'(F'[p := x], I'[p := x]) ∨ DPLL'(F'[p := ¬x], I'[p := ¬x])
```

Когда уточняется формула F', т.е. когда некоторое элементарное высказывание p сопоставляется с некоторым значением истинности x (обозначение: $F'[p \coloneqq x]$), из формулы F' удаляются клаузы, содержащие литерал p^x, а из оставшихся клауз удаляются контрарные литералы $p^{\neg x}$ (аналогично тому, как это происходит при распространении единицы).

На время работы алгоритма влияет высказывание, по которому осуществляется ветвление, и сопоставляемое ему значение истинности. Таким образом, DPLL — это не один алгоритм, а семейство алгоритмов. Несмотря на почтенный возраст, DPLL активно используется в современных SAT-решателях, например, в MiniSat [MinS]. Ветвления управляются *эвристиками*. Для оптимизации также используют *нехронологические возвраты* и запоминание

конфликтов — частичных интерпретаций, фальсифицирующих формулу [Yua06]. Такие варианты DPLL получили название CDCL (Conflict-Driven Clause Learning).

Есть и другие подходы к задаче SAT: метод на основе *двоичных решающих диаграмм* (*BDD*, *Binary Decision Diagrams*) (см. лекцию 14) и метод Сталмарка (Gunnar Stålmarck) [Har09].

8.3. Теории первого порядка и SMT-решатели

Метод резолюций может использоваться и для логики предикатов первого порядка. Общий вариант метода был предложен в 1965 г. Джоном Аланом Робинсоном [Rob65]. Известно, что любую формулу логики предикатов первого порядка можно свести к равносильной клаузальной форме. Это делается в несколько этапов [Cha73]:

1. приведение формулы к *предваренной нормальной форме* (*ПНФ*) — переименование связанных переменных (цель — сделать их уникальными) и перемещение кванторов в начало формулы;
2. преобразование бескванторной части формулы, так называемой *матрицы*, к КНФ;
3. приведение формулы к *сколемовской стандартной форме* (*ССФ*) — элиминация кванторов существования с помощью *сколемовских функций*[52].
4. запись матрицы ССФ в клаузальной форме.

8.3.1. Метод резолюций для логики предикатов первого порядка

Элементарными формулами, или *атомами*, в логике предикатов первого порядка называются формулы вида $P(t_1, \dots, t_n)$, где P — предикатный символ, а $t_1, \dots, t_n$ — термы. Как и раньше, *литералы* — это атомы и их отрицания. Общая схема метода резолюций остается прежней, но поиск контрарных литералов становится сложнее и требует привлечения механизма *унификации*.

Подстановка θ называется *унификатором* для множества формул $\{F_1, \dots, F_n\}$, если

[52] Название связано именем Торальфа Альберта Сколема (Thoralf Albert Skolem, 1887-1963), предложившего следующий подход к элиминации кванторов существования: при элиминации $\exists y$ переменная y заменяется на терм $f(x_1, \dots, x_n)$, где f — новый функциональный символ (так называемая *сколемовская функция*), а $x_1, \dots, x_n$ — переменные кванторов всеобщности, синтаксически предшествующие $\exists y$.

$$F_1\theta = \cdots = F_n\theta.$$

Унификатор θ называется *наиболее общим унификатором* (*НОУ*), если любой другой унификатор может быть представлен в виде суперпозиции θ и некоторой подстановки [Ben12].

Пусть C_1 и C_2 — клаузы, не имеющие общих переменных[53], а $L_1 = \{l_{1,i}\}_{i=1}^{n}$ и $L_2 = \{l_{2,i}\}_{i=1}^{m}$ — их подмножества, такие что L_1 и $L_2^C = \{l_{2,i}^C\}_{i=1}^{m}$ могут быть унифицированы. Пусть θ — НОУ L_1 и L_2^C. Тогда, клаузы C_1 и C_2 называются *сталкивающимися*, а клауза

$$Res(C_1, C_2) = (C_1\theta \setminus L_1\theta) \cup (C_2\theta \setminus L_2\theta)$$

называется *резольвентой* C_1 и C_2.

Процедура резолюции является значимой и полной для опровержения формул логики предикатов первого порядка, но может не завершиться для выполнимых формул. Известны оптимизации и частные случаи метода резолюций, с которыми можно познакомиться, например, в книге [Cha73].

Для построения НОУ можно использовать алгоритм, предложенный в 1979 г. Альберто Мартелли (Alberto Martelli) и Уго Монтанари (Ugo Montanari, род. в 1943 г.) [Mar82]. Алгоритм унификации Мартелли-Монтанари эффективнее первоначального алгоритма, используемого Робинсоном.

Два атома могут быть унифицированы, только если у них одинаковый предикатный символ. Таким образом, унификация атомов сводится к решению системы уравнений в термах: по одному уравнению на каждый аргумент предиката. На вход алгоритму подается система уравнений, которая преобразуется по следующим правилам:

1. изменить уравнение $t = x$, где t — терм, не являющийся переменной, а x — переменная, на $x = t$;
2. удалить уравнение $x = x$, где x — переменная;
3. для уравнения $t_1 = t_2$, где t_1 и t_2 — термы, не являющиеся переменными:
    a. если внешние функциональные символы t_1 и t_2 различаются:

[53] Для обеспечения корректности метода резолюций в каждой клаузе следует использовать свою уникальную систему переменных. Переменные клаузы неявно связаны кванторами всеобщности, что позволяет менять их имена без изменения смысла формулы.

i. *система не имеет решения*;

b. иначе, если уравнение имеет вид $f(t_1^1, \dots, t_1^k) = f(t_2^1, \dots, t_2^k)$:

i. заменить уравнение $t_1 = t_2$ на систему уравнений $\{t_1^i = t_2^i\}_{i=1}^k$;

4. Для уравнения $x = t$, где x — переменная, имеющая другие вхождения, а t — терм:

a. если x входит в t (предполагается, что терм t отличен от x):

i. *система не имеет решения*;

b. иначе, если x не входит в t:

i. заменить все другие вхождения x на t.

Результатом работы алгоритма (при наличии решения) является система в разрешенном виде: в левой части равенств располагаются переменные, не имеющие вхождений в правую часть. Эта система и задает НОУ.

8.3.2. Теория равенства

Говоря о разрешающих процедурах для теорий первого порядка, отметим, что единого подхода к их построению нет: для каждой теории ищется своя процедура, как можно более эффективная. В лекции мы рассмотрим только одну теорию — *теорию равенства* [Nel80].

Полное название теории — *бесквантворная теория равенства неинтерпретируемых функциональных символов*. Ее сигнатура включает произвольное число функциональных символов, в частности констант, и один двуместный предикатный символ $=$. Имеются аксиомы рефлективности, симметричности и транзитивности равенства, а также *аксиомы подстановочности* [Неп88]:

$$\left(\bigwedge_{i=1}^{k}(t_1^i = t_2^i)\right) \to f(t_1^1, \dots, t_1^k) = f(t_2^1, \dots, t_2^k).$$

Разрешимость теории равенства была доказана в 1954 г. Вильгельмом Фридрихом Аккерманом (Wilhelm Friedrich Ackermann, 1896-1962), а эффективная разрешающая процедура была предложена в 1976 г. Грегом Нельсоном (Greg Nelson) и Дереком Оппеном (Derek Oppen) [Nel80]. Процедура основана на построении *конгруэнтного замыкания* отношения на графе; в настоящее время известны алгоритмы с вычислительной сложностью порядка $O(n \log n)$, где n — число дуг графа [Nei07].

Ориентированным графом с помеченными вершинами и упорядоченными дугами называется пятерка $G = \langle V, E, \Sigma, \lambda, [] \rangle$, в которой:

- V — конечное множество вершин;
- E — мультимножество дуг;
- Σ — множество пометок вершин;
- $\lambda: V \to \Sigma$ — функция разметки вершин;
- $[]: V \times \mathbb{N} \to V$ — функция нумерации концевых вершин:
    - если $v \in V$ и $1 \leq i \leq \delta(v)$, где $\delta(v)$ — число дуг, исходящих из v:
        - $v[i]$ — конец i-ой исходящей из v дуги;
        - концевые вершины с разными номерами могут совпадать.

Пусть на множестве вершин графа задано бинарное отношение R. Две вершины v и u называются *конгруэнтными* по отношению R, если:

1. $\lambda(v) = \lambda(u)$;
2. $\delta(v) = \delta(u)$;
3. для всех $1 \leq i \leq \delta(v)$:
    a. либо $v[i] = u[i]$;
    b. либо $(v[i], u[i]) \in R$.

Отношение R называется *замкнутым относительно конгруэнтности*, если для всех вершин v и u, конгруэнтных по отношению R, имеет место принадлежность $(v, u) \in R$. *Конгруэнтным замыканием* отношения R называется минимальное отношение эквивалентности, содержащее R и замкнутое относительно конгруэнтности.

Напомним: *отношением эквивалентности* называется бинарное отношение, удовлетворяющее свойствам рефлексивности, симметричности и транзитивности.

Проверка общезначимости формулы F равносильна проверке невыполнимости $\neg F$. Если F представлена в КНФ, то $\neg F$ может быть легко представлена в ДНФ[54]. ДНФ невыполнима

[54] Это достигается применением правил де Моргана: $\overline{(a \vee b)} \equiv (\overline{a} \wedge \overline{b})$ и $\overline{(a \wedge b)} \equiv (\overline{a} \vee \overline{b})$.

тогда и только тогда, когда невыполнимы все дизъюнктивные члены. Каждый из них имеет вид конъюнкции [Неп88]:

$$(t_1 = t_1' \ ) \wedge \ldots \wedge \left(t_p = t_p'\right) \wedge (s_1 \neq s_1') \wedge \ldots \wedge (s_q \neq s_q').$$

Здесь t_i, t_i', s_j и s_j', где $i \in \{1, \ldots, p\}$ и $j \in \{1, \ldots, q\}$, — некоторые термы, а запись $x \neq y$ есть сокращение $\neg(x = y)$. Обработка конъюнкции состоит из следующих шагов:

1. построить граф G над множеством термов (вершинами графа являются термы, входящие в конъюнкцию, и их подтермы, а пометками — функциональные символы и переменные):
    a. пометить каждый элементарный терм v, где v — переменная или константа:
        i. $\lambda(v) = v$;
    b. пометить каждый терм $f(v_1, \ldots, v_k)$, где $v_1, \ldots, v_k$ — некоторые термы:
        i. $\lambda(f(v_1, \ldots, v_k)) = f$;
    c. для каждого терма $f(v_1, \ldots, v_k)$ добавить дуги $(f(v_1, \ldots, v_k), v_i)$:
        i. $f(v_1, \ldots, v_k)[i] = v_i$, для всех $i \in \{1, \ldots, k\}$;
2. построить конгруэнтное замыкание R^* отношения $R = \left\{ (t_i, t_i') \mid i \in \{1, \ldots, p\} \right\}$:
    a. положить отношение R^* равным R;
    b. пока возможно, добавить в R^*пару $(v, u) \notin R^*$, такую что истинно хотя бы одно из следующих условий[55]:
        i. v и u конгруэнтны по отношению R^*;
        ii. $(u, v) \in R^*$;
        iii. $(v, w) \in R^*$ и $(w, u) \in R^*$ для некоторой вершины w;
3. если для всех $i \in \{1, \ldots, q\}$ термы s_i и s'_i различны и $(s_i, s'_i) \notin R^*$:
    a. конъюнкция выполнима;
4. иначе:
    a. конъюнкция невыполнима.

[55] Имеются более эффективные алгоритмы построения конгруэнтного замыкания (см., например, [Nei07]).

8.3.3. Комбинирование теорий

Условии верификации и условии завершимости программ часто выражаются в терминах нескольких теорий [Неп88], включая целочисленную арифметику, операции над массивами, функциональные уравнения. Чтобы оперировать с такими формулами нужен метод, который по разрешающим процедурам отдельных теорий конструирует разрешающую процедуру для их совокупности. Один из таких методов был предложен в 1979 г. уже упомянутыми Нельсоном и Оппеном [Nel79]. Метод применим для бесквантворных теорий, не имеющих общих функциональных и предикатных символов за исключением символа равенства, обязательного для всех комбинируемых теорий.

Идея метода Нельсона-Оппена состоит в следующем. По исходной формуле строится множество формул, каждая из которых относится к какой-то одной теории (для этого может потребоваться введение новых переменных). Если хотя бы одна формула невыполнима, то невыполнима и исходная формула. Пока формулы выполнимы, из них выводятся равенства переменных, которые используются для уточнения других формул. Если достигается насыщение (новые равенства не выводятся), исходная формула выполнима.

Говорят, что из формулы F *выводится равенство* $x = y$, если

$$\vDash F \to (x = y).$$

Заметим: общезначимость такой импликации может быть установлена разрешающей процедурой соответствующей теории.

Метод применим (является значимым и полным) только для *выпуклых теорий*, т.е. теорий, удовлетворяющих следующему свойству: из формулы теории можно вывести непустую дизъюнкцию равенств тогда и только тогда, когда из нее можно вывести хотя бы одно равенство этой дизъюнкции в отдельности [Неп88]. Примеры выпуклых и невыпуклых теорий приведены в таблице 8.3.

Таблица 8.3. Примеры выпуклых и невыпуклых теорий

Выпуклая теория	Невыпуклая теория
Линейная арифметика (сложение, умножение на константу, сравнения $=, \neq, \leq, \geq, <, >$) над $\mathbb{Q}$	Линейная арифметика над $\mathbb{Z}$
Теория равенства	Теория массивов

Пример 8.4

Поясним работу метода Нельсона-Оппена на примере. Рассмотрим формулу, включающую арифметические соотношения (линейная арифметика над $\mathbb{Q}$) и функциональное неравенство (теория равенства) [Неп88]:

$$F \equiv (x \le y) \wedge (y + z \le x) \wedge (z \ge 0) \wedge \Big(f\big(f(x) - f(y)\big) \ne f(z)\Big).$$

Вначале по формуле F строятся две формулы: F_1 и F_2 ($F \approx F_1 \wedge F_2$):

$$F_1 \equiv (x \le y) \wedge (y + z \le x) \wedge (z \ge 0) \wedge (w_1 = w_2 - w_3);$$

$$F_2 \equiv (w_2 = f(x)) \wedge (w_3 = f(y)) \wedge \big(f(w_1) \ne f(z)\big).$$

Обратите внимание: каждая из этих формул выполнима, а их конъюнкция — нет. Из арифметических соотношений вытекает, что $x = y$ и $z = 0$, следовательно, $w_2 = w_3$ и $w_1 = 0$, а значит, $f(w_1) = f(z)$. Чтобы прийти к такому выводу необходим обмен информацией между разрешающими процедурами двух теорий.

Как происходит обмен? Из F_1 выводится равенство $x = y$, которое добавляется в F_2:

$$F_2' \equiv (w_2 = f(x)) \wedge (w_3 = f(y)) \wedge \big(f(w_1) \ne f(z)\big) \wedge (x = y).$$

Из F_2' выводится равенство $w_2 = w_3$, которое добавляется в F_1:

$$F_1' \equiv (x \le y) \wedge (y + z \le x) \wedge (z \ge 0) \wedge (w_1 = w_2 - w_3) \wedge (w_2 = w_3).$$

Из F_1' выводится равенство $w_1 = z$ ($w_1 = z = 0$), которое добавляется в F_2':

$$F_2'' \equiv (w_2 = f(x)) \wedge (w_3 = f(y)) \wedge \underline{\big(f(w_1) \ne f(z)\big)} \wedge (x = y) \ \wedge \underline{(w_1 = z)}.$$

Формула F_2'' невыполнима, откуда следует невыполнимость исходной формулы.

□

8.4. Вопросы и упражнения

1. Дайте определение разрешимой теории. Приведите примеры разрешимых и неразрешимых теорий.
2. Докажите что резольвента $Res(C_1, C_2)$ выполнима тогда и только тогда, когда выполнима формула $\{C_1, C_2\}$.

3. Покажите, что правило распространения единицы является частным случаем правила резолюции.
4. Используя метод резолюций, постройте два опровержения формулы $\{\bar{p}\bar{q}r, pr, qr, \bar{r}\}$: в первом случае придерживайтесь порядка p, q, r для сталкивания клауз; во втором — r, q, p [Ben12].
5. Приведите к ПНФ следующие формулы (A и B — бесквантерные формулы) [Лав09]:
   a. $\neg\exists x\forall y\exists z\forall u\ A$;
   b. $\exists x\forall y\ A(x,y) \wedge \exists x\forall y\ B(x,y)$;
   c. $\exists x\forall y\ A(x,y) \vee \exists x\forall y\ B(x,y)$;
   d. $\exists x\forall y\ A(x,y) \rightarrow \exists x\forall y\ B(x,y)$.
6. Приведите к ССФ следующие формулы [Лав09]:
   a. $\exists x\forall y\ Q(x,y) \rightarrow \exists x\forall y\ Q(x,y)$;
   b. $\exists x\forall y\exists z\forall v\ R(x,y,z,v)$;
   c. $\forall x\exists y\forall v\exists z\ R(x,y,z,v)$.
7. Постройте НОУ (если возможно) для следующих пар атомов [Ben12]:
   a. $p\big(a, x, f\big(g(y)\big)\big)$ $\quad$ $p\big(y, f(z), f(z)\big)$;
   b. $p\big(x, g\big(f(a)\big), f(x)\big)$ $\quad$ $p(f(a), y, y)$;
   c. $p\big(x, g\big(f(a)\big), f(x)\big)$ $\quad$ $p(f(y), z, y)$;
   d. $p\big(a, x, f\big(g(y)\big)\big)$ $\quad$ $p\big(z, h(z,u), f(u)\big)$.
8. Напишите программу (на удобном вам языке программирования), реализующую процедуру резолюцию для логики высказываний.
9. Напишите программу, реализующую алгоритм DPLL проверки разрешимости КНФ.
10. Напишите программу, реализующую алгоритм унификации Мартелли-Монтанари.
11. Напишите программу, реализующую процедуру резолюции для логики предикатов первого порядка.
12. Напишите программу, реализующую разрешающую процедуру для бесквантерной теории равенства неинтерпретируемых функциональных символов.

13. Какие теории называются выпуклыми. Приведите примеры выпуклых и невыпуклых теорий.

14. Покажите, что линейная арифметика над множеством целых чисел и теория массивов не являются выпуклыми теориями.

Лекция 9. Спецификация и верификация параллельных программ

Существует обширный печальный опыт того, что параллельные программы весьма упорно сопротивляются серьезным усилиям по выявлению в них ошибок.

С. Овицки, Л. Лэмпорт.
Доказательство свойств живости в параллельных системах.

Рассматриваются параллельные программы — программы, состоящие из нескольких процессов, исполняемых совместно и взаимодействующих друг с другом через общие переменные. Семантика таких программ определяется с помощью парадигмы чередования процессов. Особое внимание уделяется так называемым реагирующим системам — программам особого типа, работающим в «бесконечном цикле» и реагирующим на события окружения путем исполнения тех или иных действий. Описывается один из формализмов, применяемых для спецификации реагирующих систем — темпоральная логика линейного времени (LTL, Linear-time Temporal Logic). Для иллюстрации методов спецификации и верификации параллельных программ используется алгоритм Петерсона взаимного исключения процессов.

9.1. Параллельные программы

До настоящего момента мы рассматривали *последовательные* программы. Значительную долю реального ПО составляют программы, состоящие из нескольких *параллельно* функционирующих и взаимодействующих процессов: программы обработки больших массивов данных (в том числе программы, выполняющие численные расчеты), клиент-серверные приложения, разного рода системы управления и т.д.

Разработка параллельной программы — непростая задача, требующая помимо прочего аккуратной *синхронизации* процессов. Дело в том, что ошибки, связанные с синхронизацией, крайне сложны для обнаружения — они проявляются лишь при определенном порядке исполнения операторов процессов, в то время как число вариантов исполнения огромно[56]. Известный пример — случай с аппаратом лучевой терапии «Therac-25», когда ошибка синхронизации стала причиной переоблучения и смерти нескольких человек (см. лекцию 1).

[56] Явление экспоненциального роста числа состояний параллельной программы от числа процессов известно как *комбинаторный взрыв числа состояний* (*state explosion*).

Не будет преувеличением сказать, что для параллельных программ формальные методы верификации особенно актуальны — ошибки синхронизации трудно выявить, используя лишь традиционное тестирование [Кар10].

9.1.1. Параллельные программы над общей памятью

Рассмотрим следующую упрощенную модель. *Параллельной программой*, или просто *программой*, называется конечное множество последовательных программ над общим множеством переменных. Отдельная программа этого множества называется *процессом*. Переменные, используемые только в одном процессе, называются *локальными*; переменные, используемые в нескольких процессах, — *разделяемыми* или *глобальными*.

Параллельная программа $\{P_1, \dots, P_n\}$ обозначается также как $P_1 \parallel \cdots \parallel P_n$. Для удобства будем считать, что процессы программы занумерованы: P_i — процесс с *номером* (*идентификатором*) $i \in \{1, \dots, n\}$. Каждый оператор программы помечен меткой, уникальной в рамках одного процесса. Метка записывается перед оператором и отделяется от него двоеточием: $label\colon statement$. Множество всех возможных меток будем обозначать символом L.

Конфигурацией программы $P_1 \parallel \cdots \parallel P_n$ называется пара $\langle l, s \rangle$, где $l\colon \{1, \dots, n\} \to L$ — *состояние управления*, а $s\colon V \to D$ — *состояние данных* (здесь V — множество переменных программы, а D — универсальный домен)[57]. Запись $P@l$, или просто $@l$, если из контекста ясно, о каком процессе идет речь, обозначает тот факт, что исполнение процесса P «находится» на операторе, помеченном меткой l (говорят — в *точке* l). Если $L' \subseteq L$, запись $P@L'$ ($@L'$) означает, что исполнение процесса P находится в одной из точек множества L'.

9.1.2. Семантика чередований и справедливость планировщика

Семантика, или *модель вычислений*, параллельной программы может быть определена, используя парадигму *чередования* (*интерливинга*), известную также как *асинхронный па-*

[57] Конфигурацию параллельной программы можно определить и без использования меток, как это сделано в лекции 2 при рассмотрении структурной операционной семантики.

раллелизм. Пусть процессы описаны на языке программирования с формализованной операционной семантикой, например, на языке while (см. лекцию 2). Тогда отношение переходов программы $P_1 \parallel \cdots \parallel P_n$ определяется следующим семейством правил вывода:

$$\frac{l_i \neq l_\omega, \langle l_i, s\rangle \underset{i}{\rightarrow} \langle l_i', s'\rangle}{\langle l, s\rangle \rightarrow \langle l[i := l_i'], s'\rangle}, \qquad i \in \{1, \ldots, n\}.$$

Здесь $l_i \equiv l(i)$ — метка текущего оператора процесса P_i, l_ω — специальная метка, символизирующая завершение процесса, $\underset{i}{\rightarrow}$ — отношение переходов процесса P_i.

Видно, что поведение параллельной программы *недетерминировано* — оно зависит от выбора процесса для исполнения. Стратегию выбора процесса, также как и сущность, реализующую эту стратегию, мы будем называть *планировщиком*. Реальные планировщики (скажем, планировщики в операционных системах) могут быть вполне детерминированными, однако программы (пользовательские приложения) должны быть рассчитаны на работу в разном окружении, что означает, что они должны быть корректными на всех возможных вычислениях (см. лекцию 2).

Используя концепцию планировщика, исполнение параллельной программы можно записать следующим образом:

```
⟨l, s⟩ := ⟨l₀, s₀⟩;          /* инициализируется программа */
P := {i | lᵢ ≠ lω};          /* вычисляется множество активных процессов */
while P ≠ ∅ do               /* пока есть активные процессы,
  p := schedule(P);             выбирается один из них */
  ⟨l, s⟩ := execute(p);      /* исполняется оператор выбранного процесса */
  P := {i | lᵢ ≠ lω}         /* перевычисляется множество процессов */
end
```

Пример 9.1

Рассмотрим программу $P \equiv P_1 \parallel P_2$, в которой

$$P_1 \equiv l_1 : x := y \quad \text{и} \quad P_2 \equiv l_2 : y := x,$$

с условием корректности $\langle (x = a) \wedge (y = b)\rangle P \langle (x = b) \wedge (y = a)\rangle$, где a и b — некоторые константы. Выражаясь неформально, от программы P требуется поменять местами значения переменных x и y.

Начальным состоянием данных является означивание $\{x \mapsto a, y \mapsto b\}$ (для краткости обозначим его (a, b)). Очевидно, что процессы P_1 и P_2 содержат по одному переходу:

- $\langle l_1, (a, b)\rangle \underset{1}{\rightarrow} \langle l_\omega, (b, b)\rangle$;
- $\langle l_2, (a, b)\rangle \underset{2}{\rightarrow} \langle l_\omega, (a, a)\rangle$.

В соответствии с определенной выше семантикой чередования, отношение переходов программы P состоит из следующих переходов:

- $\langle (l_1, l_2), (a, b)\rangle \rightarrow \langle (l_\omega, l_2), (b, b)\rangle$;
- $\langle (l_1, l_2), (a, b)\rangle \rightarrow \langle (l_1, l_\omega), (a, a)\rangle$;
- $\langle (l_\omega, l_2), (b, b)\rangle \rightarrow \langle (l_\omega, l_\omega), (b, b)\rangle$;
- $\langle (l_1, l_\omega), (a, a)\rangle \rightarrow \langle (l_\omega, l_\omega), (a, a)\rangle$.

Таким образом, для начального состояния (a, b) у программы P есть два возможных вычисления: результатом одного из них является состояние (b, b); другого — (a, a). Ни то, ни другое не удовлетворяет постусловию (при условии, конечно, что $a \neq b$)[58].

□

Ошибка в примере может быть объяснена несоответствием замысла программиста и используемой модели вычислений. Существуют модели, в которых приведенная программа корректна. Пусть, например, исполнение программы устроено следующим образом:

- на каждом шаге вычисляется множество активных процессов;
- операторы активных процессов исполняются согласованно, используя следующую двухфазную дисциплину:
    1. считываются значения входных переменных и производятся вычисления;
    2. результаты вычислений записываются в выходные переменные.

[58] Для того чтобы программа была некорректной относительно предусловия φ и постусловия ψ, достаточно существования хотя бы одного вычисления $\pi = \{s_i\}_{i=0}^{n-1}$, такого что $s_0 \vDash \varphi$ и $s_{n-1} \nvDash \psi$.

Такая модель вычислений, известная как *синхронный параллелизм*, широко используется при проектировании цифровой аппаратуры [Бар79]. В этой модели параллельные присваивания в одну переменную либо запрещаются, либо совершается только одно из них, выбранное недетерминированным образом.

9.1.3. Реагирующие системы

Далее мы будем рассматривать исключительно *асинхронные* параллельные программы. Более того, мы будем рассматривать программы, работающие в «бесконечном цикле». Речь идет о так называемых *реагирующих*, или *реактивных*, *системах* (от англ. *reactive*). Такие системы реагируют на события окружения, выполняя в ответ те или иные действия. Это обширный класс программ, включающий операционные системы, драйверы устройств, телекоммуникационные среды, системы управления и т.п. [Кар10].

На данном этапе под *событием* понимается условие на значения разделяемых переменных (*охранное условие*, или *защита*), а под *действием* — часть программы, срабатывающая, когда условие становится истинным.

Пример 9.2

Рассмотрим *синхронный канал* передачи сообщений, также известный как *рандеву*. Канал передает сообщение от процесса-источника (src) в процесс-приемник (dst). Передача осуществляется только при одновременной готовности обеих сторон. Сообщив о готовности ни источник, ни приемник не вправе изменить свое решение. Процессы используют следующие разделяемые переменные:

- $srcRdy$ — готовность источника к отправке сообщения;
- $srcMsg$ — сообщение, передаваемое источником;
- $dstRdy$ — готовность приемника к приему сообщения;
- $dstMsg$ — сообщение, принимаемое приемником.

Реализация канала приведена ниже (процессы, использующие канал, не рассматриваются).

```
while true do
  if srcRdy = 1 ∧ dstRdy = 1   /* проверка готовности обеих сторон */
  then                         /* в случае готовности: */
    dstMsg := srcMsg;          /* осуществляется передача сообщения */
    dstRdy := 0;               /* канал уведомляет приемник и источник
    srcRdy := 0                   о завершении передачи сообщения */
```

```
    else                          /* в случае неготовности: */
      skip                        /* канал бездействует */
    end
  end
```

В примере можно выделить событие готовности к передаче сообщения

$$(srcRdy = 1) \wedge (dstRdy = 1)$$

и действие по передаче сообщения

$$dstMsg \coloneqq srcMsg; dstRdy \coloneqq 0; srcRdy \coloneqq 0.$$

□

При анализе свойств реагирующих систем предполагают, что планировщик является *справедливым*, т.е. каждый процесс периодически, время от времени, выбирается для исполнения; другими словами, невозможна ситуация, когда какой-нибудь процесс не выбирается бесконечно долго.

9.2. Формальная спецификация реактивных систем

Поведение реагирующих систем описывается в терминах последовательностей *событий* (*стимулов* и *реакций*), распределенных во *времени*. Для спецификации требований к реагирующим системам часто используются так называемые *темпоральные логики* — формальные языки, позволяющие задать взаимосвязи событий во времени: причинно-следственные связи, ограничения на относительный порядок, величины задержек между событиями и т.п.

Пример 9.3

Приведем примеры *темпоральных свойств*:

- система работает *до тех пор*, *пока* не будет остановлена;
- если запрос был принят, он *когда-нибудь* будет обработан;
- заявка с бóльшим приоритетом обрабатывается *перед* заявкой с меньшим приоритетом;
- два процесса *одновременно* не могут обратиться к общему ресурсу;
- программа *никогда* не зависает;
- данные *всегда* находятся в целостном состоянии.

□

9.2.1. Темпоральная логика LTL

Среди темпоральных логик особое распространение в информатике получили две логики: *темпоральная логика линейного времени* (*LTL*, *Linear-time Temporal Logic*) [Pnu77] и *темпоральная логика ветвящегося времени* (*CTL*, *Computation Tree Logic*) [Cla86]. В нашем курсе мы будем использовать логику LTL, предложенную в 1977 г. Амиром Пнуэли (Amir Pnueli, 1941-2009) [Pnu].

В LTL к синтаксису классической логики высказываний добавлены два *темпоральных оператора*: унарный оператор $\mathbf{X}$ (от англ. *neXt time* — в следующий момент времени)[59] и бинарный оператор $\mathbf{U}$ (от англ. *Until* — до тех пор, пока не). Эти два оператора образуют *темпоральный базис* LTL.

Пусть задано множество элементарных высказываний AP (*Atomic Propositions*). Формула логики LTL задается следующей грамматикой (при необходимости можно использовать скобки) [Кар10]:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi \ \mathbf{U} \ \varphi.$$

Здесь p — это произвольное элементарное высказывание из множества AP. Для удобства в формулах LTL можно использовать производные логические связки, например, $\wedge$ и $\rightarrow$, логические константы, $true$ и $false$, и производные темпоральные операторы, включая $\mathbf{F}$ (от англ. *in the Future* — когда-нибудь в будущем) и $\mathbf{G}$ (от англ. *Globally* — глобально, всегда). На содержательном уровне темпоральные операторы интерпретируются так:

- $\mathbf{X}\varphi$ — формула φ истинна в следующий момент времени;
- $\varphi \ \mathbf{U} \ \psi$ — формула ψ истинна *сейчас* или *обязательно* станет истинной *в будущем*, но до этого момента (не включительно) должна быть истинна формула φ;
- $\mathbf{F}\varphi \equiv true \ \mathbf{U} \ \varphi$ — формула φ истинна сейчас или станет истинной когда-нибудь в будущем;

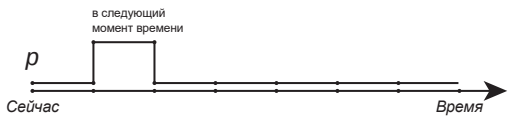
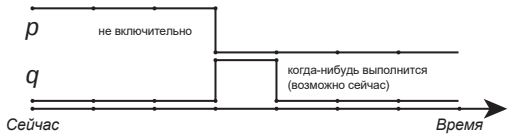
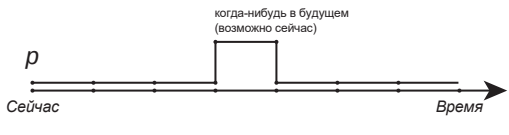
[59] Обычно рассматриваются темпоральные логики будущего времени, хотя имеются разновидности и для прошлого времени.

- $\mathbf{G}\varphi \equiv \neg\mathbf{F}\neg\varphi$ — формула $\neg\varphi$ ложна сейчас и никогда не станет истинной *в будущем* (*всегда*, начиная с настоящего момента времени, истинна формула φ).

Обратите внимание, что для истинности формулы $\mathbf{F}\varphi$ достаточно истинности φ в настоящий момент времени. Чтобы исключить из рассмотрения настоящее время, следует использовать формулу $\mathbf{XF}\varphi$.

Семантика темпоральных операторов LTL иллюстрируется в таблице 9.1 с помощью временны́х диаграмм.

Таблица 9.1. Временны́е диаграммы темпоральных операторов

Формула LTL	Временна́я диаграмма
$\mathbf{X}p$ — в следующий момент времени	
$p\ \mathbf{U}\ q$ — до тех пор, пока не	
$\mathbf{F}p$ — когда-нибудь в будущем	
$\mathbf{G}p$ — всегда	

Если нужно выразить свойство «формула φ должна быть истинной, пока не истинна формула ψ», не требуя чтобы когда-нибудь в будущем стала истинной формула ψ, это можно сделать с помощью формулы

$$\varphi\ \mathbf{W}\ \psi \equiv (\varphi\ \mathbf{U}\ \psi) \vee \mathbf{G}\varphi.$$

Темпоральный оператор $\mathbf{W}$ называется *слабым* $\mathbf{U}$ (от англ. *Weak until*).

Другим полезным оператором является $\mathbf{R}$ (от англ. *Release* — освобождение, разблокировка) — оператор двойственный $\mathbf{U}$:

$$\varphi\ \mathbf{R}\ \psi \equiv \neg(\neg\varphi\ \mathbf{U}\ \neg\psi).$$

Его семантика описывается так: либо формула ψ всегда истинна, либо до того, как она станет ложной, станет истинной φ. Или так: формула ψ истинна до того момента (включительно), когда φ станет истинной первый раз; если такой момент никогда не наступит, формула ψ истинна всегда (говорят: φ *освобождает* ψ).

Пример 9.4

Рассмотрим две формулы LTL, которые нам понадобятся в будущем:

- $\mathbf{GF}\varphi$ — формула φ будет истинной бесконечное число раз (свойство *живости*);
- $\mathbf{FG}\varphi$ — когда-нибудь формула φ станет истинной и останется такой навсегда (свойство *стабилизации*).

□

Пример 9.5

Увлекательным занятием является формализация высказываний на естественном языке в той или иной логике, например, LTL. В [Кар10] приведено множество занятных примеров.

- «Мы будем бороться, пока не победим»:
  - $(\text{мы боремся})\ \mathbf{U}\ (\text{мы победили})$ — есть уверенность в будущей победе, но для этого нужно побороться;
  - $(\text{мы боремся})\ \mathbf{W}\ (\text{мы победили})$ — победа не гарантирована, но бороться нужно до последнего.
- «Сегодня он играет джаз, а завтра Родину продаст» [В.Е. Бахнов].
  - $(\text{он играет джаз}) \to \mathbf{X}(\text{он продает Родину})$ — если он *сегодня* (прямо сейчас) играет джаз, то завтра он продаст Родину;
  - $\mathbf{G}\{(\text{он играет джаз}) \to \mathbf{X}(\text{он продает Родину})\}$ — если когда-нибудь он начнет играть джаз, то на следующий день после этого он продаст Родину;
  - $\mathbf{G}\{(\text{он играет джаз}) \to \mathbf{XF}(\text{он продает Родину})\}$ — если когда-нибудь он начнет играть джаз, то после этого (рано или поздно, но не во время игры) он продаст Родину.

□

9.2.2. Структуры Крипке и семантика логики LTL

Семантика темпоральных логик может быть определена с помощью *структур Крипке*, названных в честь Сола Аарона Крипке (Saul Aaron Kripke, род. в 1940 г.), введшего их в логико-философский обиход в конце 1950-х гг. [Kri63]. В общих словах, структура Крипке — это система возможных миров и переходов между ними: каждый мир статичен и интерпретируется каким-нибудь традиционным образом. Для логик высказываний, в частности для темпоральной логики LTL, интерпретация может задаваться отображением, сопоставляющим каждому высказыванию $p \in AP$ значение истинности $[\![p]\!] \in \{true, false\}$.

Дадим формальное определение. *Структурой Крипке* называется четверка $\langle S, S_0, R, L\rangle$, где S — множество *состояний*, $S_0 \subseteq S$ — множество *начальных состояний*, $R \subseteq S \times S$ — отношение *переходов*, $L: S \to 2^{AP}$ — *функция разметки*, помечающая каждое состояние структуры множеством истинных в нем элементарных высказываний (с помощью функции разметки задается интерпретация состояний).

Состояния структуры Крипке — это мгновенные «снимки» некоторого модельного мира: «тик часов» происходит при выполнении перехода между состояниями. Естественным ограничением, налагаемым на отношение переходов, является возможность бесконечного *прогресса времени*: для каждого состояния s должно существовать состояние s' такое, что $(s, s') \in R$. Отношение переходов, удовлетворяющее этому ограничению, называется *полным*. В дальнейшем мы будем рассматривать только структуры Крипке, имеющие полные отношения переходов.

Вычислением структуры Крипке называется бесконечная последовательность состояний $\pi = \{s_i\}_{i=0}^{\infty}$, такая что $s_0 \in S_0$ и $(s_i, s_{i+1}) \in R$ для всех $i \geq 0$. *Траекторией* структуры Крипке, индуцированной вычислением π, называется последовательность $L(\pi) = \{L(s_i)\}_{i=0}^{\infty}$, т.е. последовательность, состоящая из пометок состояний, входящих в π. Пусть $t \geq 0$ — некоторый момент времени (целое число). Через $\pi^{(t)}$ обозначим суффикс π, состоящий из элементов π, начиная с номера t включительно: $\pi^{(t)} = \{s_i\}_{i=t}^{\infty}$.

Формулы LTL интерпретируются на вычислениях структур Крипке. Истинность формулы φ на вычислении $\pi = \{s_i\}_{i=0}^{\infty}$ (обозначается: $\pi \vDash \varphi$) определяется индуктивно по структуре формулы, как показано в таблице 9.2.

Таблица 9.2. Семантика логики LTL

(1)	$\pi \vDash p$, где $p \in AP$	$s_0 \vDash p$, т.е. если $p \in L(s_0)$
(2)	$\pi \vDash \neg\varphi$	$\pi \nvDash \varphi$
(3)	$\pi \vDash \varphi \vee \psi$	$\pi \vDash \varphi$ или $\pi \vDash \psi$
(4)	$\pi \vDash \mathrm{X}\varphi$	$\pi^{(1)} \vDash \varphi$
(5)	$\pi \vDash \varphi\ \mathbf{U}\ \psi$	существует такое $T \geq 0$, что $\pi^{(T)} \vDash \psi$ и для всех $0 \leq t < T$ выполняется $\pi^{(t)} \vDash \varphi$

Как следствие, семантика операторов $\mathbf{F}$ и $\mathbf{G}$ определяется в соответствии с таблицей 9.3.

Таблица 9.3. Семантика темпоральных операторов $\boldsymbol{F}$ и $\boldsymbol{G}$

(6)	$\pi \vDash \mathrm{F}\varphi$	существует такое $T \geq 0$, что $\pi^{(T)} \vDash \varphi$
(7)	$\pi \vDash \mathrm{G}\varphi$	для любого $t \geq 0$ выполняется $\pi^{(t)} \vDash \varphi$

Заметим, что истинность формул LTL определяется не столько вычислениями структур Крипке, сколько индуцированными ими траекториями (важны не сами состояния, а их пометки). Приведенное выше определение можно практически без изменений переписать в терминах траекторий.

9.2.3. Аксиоматическое определение логики LTL

Логику LTL можно определить и иным, более абстрактным способом — аксиоматически. В этом случае представленная выше интерпретация LTL с помощью вычислений структур Крипке — лишь одна из возможных моделей этой логики. Аксиоматическое определение LTL задается дедуктивной системой, представленной в таблице 9.4 [Dam].

Таблица 9.4. Аксиоматическое определение логики LTL

(0)	φ	**Аксиома**: результат произвольной подстановки в произвольную общезначимую формулу логики высказываний
(1)	$\mathbf{G}\{\varphi \to \psi\} \to (\mathbf{G}\varphi \to \mathbf{G}\psi)$	**Аксиома**: дистрибутивность оператора $\mathbf{G}$ относительно импликации
(2)	$\mathbf{X}\{\varphi \to \psi\} \to (\mathbf{X}\varphi \to \mathbf{X}\psi)$	**Аксиома**: дистрибутивность оператора $\mathbf{X}$ относительно импликации
(3)	$\mathbf{G}\{\varphi \to \mathbf{X}\varphi\} \to (\varphi \to \mathbf{G}\varphi)$	**Аксиома**: индукция по времени
(4)	$\mathbf{X}\varphi \leftrightarrow \neg\mathbf{X}\neg\varphi$	**Аксиома**: самодвойственность оператора $\mathbf{X}$
(5)	$(\varphi\ \mathbf{U}\ \psi) \leftrightarrow \big(\psi \vee (\varphi \wedge \mathbf{X}\{\varphi\ \mathbf{U}\ \psi\})\big)$	**Аксиома**: развертка оператора $\mathbf{U}$ во времени
(6)	$(\varphi\ \mathbf{U}\ \psi) \to \mathbf{F}\psi$	**Аксиома**: осуществимость в будущем
(7)	$\dfrac{\varphi,\ \varphi \to \psi}{\psi}$	**Правило вывода**: классическое правило *modus ponens*[60]

[60] Заметьте, что правило *modus ponens* является частным случаем правила резолюции (см. лекцию 8).

(8)	$\frac{\varphi}{\mathbf{G}\varphi}$	**Правило вывода**: обобщение для оператора $\mathbf{G}$ (глобализация)[61]
(9)	$\frac{\varphi}{\mathbf{F}\varphi}$	**Правило вывода**: обобщение для оператора $\mathbf{F}$ (осуществимость)[62]

В работе [Ben12] вместо правила обобщения для оператора $\mathbf{F}$ используется аксиома «развертки» оператора $\mathbf{G}$ во времени (см. таблицу 9.5).

Таблица 9.5. Аксиома «развертки» темпорального оператора $\boldsymbol{G}$

(9)	$\mathbf{G}\varphi \to (\varphi \wedge \mathbf{X}\varphi \wedge \mathbf{XG}\varphi)$	**Аксиома**: развертка оператора $\mathbf{G}$ во времени

Известно, что приведенная дедуктивная система является значимой и полной [Dam], [Ben12]. Для нас она интересна как источник тождеств и правил преобразования формул.

9.3. Дедуктивная верификация параллельных программ

Параллельные программы, как и последовательные, можно верифицировать дедуктивно. Это справедливо как для спецификаций, заданных в форме пред- и постусловий, так и для спецификаций, выраженных на языке LTL. Мы не будем касаться методов верификации параллельных программ обработки данных; желающим изучить этот вопрос самостоятельно мы рекомендуем монографии [Неп88] и [Apt09]. Предмет нашего интереса — реагирующие системы, специфицированные с помощью LTL.

9.3.1. Алгоритм Петерсона взаимного исключения процессов

Для примера обратимся к алгоритму Петерсона (Gary Peterson) взаимного исключения процессов [Pet81]. Пусть программа P состоит из двух процессов, P_0 и P_1, работающих в бесконечном цикле. Периодически каждый из них входит в *критическую секцию* (для процесса

[61] Правило обобщения для оператора $\mathbf{G}$ выражает следующую мысль: если формула истинна *вообще* (т.е. без относительно времени), то она истинна *всегда*. В определенном смысле оно соответствует правилу *универсального обобщения*: $\frac{\varphi}{\forall t\varphi}$, где t — переменная (интерпретируемая здесь как время).

[62] Правило обобщения для оператора $\mathbf{F}$ выражает такую мысль: если формула истинна *вообще*, то она станет истинной *когда-нибудь* (время не остановилось, и очередной момент времени обязательно наступит). Его аналогом является правило *экзистенциального обобщения*: $\frac{\varphi[t:=t_0]}{\exists t\varphi}$, где t — переменная времени (поскольку время в формулах явно не присутствует, правило можно записать как $\frac{\varphi}{\exists t\varphi}$).

P_i критическая секция помечена меткой CRS_i). Вход процесса в критическую секцию осуществляется в два этапа: сначала он сообщает о своем намерении (метка SET_i), затем ожидает возможности войти (метка TST_i). Выход процесса из критической секции осуществляется путем сброса флага $flag_i$, установленного при входе (метка RST_i). Описание процесса P_i приведено ниже.

```
while true do
  NCSᵢ: /* некритическая секция */
  SETᵢ: flagᵢ := 1; turn := i; /* запрос */
  TSTᵢ: while (flagᵢ₋ᵢ = 1) ∧ (turn = i) do skip end; /* ожидание */
  CRSᵢ: /* критическая секция */
  RSTᵢ: flagᵢ := 0 /* выход */
end
```

Начальное состояние задается равенствами $turn = 0$ и $flag_i = 0$, где $i \in \{0, 1\}$. Докажем корректность приведенной реализации алгоритма Петерсона относительно следующей LTL-спецификации [Ben12]:

- $\mathbf{G}\{\neg(@CRS_0 \wedge @CRS_1)\}$ — свойство *безопасности* (*safety*): два процесса не могут одновременно пребывать в критической секции;
- $\mathbf{G}\{@SET_i \rightarrow \mathbf{F}@CRS_i\}$, где $i \in \{0, 1\}$, — свойство *живости* (*liveness*): если процесс запросил доступ к критической секции, то рано или поздно он его получит.

9.3.2. Уточнение понятия справедливости планировщика

Свойство живости (в рассматриваемом примере и не только) может быть доказано только в предположении *справедливости* (*fairness*) планировщика:

$$fairness \rightarrow liveness.$$

Для процессов, описанных на языке while, справедливость планировщика формализуется следующим образом: для каждого оператора программы строится формула LTL, как показано в таблице 9.6 (считаем, что в конце каждого процесса есть метка l_ω, означающая его завершение)[63], после чего составляется конъюнкция всех формул — это и есть условие справедливости.

[63] Здесь используются метки операторов, однако можно использовать конфигурации в том виде, в котором они определяются в структурной операционной семантике (см. лекцию 2).

Таблица 9.6. Формулы LTL, выражающие справедливость планировщика

(1)	l: **skip**; l': P	$\mathbf{G}\{@l \to \mathbf{F}@l'\}$
(2)	l: $x \coloneqq t$; l': P	см. (1)
(3)	l: **if** B **then** l_1: P_1 **else** l_2: P_2 **end**; l': P	$\mathbf{G}\{@l \to \mathbf{F}\{@l_1 \vee @l_2\}\}$ $\mathbf{G}\{((@l \wedge \mathbf{G}B) \to \mathbf{F}@l_1) \wedge ((@l \wedge \mathbf{G}\neg B) \to \mathbf{F}@l_2)\}$
(4)	l: **while** B **do** l_1: P_1 **end**; l_2: P_2	см. (3)

Формула $\mathbf{G}\{((@l \wedge \mathbf{G}B) \to \mathbf{F}@l_1) \wedge ((@l \wedge \mathbf{G}\neg B) \to \mathbf{F}@l_2)\}$, определенная для условного оператора (3) и оператора цикла (4), представляется «перестрахованной». Можно ли в ней заменить $\mathbf{G}\varphi$ на φ, т.е. написать

$$\mathbf{G}\{((@l \wedge B) \to \mathbf{F}@l_1) \wedge ((@l \wedge \neg B) \to \mathbf{F}@l_2)\}?$$

Предположим, что исполнение процесса находится на условном операторе с меткой l и истинно условие B. Пусть перед тем, как процесс проверил истинность условия B, планировщик выбрал другой процесс, который изменил состояние программы так, что условие B стало ложным. Когда наступит черед первого процесса, его вычисление пойдет по ветви **else**, а не **then**, как того требует формула. Можно предложить другой вариант:

$$\mathbf{G}\{@l \to \mathbf{F}\{(@l_1 \wedge B) \wedge (@l_2 \wedge \neg B)\}\}.$$

Он тоже не вполне корректен и может применяться, только если переход к операторам ветви происходит сразу после проверки условия, без прерывания исполнения другими процессами. Так что формулу, указанную в таблице, не так-то просто заменить на более точную, однако для наших целей она годиться.

9.3.3. Доказательство корректности алгоритма Петерсона

Доказательство корректности программы P, реализующей алгоритм Петерсона, относительно приведенной выше спецификации можно провести поэтапно с использованием следующих лемм [Ben12].

Леммы для доказательства свойства взаимного исключения ($i \in \{0, 1\}$)[64]:

[64] Запись $P \vDash \varphi$ означает истинность свойства φ на программе P (в инженерии — удовлетворение программой P требования φ).

1. $P \vDash \mathbf{G}\{(flag_i = 1) \leftrightarrow @(L_i \setminus \{NCS_i, SET_i\})\}$, где L_i — множество меток[65];
2. $P \vDash \mathbf{G}\big\{(@TST_i \wedge @CRS_{1-i}) \rightarrow \big((flag_{1-i} = 1) \wedge (turn = i)\big)\big\}$.

Леммы для доказательства свойства живости, где $fairness$ — справедливость планировщика ($i \in \{0, 1\}$):

3. $P \vDash fairness \rightarrow \big((@TST_i \wedge \mathbf{G}\neg @CRS_i) \rightarrow \mathbf{GF}\{(flag_{1-i} = 1) \wedge (turn = i)\}\big)$;
4. $P \vDash fairness \rightarrow (\mathbf{FG}\{flag_{1-i} \neq 1\} \vee \mathbf{F}\{turn = 1 - i\})$;
5. $P \vDash fairness \rightarrow \big((@TST_i \wedge \mathbf{G}\neg @CRS_i \wedge \mathbf{F}\{turn = 1 - i\}) \rightarrow \mathbf{FG}\{turn = 1 - i\}\big)$.

Пусть i — произвольный идентификатор (0 или 1). Докажем лемму 1. В начальной конфигурации импликация $(flag_i = 1) \rightarrow @(L_i \setminus \{NCS_i, SET_i\})$ истинна, так как $flag_i = 0$. Покажем, что если импликация истинна в некоторый момент времени, она останется истинной в будущем. Истинность импликации, очевидно, может нарушиться только в двух случаях:

1. когда посылка и следствие истинны, и следствие становится ложным;
2. когда посылка и следствие ложны, и посылка становится истинной.

Случай 1: условие $@(L_i \setminus \{NCS_i, SET_i\})$ перестает быть истинным только при переходе от RST_i к NCS_i, однако в этом случае обнуляется $flag_i$, что делает ложной посылку. Случай 2: установка $flag_i$ осуществляется только при переходе от SET_i к TST_i (точнее, к метке следующего оператора присваивания), однако в этом случае становится истинным условие $@TST_i$, что делает истинным следствие.

Доказательство оставшейся части леммы 1 и леммы 2 мы оставляем читателю в качестве упражнения.

Докажем, что имеет место взаимное исключение процессов: $P \vDash \mathbf{G}\{\neg(@CRS_0 \wedge @CRS_1)\}$. Очевидно, что свойство $\neg(@CRS_0 \wedge @CRS_1)$ истинно в начальной конфигурации. Его истин-

---

[65] Здесь следует сделать два замечания:

1. $L_i \neq \{NCS_i, SET_i, TST_i, CRS_i, RST_i\}$ — по крайней мере, есть еще одна метка между присваиваниями $flag_i := 1$ и $turn := i$;
2. для простоты некритическая и критическая секции представлены одиночными метками: запись $@NCS_i$ ($@CRS_i$) можно интерпретировать как «процесс P_i находится в одной из точек некритической (критической) секции».

ность может нарушиться, только когда один из процессов уже находится в критической секции, а второй входит в нее: $@CRS_{1-i} \wedge @TST_i$ и исполняется переход $@TST_i \to @CRS_i$ при сохранении $@CRS_{1-i}$. Переход возможен, только если истинно условие $(flag_{1-i} = 0) \vee (turn \neq i)$. В то же время из леммы 2 следует, что $(flag_{1-i} = 1) \wedge (turn = i)$. Таким образом, переход $@TST_i \to @CRS_i$ невозможен, а значит, свойство $@CRS_0 \wedge @CRS_1$ не может стать истинным ни при каких обстоятельствах.

Докажем лемму 3. Рассмотрим оператор, расположенный по метке TST_i:

$$TST_i\text{: } \textbf{while } (flag_{1-i} = 1) \wedge (turn = i) \textbf{ do skip end}.$$

Из определения справедливости следует:

$$P \vDash fairness \to \Big(\big(@TST_i \wedge \mathbf{G}\{\neg\big((flag_{1-i} = 1) \wedge (turn = i)\big)\}\big) \to \mathbf{F}@CRS_i\Big).$$

Используя тождества $\big((\varphi \wedge \psi) \to \chi\big) \equiv \big((\varphi \wedge \neg\chi) \to \neg\psi\big)$ и $\mathbf{G}\varphi \equiv \neg\mathbf{F}\neg\varphi$, получаем:

$$P \vDash fairness \to \big((@TST_i \wedge \mathbf{G}\neg@CRS_i) \to \mathbf{F}\{(flag_{1-i} = 1) \wedge (turn = i)\}\big).$$

Применяя правило обобщения и аксиому дистрибутивности $\mathbf{G}$ относительно импликации, получим:

$$P \vDash fairness \to (\mathbf{G}\{@TST_i \wedge \mathbf{G}\neg@CRS_i\} \to \mathbf{GF}\{(flag_{1-i} = 1) \wedge (turn = i)\}).$$

Нетрудно убедиться в том, что $P \vDash (@TST_i \wedge \mathbf{G}\neg@CRS_i) \to \mathbf{G}\{@TST_i \wedge \mathbf{G}\neg@CRS_i\}$, что доказывает лемму.

Доказательство лемм 4 и 5 мы оставляем читателю в качестве упражнения.

Докажем живость процесса P_i. Планировщик, будучи справедливым, гарантирует, что процесс, находящийся в точке SET_i, рано или поздно достигнет точки TST_i. Для доказательства живости достаточно показать, что:

$$P \vDash fairness \to \mathbf{G}\{@TST_i \to \mathbf{F}@CRS_i\}.$$

Предположим противное:

$$P \vDash fairness \wedge \neg\mathbf{G}\{@TST_i \to \mathbf{F}@CRS_i\}.$$

Преобразуем это утверждение, используя известные тождества:

$$P \vDash fairness \wedge \mathbf{F}\{@TST_i \wedge \mathbf{G}\neg @CRS_i\}.$$

Рассмотрим момент времени, когда истинно условие $@TST_i \wedge \mathbf{G}\neg @CRS_i$ (этот момент обязан наступить). Из леммы 3 следует, что в этот момент времени истинна формула $\Phi \equiv \mathbf{GF}\{(flag_{1-i} = 1) \wedge (turn = i)\}$. Если при этом $\mathbf{F}\{turn = 1 - i\}$, из леммы 5 вытекает $\mathbf{FG}\{turn = 1 - i\}$, что противоречит Φ. В противном случае из леммы 4 вытекает $\mathbf{FG}\{flag_{1-i} \neq 1\}$, что также противоречит Φ. Единственная возможность состоит в том, что исходное предположение неверно, а верно, что $P \vDash fairness \rightarrow \mathbf{G}\{@TST_i \rightarrow \mathbf{F}@CRS_i\}$.

Приведенный фрагмент доказательства не вполне формален, однако неплохо иллюстрируют громоздкость дедуктивной верификации параллельных программ. На следующих лекциях мы познакомимся с более эффективным методом — методом проверки моделей.

9.4. Вопросы и упражнения

1. Дайте определение параллельной программы.
2. В чем, на ваш взгляд, состоит основная причина трудности верификации параллельных программ?
3. В чем разница между синхронным и асинхронным параллелизмом?
4. Расширим язык while оператором **parallel** $P \parallel \cdots \parallel P$ **end**, суть которого состоит в порождении нескольких параллельных процессов ($fork$) и ожидании их завершения ($join$). Определите операционную семантику полученного языка.
5. Дополнительно (см. предыдущее задание) расширим язык while оператором **atomic** P **end** (ограничение: P — программа, не содержащая операторов **parallel**), ограничивающим переключения процессов: если процесс находится во внутренней точке **atomic**-блока, он не может быть прерван другим процессом. Определите операционную семантику полученного языка.
6. Опишите формально в логике LTL приведенные в лекции примеры временны́х взаимосвязей.
7. В асинхронных параллельных программах обычно не используется темпоральный оператор **X**. Чем это можно объяснить?

8. Обозначим через p высказывание «Саша любит Машу», а через q — «Саша любит Дашу». Что на содержательном уровне означают следующие формулы LTL [Кар10]:
   a. $p \wedge q$;
   b. $\mathbf{G}\{(p \vee q) \wedge \neg(p \wedge q)\}$;
   c. $\mathbf{GF}p$;
   d. $\mathbf{FG}q$?
9. Формализуйте утверждения в логике LTL следующие высказывания (говорящая не замужем и никогда замужем не была) [Кар10]:
   a. «Я никогда не выйду замуж»;
   b. «Я выйду замуж ровно один раз»;
   c. «Я выйду замуж не менее двух раз»;
   d. «Я выйду замуж не более двух раз»;
   e. «Я выйду замуж ровно два раза».
10. Представьте в виде формулы LTL следующие высказывания [Кар10]:
   a. «Канал может терять сообщение только конечное число раз»;
   b. «Между каждой парой состояний, удовлетворяющих свойству φ, обязательно встретится состояние, удовлетворяющее свойству ψ».
11. Приведите формулу $\neg(\varphi \mathbf{U} \psi)$ к *отрицательной нормальной форме*, т.е. к форме в которой отрицания стоят непосредственно перед элементарными высказываниями [Кар10].
12. Определите семантику темпоральных операторов $\mathbf{W}$ и $\mathbf{R}$ на вычислении структуры Крипке. Нарисуйте соответствующие временны́е диаграммы.
13. Докажите следующие эквивалентности:
   a. $\varphi \mathbf{U} \psi \equiv \mathbf{F}\psi \wedge (\varphi \mathbf{W} \psi)$;
   b. $\varphi \mathbf{R} \psi \equiv \varphi \mathbf{W} (\varphi \wedge \psi)$;
   c. $\varphi \mathbf{W} \psi \equiv \psi \mathbf{R} (\varphi \vee \psi)$.
14. Докажите или опровергните [Кар10]:
   a. $(\mathbf{F}\varphi) \mathbf{U} (\mathbf{F}\psi) \equiv \mathbf{F}(\varphi \mathbf{U} \psi)$;

b. $\mathbf{G}\{\varphi \to \mathbf{X}\varphi\} \equiv \mathbf{G}\neg\varphi \vee \mathbf{FG}\varphi$;

c. $\varphi\ \mathbf{U}\ \psi \equiv (\varphi \vee \psi)\ \mathbf{U}\ \psi$.

15. Обоснуйте на содержательном уровне значимость следующих правил вывода LTL:

a. $\frac{\varphi \to \psi}{\mathbf{G}\varphi \to \mathbf{G}\psi}$;

b. $\frac{\varphi \to \psi}{\mathbf{X}\varphi \to \mathbf{X}\psi}$;

c. $\frac{\varphi \to \mathbf{X}\varphi}{\varphi \to \mathbf{G}\varphi}$.

16. Докажите формально следующие утверждения LTL:

a. $\mathbf{XG}\{\varphi \to \psi\} \to (\mathbf{XG}\varphi \to \mathbf{XG}\psi)$;

b. $\varphi \to (\varphi\ \mathbf{U}\ \varphi)$;

c. $\mathbf{FG}\varphi \to \mathbf{GF}\varphi$.

17. Завершите доказательство корректности алгоритма Петерсона взаимного исключения процессов.

Лекция 10. Инструменты проверки моделей

Интенсивные исследования в области model checking привели к тому, что разработанная к настоящему времени техника применения этого подхода для верификации намного проще, чем его теоретический базис.

Ю.Г. Карпов.
Model Checking. Верификация параллельных и распределенных программных систем

Рассматриваются базовые возможности языка PROMELA и основанного на нем инструмента SPIN. Язык PROMELA (PROcess MEta-LAnguage) предназначен для моделирования асинхронных параллельных систем. Инструмент SPIN (Simple Promela INterpreter) позволяет анализировать PROMELA-модели, в том числе устанавливать соответствие моделей LTL-спецификациям, т.е. проверять, является ли модель (программа на языке PROMELA) моделью (в логическом понимании) заданной формулы LTL. Описанные средства могут быть использованы на практике при проектировании параллельных программ и реагирующих систем.

10.1. Введение в язык PROMELA

PROMELA (Process Meta-Language) — это язык моделирования асинхронных параллельных систем, ориентированный на задачи формальной верификации [Hol03]. Язык PROMELA вместе с инструментом SPIN (Simple PROMELA Interpreter) был разработан Джерардом Хольцманом (Gerard J. Holzmann) в Исследовательском центре компьютерных наук (Computer Sciences Research Center) при лабораториях Bell (Bell Laboratories) в 1980-х гг. Язык не предназначен для реализации программ — это именно язык *моделирования*, причем упор в нем сделан на средствах синхронизации и координации процессов, а не на вычислениях.

Модели на языке PROMELA состоят из *процессов*, работающих параллельно и взаимодействующих через общие *переменные* или посредством передачи *сообщений* через *каналы*. Процессы могут создаваться как *статически*, в момент создания модели, так и *динамически*, при исполнении процессов. Метод проверки моделей предполагает, что общее число созданных в модели процессов конечно (подробнее об этом мы поговорим в следующих лекциях). В языке допускаются только переменные с конечными доменами: целочисленные переменные, одномерные массивы фиксированного размера и структуры.

10.1.1. Типы данных, переменные и выражения

Типы данных языка PROMELA делятся на базовые и составные. Первые включают в себя *целочисленные типы*, а также специализированные типы: **chan**, **mtype** и **pid**; вторые — *массивы* и *структуры*. Целочисленные типы данных приведены в таблице 10.1.

Таблица 10.1. Целочисленные типы данных языка PROMELA

Тип данных	Множество значений	Битовая длина
bit	$\{0, 1\}$	1
bool	$\{\textbf{false}, \textbf{true}\}$	1
byte	$[0, 255]$	8
short	$[-2^{15}, 2^{15} - 1]$	16
int	$[-2^{31}, 2^{31} - 1]$	32
unsigned: n	$[0, 2^n - 1]$	$n \leq 32$

Тип **bool** является синонимом типа **bit**, при этом **false** — это 0, а **true** — это 1. Типы **bit**, **byte** и **unsigned** являются беззнаковыми; типы **short** и **int** — знаковыми. К базовым типам также относятся следующие типы:

- **chan** — идентификатор канала (множество значений: $[1, 255]$);
- **mtype** — идентификатор типа сообщения ($[1, 255]$);
- **pid** — идентификатор процесса ($[0, 255]$).

Декларация переменных в PROMELA осуществляется так же, как в C:

```
T x1 [ = t1], ..., xn [ = tn];
```

Здесь T — тип данных, x_i — имя переменной, t_i — необязательное выражение, называемое *инициализатором переменной*. По умолчанию переменные заполняется нулями.

Переменные делятся на *глобальные* (*разделяемые*), декларируемые вне процессов, и *локальные*, декларируемые и используемые внутри процессов.

Пример 10.1

Ниже приведены примеры деклараций переменных [Hol03]. Обратите внимание: комментарии в PROMELA пишутся как в C.

```
bit x, y;           /* две 1-битные переменные (значение 0) */
bool turn = true;   /* булева переменная (значение true) */
unsigned v:5;/* беззнаковая 5-битная переменная (значение 0) */
unsigned w:3 = 5;   /* беззнаковая 3-битная переменная (значение 5) */
```

□

Декларация массивов осуществляется следующим образом:

```
T x[N] [ = t];
```

Здесь T — тип данных элементов массива, x — имя массива, N — положительное целое число, задающее длину массива (в PROMELA поддерживаются только одномерные массивы), t — необязательное выражение, используемое для инициализации элементов массива (все элементы получают одинаковое начальное значение).

Доступ к элементам массива осуществляется так же, как в C: индекс указывается в квадратных скобках; индексация начинается с 0. Указателей и адресной арифметики нет.

Пример 10.2

Приведем еще примеры деклараций переменных, в том числе массивов.

```
byte a[12];              /* a[0] = ... = a[11] = 0 */
short b[4] = 89;         /* b[0] = ... = b[3] = 89 */
int c, d[3] = 1, e = 4;  /* c = 0, d[0] = ... = d[2] = 1, e = 4 */
```

□

Тип **mtype** является аналогом перечислений (**enum**) языка C. Название происходит от «message type»: символы, составляющие **mtype**, обычно используются для обозначения типов сообщений. Символы констант вводятся, как показано ниже:

```
mtype = {c1, ..., cn};
```

Здесь c_i — разные идентификаторы, отличные от ключевых слов. В модели может быть несколько объявлений **mtype**, что эквивалентно одному объявлению с объединенным множеством символов.

Пример 10.3

Ниже вводятся символьные обозначения для типов сообщений.

```
/* типы сообщений */
mtype = { req, ack, nack, err };
```

□

Декларация структурного типа осуществляется следующим образом:

```
typedef S { T1 f1; ... ; Tn fn };
```

Здесь S — вводимое имя типа, T_i — тип i-ого поля, f_i — имя i-ого поля. Типы полей могут быть как базовыми, так и структурными (в частности, допускаются поля-массивы[66]).

Доступ к полям структуры осуществляется традиционно: $x.f$, где x — имя переменной структурного типа, а f — имя поля.

Пример 10.4

Определим структурный тип для представления пакетов данных, объявим переменную этого типа и совершим над ней некоторые действия.

```
/* определение структурного типа Packet */
typedef Packet {
  mtype type;      /* тип сообщения */
  short source;    /* адрес источника */
  short target;    /* адрес приемника */
  byte  data[4]    /* передаваемые данные */
};

/* декларация переменной p структурного типа Packet */
Packet p;

/* доступ к полям переменной p */
p.type    = req;
p.target  = 1;
p.data[0] = 2;
```

□

Выражения PROMELA в целом похожи на выражения C. Одно из немногих отличий заключается в форме записи *условного выражения*: $(B -> t_1 : t_2)$: вместо символа ?, как это принято в C, используется $->$; кроме того, условное выражение должно окружаться скобками. Другое отличие состоит в том, что выражения PROMELA не должны иметь побочных эффектов (присваивания не являются выражениями).

Поддерживаемые операции (в порядке убывания приоритетов) и их ассоциативность (порядок вычислений) приведены в таблице 10.2 [Hol03].

[66] Эту особенность можно использовать для определения многомерных массивов.

Таблица 10.2. Операции языка PROMELA, их приоритет и ассоциативность

Операции	Ассоциативность	Комментарий
`() [] .`	слева направо	Скобки, индексирование массива, доступ к полю структуры
`! ~ ++ --`	справа налево	Отрицание, побитовое отрицание, инкремент, декремент
`* / %`	слева направо	Умножение, деление, остаток от деления
`+ -`	слева направо	Сложение, вычитание
`<< >>`	слева направо	Сдвиг влево, сдвиг вправо
`< <= > >=`	слева направо	Сравнение на меньше / больше
`== !=`	слева направо	Сравнение на равенство / неравенство
`&`	слева направо	Побитовое И
`^`	слева направо	Побитовое исключающее ИЛИ
`|`	слева направо	Побитовое ИЛИ
`&&`	слева направо	Конъюнкция
`||`	слева направо	Дизъюнкция
`-> :`	справа налево	Условная операция
`=`	справа налево	Присваивание

Присваивание $x = t$ исполняется следующим образом [Hol03]:

1. значения всех переменных и констант выражения t приводятся к типу **int**;
2. совершаются операции согласно приоритетам и ассоциативности;
3. вычисленное значение приводится к типу переменной x и ей присваивается.

10.1.2. Процессные типы и процессы

Процессный тип определяется с помощью ключевого слова **proctype**, после которого следуют его имя, параметры и тело. Локальные переменные, если они нужны, должны определяться в начале тела[67]:

```
proctype P(T1 p1, ..., Tn pn) { S1 x1; ..., Sm xm; ... }
```

Процессный тип описывает поведение процессов, но не порождает их (если не указан модификатор **active**).

Точкой входа в PROMELA-модель является *процесс инициализации* **init**:

```
init { ... }
```

В нем, как и в других процессах, можно порождать процессы, используя конструкцию

```
run P(expr1, ..., exprn).
```

[67] Синтаксис PROMELA допускает декларацию переменных в любом месте, однако неявно все декларации переносятся в начало.

Здесь P — имя процессного типа, $expr_i$ — выражения, задающие значения параметров процесса (их число должно совпадать с числом параметров процессного типа P).

Процесс завершается при достижении конца тела соответствующего процессного типа, но не раньше, чем завершатся все порожденные им процессы.

Еще одним способом запуска процессов является указание модификатора **active** перед **proctype**. Если модификатор указан, процесс этого типа будет создан в начале работы модели[68]. Чтобы запустить несколько процессов, можно использовать запись **active** $[N]$, где N — нужное число процессов.

У каждого процесса имеется предопределенная локальная переменная _pid типа **pid**, хранящая его идентификатор.

Пример 10.5

Определим простой процессный тип: процесс печатает строку «hello world: n», где n — идентификатор процесса.

```
active [10] proctype main() {
  /* функция печати аналогична функции printf в C */
  printf("hello world: %d\n", _pid)
}
```

В определении указан модификатор **active** $[10]$, что говорит о запуске 10 процессов.

□

10.1.3. Средства межпроцессной коммуникации

Основное средство межпроцессной коммуникации в PROMELA — *каналы передачи сообщений*. Канал может буферизовать не более чем заданное число сообщений. Процесс, записывающий сообщение в полностью заполненный канал, блокируется до тех пор, пока из канала не будет произведено чтение другим процессом[69]. Аналогично процесс, читающий из пустого канала, блокируется до тех пор, пока в канал не будет записано сообщение.

---

[68] Если у процессного типа есть параметры, при создании процесса им присваиваются нулевые значения.

[69] Возможна и другая семантика операции посылки сообщений: если буфер канала заполнен, процесс не блокируется, но передаваемое сообщение теряется. В инструменте SPIN способ посылки сообщений задается опцией `-m`. По умолчанию включен режим блокировки процессов [Prom].

Синтаксически каналы являются переменными типа **chan**. Декларация начинается с ключевого слова **chan**, после которого указывается имя канала или несколько имен через запятую. Инициализатор канала имеет вид $[N]$ **of** $\{T_1, \ldots, T_n\}$, где N — емкость канала (размер буфера), а $T_1 \times \ldots \times T_n$ — тип передаваемых сообщений[70]. Если канал не инициализирован, он не может быть использован для передачи сообщений. Разрешается объявлять массивы каналов.

Пример 10.6

Рассмотрим примеры деклараций каналов и массивов каналов.

```
/* три неинициализированных канала */
chan a, b, c;

/* массив из 10 неинициализированных каналов */
chan d[10];

/* канал емкости 0 (синхронный канал или рандеву),
   по которому передаются сообщения типа byte */
chan e = [0] of {byte};

/* канал емкости 16, по которому передаются каналы[71] */
chan f = [16] of {chan};

/* канал емкости 1024, по которому передаются сообщения,
   включающие два компонента: типа mtype и типа Packet */
chan g = [1024] of {mtype, Packet};
```

□

Проверить состояние канала можно с помощью встроенных функций **len**, **empty**, **nempty**, **full** и **nfull**:

- $\mathbf{len}(c)$ — число сообщений в буфере канала c;
- $\mathbf{empty}(c) \equiv \mathbf{len}(c) == 0$;
- $\mathbf{nempty}(c) \equiv \mathbf{len}(c) > 0$;
- $\mathbf{full}(c) \equiv \mathbf{len}(c) == N$, где N — емкость канала c;
- $\mathbf{nfull}(c) \equiv \mathbf{len}(c) < N$, где N — емкость канала c.

---

[70] Если $n > 1$, сообщения имеют вид $t_1, \ldots, t_n$, где t_i — выражение типа T_i, $i \in \{1, \ldots, n\}$.

[71] Здесь уместнее говорить об идентификаторах каналов или о ссылках на каналы.

Запись в канал осуществляется с помощью конструкции $c\,!\,m$, где c — канал, а m — выражение, задающее передаваемое сообщение[72]: сообщение m добавляется в конец буфера канала c.

Чтение из канала осуществляется с помощью одной из конструкций: $c\,?\,p$ или $c\,?<p>$, где c — канал, а p — образец (паттерн) читаемого сообщения:

- $c\,?\,p$ — *прием с удалением сообщения из канала*:

  чтение может быть исполнено, если первое сообщение буфера канала c соответствует образцу p (что это значит, рассматривается ниже); данные сообщения переносятся в переменные p, а сообщение удаляется из буфера;

- $c\,?<p>$ — *прием с сохранением сообщения в канале*:

  чтение может быть исполнено, если первое сообщение буфера канала c соответствует образцу p; данные сообщения переносятся в переменные p; сообщение сохраняется в буфере.

Образец — это кортеж, элементами которого могут быть константы, переменные и специальный символ _. Сообщение *соответствует* образцу, если константные поля образца равны значениям соответствующих полей сообщения. Если в образец входят переменные, то при чтении сообщения из канала в них записываются значения полей.

Пример 10.7

Рассмотрим примеры выражений для передачи и приема сообщений через каналы:

```
c!1          /* посылка (сообщение 1 добавится в конец буфера) */

c!ack,p      /* посылка составного сообщения */

c?_          /* удаление первого сообщения из буфера
                (блокируется, если канал пуст) */

c?x          /* чтение первого сообщения буфера в переменную x */

c?5          /* блокировка, если первое сообщение не 5 */

c?eval(x)    /* блокировка, если первое сообщение отлично от
                значения переменной x в момент вызова функции eval */
```

[72] Канал c должен быть проинициализирован, а тип выражения m должен совпадать с типом сообщений, передаваемых по каналу c.

```
c?ack,p       /* блокировка до тех пор, пока первым в буфере не
                 станет сообщение типа ack; чтение информационной
                 части сообщения в переменную p */

c?<_>         /* блокировка, если канал пуст */

c?<ack,_>     /* блокировка, если канал пуст или
                 его первое сообщение имеет тип, отличный от ack */
```

□

10.1.4. Управляющие операторы

Основной управляющей конструкцией языка PROMELA является *оператор выбора* **if**. Оператор задает непустое множество *охраняемых действий* (*guarded actions*) — пар вида *условие – действие*, — из которых в процессе исполнения[73] выбирается только одно:

```
if
:: guard_1 -> action_1
...
:: guard_n -> action_n
fi
```

Более точно семантика оператора выбора определяется следующим образом:

1. в текущем состоянии s проверяется *исполнимость* условий $guard_i$ и составляется множество разрешенных действий:

$$Enabled = \{guard_i -> action_i \mid s \models guard_i\};$$

2. пока $Enabled = \emptyset$, процесс блокируется;
3. среди действий множества $Enabled$ *недетерминировано* выбирается одно, которое и исполняется[74].

В операторе **if** можно использовать специальные условия **else** и **timeout**. Условие **else** может быть только в последнем действии; оно исполнимо тогда и только тогда, когда не исполнимы условия всех остальных действий. Условие **timeout** применяется для отслеживания *тупиков*: оно исполнимо тогда и только тогда, когда все процессы заблокированы.

---

[73] Мы говорим об *исполнении*, хотя исполнения (в традиционном понимании) при проверке модели не происходит.

[74] Заметим: семантика оператора выбора отличается от семантики аналогичного оператора в языке охраняемых команд (Guarded Command Language), предложенного Дейкстрой [Dij75]. В языке Дейкстры оператор выбора завершается, если все условия в нем ложны.

Важное замечание: в качестве условий можно использовать не только формулы, но и операторы, включая операторы присваивания и передачи сообщений. *Исполнимость* — это возможность исполнения оператора, т.е. отсутствие блокировки. Если оператором является формула, исполнимость оператора — истинность формулы. Такие операторы, как **skip** и **printf**, исполнимы всегда.

Заметим, что символы −> и ; являются взаимозаменяемыми — в качестве условия рассматривается первый оператор последовательности. Заметим также, что последовательность может состоять из одного оператора.

Пример 10.8

Ниже приведены примеры операторов выбора и пояснения к ним.

```
if
/* A1 */ :: (a != b) -> ... /* исполняется либо A1,
/* A2 */ :: (a == b) -> ...    либо A2 (блокировка невозможна) */
fi

if
/* B1 */ :: (a >  b) -> ... /* если a < b, возникает блокировка,
/* B2 */ :: (a == b) -> ...    поскольку нет соответствующего действия */
fi

if
/* C1 */ :: (a >  b) -> ... /* исполняется либо C1,
/* C2 */ :: (a == b) -> ...    либо C2,
/* C3 */ :: else     -> ...    либо C3 (блокировка невозможна) */
fi

if
/* D1 */ :: in?x     -> ... /* D1 разрешено, если канал in не пуст,
/* D2 */ :: out!y    -> ...    D2 разрешено, если канал out не полон */
fi
```

□

Еще одной управляющей конструкцией является *оператор повторения* **do**. Его синтаксис аналогичен синтаксису оператора выбора — разница лишь в том, что вместо ключевых слов **if** и **fi** используются **do** и **od**.

Оператор интерпретируется следующим образом:

- пока множество разрешенных действий пусто, процесс блокируется;
- недетерминированно выбирается и исполняется одно из разрешенных действий;
- процесс повторяется.

Для выхода из цикла можно использовать операторы **break** и **goto**. Оператор **break** передает управление на оператор, следующий за оператором повторения. Оператор **goto** передает управление на указанную метку. Метки можно ставить только перед операторами (для перехода в конец тела процесса можно добавить фиктивный оператор **skip** и снабдить его меткой).

Пример 10.9

Использование оператора повторения поясняется на следующих примерах.

```
do
:: count++                    /* счетчик инкрементируется или */
:: count--                    /* декрементируется до тех пор, пока */
:: (count == 0) -> break      /* его значение не станет равным 0 */
                              /* (в этом случае возможен выход) */
od

do
:: c?m-> ...                  /* повторяется чтение сообщения из канала; */
:: timeout -> goto deadlock /* в случае тупика чтение прекращается */
od
deadlock: skip
```

□

В PROMELA можно определять *атомарные действия*: **atomic** $\{action\}$. В идеале атомарная цепочка не должна разрываться планировщиком: ее операции не должны перемешиваться с операциями других процессов. Однако если при исполнении цепочки какой-нибудь оператор блокируется, она принудительно разрывается и управление получает другой процесс. Когда блокировка снимается и исполнение возобновляется, атомарность по возможности обеспечивается для оставшихся операций [Hol03].

Пример 10.10

Рассмотрим описание двоичного семафора — мьютекса (mutex).

```
init:  /* инициализация мьютекса */
  count = 1
...
enter: /* захват мьютекса (вход в критическую секцию) */
  do
  :: atomic { count > 0 -> count-- }; break
  :: else
  od
...
```

```
leave: /* освобождение мьютекса (выход из критической секции) */
  count++
```

□

10.1.5. Средства спецификации требований

Важную часть PROMELA составляют средства спецификации требований. Самое простое из них — *оператор* **assert**, позволяющий задавать *утверждения* для конкретных точек модели. Оператор имеет следующий синтаксис: **assert**(B), где B — логическая формула или целочисленное выражение[75]. Если проверка модели выявляет, что условие, заданное в **assert**, может быть ложным, сообщается об ошибке. Оператор **assert** всегда исполним и не имеет побочных эффектов.

Пример 10.11

Ниже приведена простейшая ошибочная модель.

```
init { assert(false) }
```

□

Для описания темпоральных свойств применяются **never**-*блоки*. Как следует из названия, такие конструкции описывают вычисления модели, которые *никогда* не должны произойти, т.е. являются ошибочными. Для каждого **never**-блока строится *монитор*, который сообщает об ошибке, если поведение модели становится подобным описанному в блоке. Монитор работает[76] *синхронно* с *асинхронной композицией* процессов модели (см. лекцию 12) — он получает управление непосредственно перед исполнением каждого действия и сразу после исполнения последнего. Завершение монитора говорит об ошибочности модели: пока все мониторы работают, поведение модели считается корректным (если, конечно, не были нарушены условия **assert**).

Пример 10.12

Рассмотрим пару тривиальных примеров использования **never**-блоков.

---

[75] Как и в языке C, выражение считается истинным тогда и только тогда, когда его значение отлично от нуля.

[76] Для наглядности мы говорим о *работе* монитора, хотя при проверке модели исполнения не происходит (см. лекцию 12).

```
never { true } /* что бы модель ни делала, она ошибочна */
never { P@L }  /* ошибка, если вначале процесс P находится на метке L */
```

□

Пример 10.13

Определим **never**-блок, проверяющий выполнимость свойства $\mathbf{G}p$ [Hol03]. В негативном варианте это свойство выглядит как $\mathbf{never}\ \mathbf{F}\neg p$ (никогда высказывание p не становится ложным).

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

Как только нарушается свойство p, осуществляется выход из цикла, процесс **never**-блока завершается, что интерпретируется как ошибка. Пока высказывание p истинно, цикл исполняется.

□

Never-блоки позволяют описывать не только ошибочное поведение, выявляемое на конечных вычислениях, но и ошибки, обнаружить которые можно только на бесконечных траекториях (мы поговорим об этом в лекции 12). Для этого используются специальные метки с префиксом *accept*: если оператор, помеченный такой меткой, может исполниться бесконечное число раз, фиксируется ошибка.

Пример 10.14

В следующем примере осуществляется проверка свойства $\mathrm{GF}p$. Негативная формулировка имеет вид $\mathrm{never}\ \mathrm{FG}\neg p$ (высказывание p не становится ложным навсегда).

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

Каждый раз, когда высказывание p становится ложным, есть две возможности: выйти из цикла или остаться в нем. При выходе из цикла вычисления попадают в цикл, помеченный меткой $accept$, который исполняется, пока высказывание p ложно.

□

В новой редакции PROMELA, поддерживаемой в SPIN версии не ниже 6, имеются средства спецификации требований с помощью формул LTL [Ben10][77]:

$$\textbf{ltl}\ name\ \{\varphi\},$$

где $name$ — необязательное имя, а φ — формула, задаваемая следующей грамматикой:

$$\varphi ::= p \mid true \mid false \mid (\varphi) \mid []\varphi \mid <> \varphi \mid !\,\varphi \mid \varphi\ \mathbf{U}\ \varphi \mid \varphi\ \mathbf{W}\ \varphi \mid \varphi\ \mathbf{V}\ \varphi \mid$$

$$\mid \varphi\ \&\&\ \varphi \mid \varphi \mid\mid \varphi \mid \varphi -> \varphi \mid \varphi <-> \varphi.$$

Здесь p — произвольная логическая формула или целочисленное выражение. Смысл указанных операций приводится в таблице 10.3.

Таблица 10.3. Пояснения к операциям LTL в языке PROMELA

Операция	Комментарий
[] или **always**	темпоральный оператор **G** (*Globally*)
<> или **eventually**	темпоральный оператор **F** (*in the Future*)
!	отрицание
U	темпоральный оператор **U** (*Until*)
W	темпоральный оператор **W** (*Weak until*)
V	темпоральный оператор **R** (*Release*)
&& или /\	конъюнкция
|| или \/	дизъюнкция
−>	импликация
<−>	эквивалентность

Темпоральный оператор **X** (*neXt time*) не поддерживается — для асинхронных систем он не имеет смысла. Операторы **U**, **W** и **V** ассоциативны слева направо.

[77] SPIN автоматически берет отрицание LTL-формулы и строит соответствующий **never**-блок.

Пример 10.15

Ниже приводится LTL-спецификация алгоритма Петерсона взаимного исключения процессов (см. лекцию 9).

```
ltl mutex     { []!(P[0]@CRS && P[1]@CRS) }
ltl liveness0 { [](P[0]@SET -> <>P[0]@CRS) }
ltl liveness1 { [](P[1]@SET -> <>P[1]@CRS) }
```

□

10.2. Введение в инструмент SPIN

Инструмент SPIN (Simple Promela Interpreter) предназначен для анализа PROMELA-моделей. Он был разработан в 1980-х гг. как средство проектирования и валидации компьютерных протоколов [Hol91]. С 1991 г. инструмент распространяется бесплатно вместе с исходными кодами. В 2002 г. за разработку SPIN Джерард Хольцман был удостоен награды ACM Software System Award.

Нас прежде всего интересуют функции SPIN, связанные с формальной верификацией, а именно с *проверкой моделей* (*model checking*). Слово «модель» здесь допускает двоякую трактовку: *инженерную* (модель — это любая PROMELA-программа) и *логическую* (модель — это интерпретация формулы, в которой эта формула истинна). Под проверкой модели понимается процедура, выявляющая, является ли заданная модель в инженерном понимании моделью в логическом понимании для заданной спецификации, т.е. истинны ли спецификации на модели. Теоретические основы метода проверки моделей для логики LTL рассматриваются в следующих лекциях.

Одна из причин популярности SPIN — его эффективность. За годы существования инструмента в нем были воплощены многие оптимизации: редукция частичных порядков, сжатие битового представления состояний, хэширование без разрешения коллизий и многие другие [Hol03]. Кроме того, для достижения высокой производительности SPIN генерирует программу на языке C, которая решает поставленную задачу анализа.

10.2.1. Структура инструмента и сценарий его использования

Инструмент SPIN включает несколько компонентов: PROMELA-парсер, симулятор моделей, генератор программы проверки и графическую оболочку [Hol03]. Типичный сценарий использования SPIN выглядит следующим образом:

1. Разрабатывается модель компьютерной системы на языке PROMELA и специфицируются требования к ней. Исправляются синтаксические ошибки, выданные PROMELA-парсером.
2. Осуществляется интерактивная симуляция и отладка модели. Процесс завершается, как только пользователь — инженер-верификатор — становится более-менее уверенным в корректности модели.
3. Генерируется программа проверки на языке C (при этом могут быть указаны опции оптимизации). Программа компилируется и запускается.
4. Если при проверке модели обнаруживаются ошибки, для них автоматически строятся *контрпримеры* – вычисления, нарушающие спецификации. Контрпримеры могут быть поданы симулятору для отладки модели.

10.2.2. Основные опции SPIN

Рассмотрим некоторые опции инструмента SPIN в том порядке, в котором они обычно используются при верификации PROMELA-моделей [Hol03].

Рандомизированная симуляция модели:

```
spin [-p] [-uчисло-шагов] имя-файла
```

Если указана опция $-p$, выводится детальная информацию по каждому шагу исполнения; иначе — только результаты вызовов **printf**. Опция $-u$ предназначена для ограничения времени симуляции (например, $-u100$ ограничивает симуляцию 100 шагами).

Генерация, компиляция и запуск программы проверки:

```
spin -a имя-файла
cc -o pan pan.c
./pan
```

Первая команда генерирует С-программу `pan.c`, осуществляющую проверку модели; вторая компилирует ее (это можно сделать любым компилятором); третья запускает.

Воспроизведение контрпримера на симуляторе:

```
spin -t имя-файла
```

Если при верификации модели была обнаружена ошибка и построен контрпример, приведенная команда воспроизводит ошибку на симуляторе. Для детальной диагностики может использоваться опция $-p$.

По соглашению, PROMELA-код помещают в файлы с расширением `pml`.

10.3. Вопросы и упражнения

1. Реализуйте на языке PROMELA алгоритм Евклида нахождения наибольшего общего делителя. С помощью оператора **assert** специфицируйте пред- и постусловия. Используя симулятор SPIN, протестируйте модель на некоторых значениях аргументов.

2. Определите, сколько состояний в следующей PROMELA-модели [Hol03]:

   ```
   init {
     byte i = 0;
     do
     :: i = i + 1
     od
   }
   ```

   Запустите эту модель в симуляторе SPIN.

3. Гарантирует ли приведенный ниже монитор проверку свойства $invariant$ во всех состояниях модели в режиме симуляции?

   ```
   active proctype monitor() {
     do
     :: assert(invariant)
     od
   }
   ```

4. Каким может стать значение переменной $state$ после исполнения модели следующего вида [Hol03]?

   ```
   byte state = 1;
   ```

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

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

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

Проверьте свою догадку с помощью инструмента SPIN.

5. Каким станет множество возможных значений переменной $state$ (см. предыдущее задание), если внести следующие изменения в код процессных типов A и B?

   Вариант a:

   ```
   (state == 1) -> atomic { tmp = state; tmp = tmp + 1; state = tmp }
   (state == 1) -> atomic { tmp = state; tmp = tmp - 1; state = tmp }
   ```

   Вариант b:

   ```
   atomic { (state == 1) -> tmp = state; tmp = tmp + 1; state = tmp }
   atomic { (state == 1) -> tmp = state; tmp = tmp - 1; state = tmp }
   ```

6. Смоделируйте на языке PROMELA распределенную систему следующего вида. Имеется один сервер, который обрабатывает запросы от N клиентов. Для передачи запросов используется один буферизованный канал емкости M. Ответы возвращаются по индивидуальным синхронным каналам. Запрос — это сообщение, содержащее номер клиента. Ответ — тот же номер, но увеличенный на 1. Клиент не посылает запрос, пока не завершится обработка предыдущего. Сервер не переходит к обработке запроса, пока не завершит обработку предыдущего.

   Задайте требования к системе, используя конструкцию **assert**, и проверьте корректность модели с помощью инструмента SPIN (для каких-нибудь фиксированных значений N и M).

7. Для предыдущего задания опишите с помощью формулы LTL свойство живости системы: если клиент посылает запрос, рано или поздно он его пошлет и, более того, получит ответ.

   Проверьте истинность этого свойства на модели с помощью инструмента SPIN.

8. Добавьте в модель сервера «балансировку» запросов от клиентов. Сервер хранит номер приоритетного клиента (начальное значение — 0). Если во входном канале есть запрос от клиента с этим номером, запрос принимается, а номер инкрементируется (по модулю N); в противном случае попытка осуществляется с измененным номером.

   Изменится ли от добавления планировщика истинность свойства живости?

9. Представьте следующие требования в форме **never**-блока:

   a. в процессе работы системы состояние, удовлетворяющее свойству φ может возникать только конечное число раз;

   b. между каждой парой состояний, удовлетворяющих свойству φ, обязательно встретится состояние, удовлетворяющее свойству ψ.

10. Определите **never**-блоки для отрицаний следующих формул LTL:

    a. $<> []p$;

    b. $[](p -> <> q)$;

    c. $[](p -> (p \mathbf{U} q))$.

11. Реализуйте на языке PROMELA алгоритм Петерсона (см. лекцию 9), задайте условия корректности, используя **ltl**-блоки, и проверьте корректность алгоритма с помощью инструмента SPIN.

12. Специфицируйте условия корректности алгоритма Петерсона, используя **never**-блоки.

13. Предложите на языке PROMELA решение *проблемы обедающих философов*. N молча думающих философов сидят за круглым столом. Перед каждым философом стоит тарелка спагетти, а между каждой парой лежит вилка. Философ может либо есть, либо размышлять. Есть одно ограничение — философ может есть только тогда, когда одновременно держит в руках две вилки. Философ может взять одну из ближайших вилок со стола (если она доступна) или положить вилку на стол (если уже держит ее). Взятие и возвращение разных вилок являются раздельными действиями. Задача состоит в том, чтобы разработать модель поведения, в рамках которой ни один из философов не будет голодать, т.е. будет вечно чередовать прием пищи и размышления.

14. Формализуйте операционную семантику языка PROMELA.

Лекция 11. Моделирование программ структурами Крипке

Простота — вот единственная почва, на которой мы можем строить здание наших обобщений. Но если эта простота — только кажущаяся, то будет ли достаточно надежной и сама почва? Вопрос этот заслуживает исследования.

А. Пуанкаре.
Наука и гипотеза

Рассматриваются вопросы адекватности моделирования программ структурами Крипке, в том числе вопросы выбора уровня абстракции данных и гранулярности действий. Обсуждаются возможные аномалии: появление поведения неприсущего исходной программе и связанные с этим ложные сообщения об ошибках (false positives), а также исчезновение возможного поведения и связанные с этим пропуски ошибок (false negatives). Дается представление о предикатной абстракции программ и адаптивном уточнении абстракции на основе контрпримеров — методе CEGAR (Counter Example Guided Abstraction Refinement). Рассматриваются базовые подходы к исследованию пространства состояний параллельной программы, позволяющие по моделям процессов строить модель всей программы.

11.1. Адекватность моделирования программ структурами Крипке

Семантика LTL определяется на траекториях структур Крипке, т.е. на бесконечных последовательностях пометок — множеств истинных элементарных высказываний (см. лекцию 9). В вычислениях программ (см. лекцию 3) таких пометок нет, однако никто не мешает ввести высказывания о состояниях программы и использовать их для выражения требований. Например, если в программе есть переменные x и y, можно определить высказывание (предикат) $p \equiv (y > x)$ и, «измерив» его истинность на всех состояниях некоторого вычисления проверить свойство $\mathbf{G}\neg p$ (никогда значение переменной y не превосходит значения переменной x). Таким образом, формально определенная операционная семантика языка программирования и множество предикатов над состояниями программы «превращают» заданное вычисление программы в траекторию.

Для формальной верификации интерес представляют не одиночные траектории, а *все* возможные траектории. Для их представления нужно по программе построить ее модель в форме структуры Крипке. Решение «в лоб» — построить граф состояний программы, используя операционную семантику языка программирования, и пометить каждое состояние

множеством всех истинных в нем высказываний (предикатов, используемых в формулировке требований). Очевидно, что такой подход работает не всегда: программа как математический объект может иметь бесконечное число состояний; даже если их число конечно, оно может быть чрезмерно большим для практического анализа. Нужна абстракция — игнорирование тех или иных деталей программы (состояний, переходов и т.п.).

Следующие разделы рассматривают кратко вопросы абстракции данных и действий, а также вопросы адекватности моделирования.

11.1.1. Состояния: абстракция данных

Состояние, или конфигурация, является естественной концепцией, используемой в программировании (см. лекции 2, 5 и 9). Это «мгновенный снимок» исполнения программы, включающий *состояние данных* (значения переменных) и *состояние управления* (точки исполнения процессов). Пренебрегая некоторыми деталями, состояния можно объединять в классы эквивалентности, называемые *абстрактными состояниями*. Формализуем эту идею применительно к структурам Крипке.

Пусть заданы множество элементарных высказываний AP и структура Крипке $\langle S, S_0, R, L\rangle$, которую мы будем называть *конкретной системой переходов*. Пусть $\hat{S}$ — некоторое непустое множество — множество *абстрактных состояний*. *Функцией абстракции* называется отображение $f: S \to \hat{S}$, такое что для всех $s, s' \in S$ имеет место импликация

$$\big(f(s) = f(s')\big) \to \big(L(s) = L(s')\big).$$

Абстрактной системой переходов, порожденной функцией абстракции f, называется система $\langle \hat{S}, \hat{S}_0, \hat{R}, \hat{L}\rangle$, где [Bai08]

$$\hat{S}_0 = \{f(s) \mid s \in S_0\}, \quad \hat{R} = \{(f(s), f(s')) \mid (s, s') \in R\} \quad \text{и}$$

$$\hat{L}\big(f(s)\big) = L(s) \text{ для всех } s \in S.$$

Пример 11.1

При моделировании протоколов передачи данных можно абстрагироваться от содержимого передаваемых сообщений. Как правило, логика протокола определяется типами со-

общений, их порядковыми номерами, но не передаваемыми данными. Удалив из структуры сообщения поля с данными (см. таблицу 11.1), можно значительно сократить число состояний модели, сохранив при этом ее адекватность.

Таблица 11.1. Абстрагирование от содержимого сообщений

Структура конкретных сообщений	Структура абстрактных сообщений

typedef ConcreteMessage {
/* порядковый номер сообщения */
unsigned seq:4;
/* контрольная сумма */
unsigned checksum:16;
/* передаваемые данные */
byte data[32];
}
	typedef AbstractMessage {
	/* порядковый номер сообщения */
	unsigned seq:4;
	/* признак ошибки передачи */
	bool error;
	/* передаваемые данные */
	/* не моделируются */
	}

□

Заметим: абстракция может приводить к появлению траекторий, не присущих исходной модели. Предположим, что в модели есть два состояния s_1 и s_2, имеющие одинаковые пометки, $L(s_1) = L(s_2)$, и объединяемые в одно абстрактное состояние: $f(s_1) = f(s_2)$; здесь f — некоторая функция абстракции (см. рис. 11.1). Пусть в исходной модели имеются переходы $s_1' \to s_1 \to s_1''$ и $s_2' \to s_2 \to s_2''$ (из s_1 и s_2 есть только переходы в s_1'' и s_2'' соответственно), при этом

$$L(s_1') = b \neq c = L(s_2'), \qquad L(s_1) = a = L(s_2), \qquad L(s_1'') = d \neq e = L(s_2'').$$

Тогда в абстрактной модели возможны траектории, включающие цепочки bae и cad, что невозможно в исходной модели.

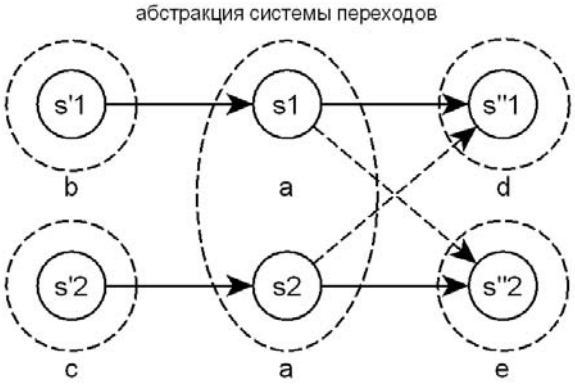


Рисунок 11.1. Появление дополнительных траекторий при абстракции

Появление дополнительных траекторий в абстрактных моделях может привести к *ложным сообщениям об ошибках* (*false positives*), или, как их называют в математической статистике, *ошибкам первого рода*[78]. Для каждого найденного примера ошибочного поведения, так называемого *контрпримера*, нужно проверить, реализуем ли он в исходной модели: если нет, это не ошибка. Безусловно, ложные тревоги доставляют неприятности при анализе моделей, но гораздо более опасны *пропуски ошибок* (*false negatives*) — ошибки второго рода.

Если ошибки первого рода допустимы, то ошибки второго рода неприемлемы ни при каких обстоятельствах. Будем называть модель *адекватной*, если она не допускает ошибок второго рода, а ее анализ требует привлечения приемлемых ресурсов (включая затраты на разбор ошибок первого рода).

Обратите внимание: при абстракции рассмотренного вида (основанной на объединении состояний в классы эквивалентности) пропуски ошибок исключены.

Пример 11.2

В таблице 11.2 приведен фрагмент кода на языке PROMELA (слева) и результат его абстракции (справа). В абстрактной модели игнорируется переменная x: присваивания в x удалены, а условия с участием x заменены на булевы переменные $p0$ и $p1$, которые могут принимать произвольные значения истинности.

Таблица 11.2. Абстрагирование от переменной x (появление ошибочных вычислений)

Исходная модель	Результат абстракции (1)

`x = x * y;`
 	
`if`
`:: (x % 3 == 0) -> r = 0`
`:: (x % 3 == 1) -> r = 1`
`:: else -> r = 2`
`fi;`
	`select(p0 : 0 .. 1);`
	`select(p1 : 0 .. 1);`
	
	`if`
	`:: p0 -> r = 0`
	`:: p1 -> r = 1`
	`:: else -> r = 2`
	`fi;`

[78] Ошибки первого и второго родов — ключевые понятия проверки статистических гипотез. Ошибка первого рода — *нулевая гипотеза* (например, гипотеза «в программе нет ошибок») неверно отвергнута; ошибка второго рода — нулевая гипотеза неверно принята.

```
if
:: (x % 3 == 0) -> assert(r == 0)
:: (x % 3 == 1) -> assert(r == 1)
:: else         -> assert(r == 2)
fi
```

```
if
:: p0   -> assert(r == 0)
:: p1   -> assert(r == 1)
:: else -> assert(r == 2)
fi
```

Очевидно, что абстрактная модель допускает ошибочные вычисления, которые невозможны в исходной модели, например:

$$r = 0;\ \textbf{assert}(r == 1).$$

Это связано с тем, что в модели не учитывается семантика условий: $p0$ и $p1$ могут одновременно получить значение $true$, что соответствует формуле

$$(x\ \%\ 3 == 0)\ \&\&\ (x\ \%\ 3 == 1),$$

очевидно, невыполнимой. Такой ситуации можно избежать, если при выборе значений истинности исключить заведомо недостижимые комбинации:

```
select(p0 : 0 .. 1);         /* p0 принимает произвольное значение */
select(p1 : 0 .. (1 - p0)); /* если p0 = 1, p1 может быть только 0 */
```

□

Если число состояний данных огромно, но данные используются простым образом (условий на данные немного, между условиями нет скрытых тавтологий, данные не задействуются в условиях корректности), можно абстрагироваться от состояния данных, построив при этом адекватную модель со сравнительно небольшим числом состояний.

11.1.2. Переходы: гранулярность действий

Понятие состояния неотделимо от понятия перехода — элементарного изменения состояния. Важным вопросом является соотношение между переходами программы и ее модели. Переходы модели по определению *атомарны*, в то время как аналогичные действия программы могут состоять из нескольких шагов. Моделирование длительных процессов атомарными переходами может приводить к пропуску ошибок. Излишняя детализация также вредна: если неделимое действие моделируется несколькими переходами, при анализе могут возникать ложные сообщения об ошибках; кроме того, это увеличивает число состояний модели и делает верификацию более ресурсоемкой.

Пример 11.3

Процедура вычисления значения выражения, как правило, не атомарна. В машинном коде ей соответствует несколько команд микропроцессора: загрузка данных из памяти в регистры; исполнение операций над регистрами; сохранение результата в память. Исполнение программы из n параллельных процессов вида $x := x + 1$ (точнее, $t_i := x;\ x := t_i + 1$, где t_i — локальная переменная i-ого процесса) для начального состояния $x = 0$ может привести к любому из вариантов: $x = 1, \ldots, x = n$ (см. таблицу 11.3).

Таблица 11.3. Варианты исполнения n параллельных присваиваний $x := x + 1$

Вариант исполнения 1			Вариант исполнения n	
$P_{i_1}: t_{i_1} := x$	$t_{i_1} = 0$		$P_{i_1}: t_{i_1} := x$	$t_{i_1} = 0$
оставшиеся процессы			$P_{i_1}: x := t_{i_1} + 1$	$x = 0 + 1 = 1$
могут выполняться в	...	...	...	...
произвольном порядке			$P_{i_n}: t_{i_n} := x$	$t_{i_n} = n - 1$
$P_{i_1}: x := t_{i_1} + 1$	$x = 0 + 1 = 1$		$P_{i_n}: x := t_{i_n} + 1$	$x = (n - 1) + 1 = n$

□

11.2. Представление программ структурами Крипке

В этом разделе мы на конкретных примерах рассмотрим представление программ структурами Крипке. В каждом примере для заданной программы P и ее спецификации φ строится структура Крипке M, определяющая поведение P на том или ином уровне абстракции и позволяющая выразить свойство φ. Также поднимается вопрос построения модели параллельной программы по моделям составляющих ее процессов.

11.2.1. Примеры представления программ структурами Крипке

Пример 11.4

Начнем с нетипичного для проверки моделей примера — вычислительной программы, а именно программы целочисленного деления.

```
proctype DIV(int a, b) {
  int q, r;
  assert(a >= 0 && b > 0); /* предусловие */
```

```
  atomic {
    q = 0;
    r = a
  }
  do
  :: atomic {
       r >= b -> q = q + 1;
                 r = r - b
     }
  :: else  -> break
  od;
  assert(a == b*q + r && 0 <= r && r < b) /* постусловие */
}
```

Представим состояния программы парами значений переменных q и r. Для любых $a \geq 0$ и $b > 0$ число состояний конечно; например, для $a = 5$ и $b = 2$ имеем 4 состояния (включая начальное состояние с нулевыми значениями q и r). Отношение переходов между состояниями показано на рис. 11.2: начальное состояние отмечено входящей в него стрелкой; петля в последнем состоянии добавлена для того, чтобы сделать отношение переходов полным (см. лекцию 9).

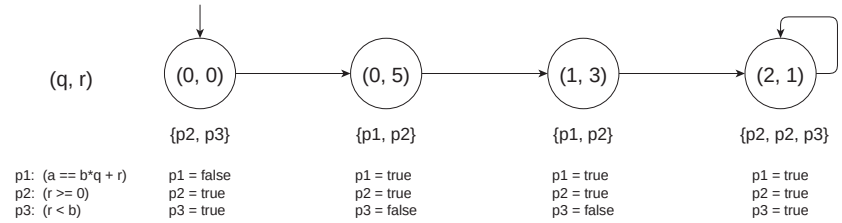


Рисунок 11.2. Структура Крипке, моделирующая программу целочисленного деления для заданных значений входных параметров

Введя элементарные высказывания $p_1 \equiv (a = b \cdot q + r)$, $p_2 \equiv (r \geq 0)$ и $p_3 \equiv (r < b)$, а также сопоставив состояниям программы значения истинности этих высказываний, получим структуру Крипке. Эта структура имеет единственное вычисление π, на котором можно интерпретировать формулы LTL над множеством $AP = \{p_1, p_2, p_3\}$, например, следующие:

- $\pi \vDash \mathbf{F}\{p_1 \wedge p_2 \wedge p_3\}$ — это, по сути, условие корректности программы: когда-нибудь будет достигнуто состояние, в котором переменная q содержит частное от деления a на b, а r — остаток;

• $\pi \vDash \mathbf{FG}\{p_1 \wedge p_2 \wedge p_3\}$ — это усиленный вариант условия корректности, выражающий также свойство *стабилизации*: через конечное число шагов будет достигнуто состояние, содержащее корректный результат, после чего состояние меняться не будет;

• $\pi \vDash \mathbf{XG}\{p_1 \wedge p_2\}$ — это свойство утверждает, что всегда (за исключением начального состояния) истинен инвариант цикла $(a = b \cdot q + r) \wedge (r \geq 0)$.

□

Приведенная в примере структура Крипке построена для конкретных значений a и b (*параметров* модели), поэтому не пригодна для верификации программы целочисленного деления в целом. Для верификации *параметризованных моделей* (для всех допустимых значений параметров) разработаны специальные методы (*parameterized model checking*). Они активно применяются для верификации распределенных алгоритмов и протоколов в их общем виде [Mcm01].

Отметим: приведенная структура Крипке имеет мало общего с представлением систем переходов в инструменте SPIN [Hol03].

Пример 11.5

Рассмотрим алгоритм Петерсона решения задачи взаимного исключения двух процессов (см. лекцию 9).

```
active [2] proctype P() {
  int i = _pid;
  NCS: skip; /* некритическая секция */
  SET: flag[i] = 1; turn = i;
  TST: !(flag[1 - i] == 1 && turn == i);
  CRS: skip; /* критическая секция */
  RST: flag[i] = 0; goto NCS
}
```

Если в прошлом примере в качестве состояний структуры Крипке мы использовали состояния данных программы (значения переменных), то на этот раз нас будут интересовать состояния управления (метки операторов). Система переходов одного процесса приведена на рис. 11.3. Состояние SET' соответствует оператору $turn = i$; состояние RST' — **goto** NCS. Для выражения свойства взаимного исключения нужны два элементарных высказывания:

1. $p_0 \equiv P[0]@CRS$ — процесс $P[0]$ находится в критической секции;
2. $p_1 \equiv P[1]@CRS$ — процесс $P[1]$ находится в критической секции.

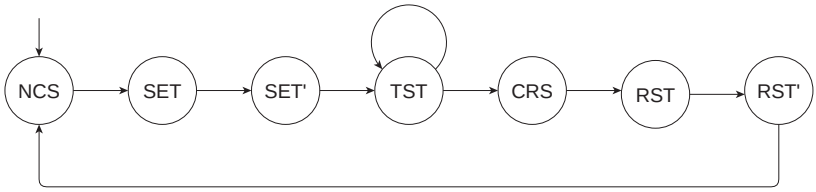


Рисунок 11.3. Система переходов одного процесса в алгоритме Петерсона

Чтобы проверить свойство взаимного исключения процессов, $\mathbf{G}\{\neg(p_0 \wedge p_1)\}$, необходимо построить модель параллельной программы — *асинхронную композицию* моделей процессов. Состояниями модели являются комбинации состояний, составляющих ее процессов, а каждый переход соответствуют переходу в одном из процессов (см. лекцию 9).

Асинхронная композиция независимых процессов включает все возможные комбинации их состояний; если же между процессами есть зависимости, то некоторые из комбинаций оказываются недостижимыми. Резкий, как правило, экспоненциальный, рост числа состояний параллельной программы в зависимости от числа процессов называется *комбинаторным взрывом числа состояний* (*state explosion*).

На рис. 11.4 изображена асинхронная композиция процессов из рассматриваемого примера. Для наглядности состояния организованы в форме *дерева поиска*. Каждое состояние изображено кружком и помечено уникальной меткой — порядковым номером при обходе. Если из состояния есть переход в уже пройденное состояние, это изображается стрелкой, ведущей в квадрат с соответствующей меткой. Дерево содержит все комбинации состояний процессов, включая недостижимые (таким было бы пространство состояний, если бы процессы были независимыми), недостижимые состояния закрашены серым цветом. Видно, что свойство $\neg(p_0 \wedge p_1)$ истинно во всех достижимых состояниях.

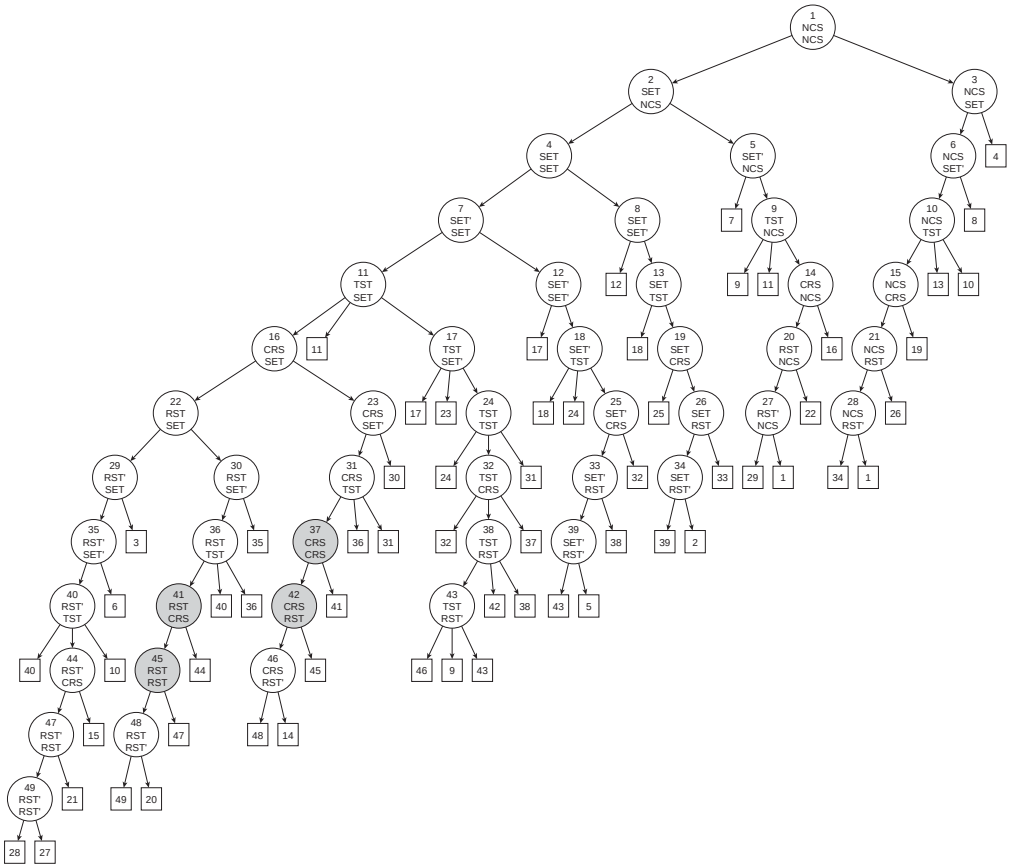


Рисунок 11.4. Асинхронная композиция систем переходов процессов в алгоритме Петерсона

□

11.2.2. Построение асинхронной композиции систем переходов

Существует два базовых подхода к *исследованию пространства состояний* параллельной программы (*state space exploration*): *поиск в ширину* (*BFS*, *Breadth-First Search*) и *поиск в глубину* (*DFS*, *Depth-First Search*).

При поиске в ширину состояния просматриваются по уровням:

- уровень 0 — множество начальных состояний;
- уровень $(n + 1)$ — множество состояний, непосредственно достижимых из состояний уровня n (за исключением состояний предшествующих уровней).

У поиска в глубину иная стратегия: если у текущего состояния есть еще не пройденные последователи, поиск продолжается с одного из них; в противном случае выполняется возврат к предыдущему состоянию.

В таблице 11.4 представлено более формальное описание методов исследования пространства состояний. Предполагается, что есть только одно начальное состояние (s_0). Множество S содержит посещенные состояния (вначале $S = \{s_0\}$). Вызов $succ(s)$ возвращает множество всех непосредственных последователей состояния s. Функция $choose$ выбирает произвольный элемент непустого множества.

Таблица 11.4. Поиск в ширину и поиск в глубину

Поиск в ширину	Поиск в глубину

`S := {s0};`
`queue := {s0};`
`while queue ≠ ∅ do`
`  s := queue.pop();`
`  /* действие над состоянием s */`
`  for s' in (succ(s) \ S) do`
`  queue.push(s')`
`  end;`
`  S := S ∪ succ(s)`
`end`
	`S := {s0};`
	`stack := {s0};`
	`while stack ≠ ∅ do`
	`  s := stack.top();`
	`  if (succ(s) \ S) = ∅ then`
	`  /* действие над состоянием s */`
	`  stack.pop()`
	`  else`
	`  s' := choose(succ(s) \ S);`
	`  S := S ∪ {s'};`
	`  stack.push(s')`
	`  end`
	`end`

У каждого подхода есть как достоинства, так и недостатки. Поиск в ширину предпочтительнее, когда велика вероятность того, что проверяемое свойство состояния может нарушаться в состояниях, недалеко отстоящих от начального. Поиск в глубину более естественен в том смысле, что он сохраняет порядок вычислений — в каждом состоянии (за исключением случаев, когда все последователи пройдены) исполняется переход в последующее состояние. По этой причине поиск в глубину широко применяется в *тестировании на основе моделей* (*model-based testing*) (см. лекцию 15), — реальную систему, в отличие от модели, трудно перевести в произвольное состояние [Бур03a], [Бур03b].

11.3. Предикатная абстракция и уточнение абстракции по контрпримерам

Вернемся к теме построения моделей, или, как еще говорят, абстракций, последовательных программ. В этом разделе мы коснемся подхода, известного как *предикатная абстракция*, и связанного с ним метода *адаптивного уточнения абстракции по контрпримерам — CEGAR* (*Counter Example Guided Abstraction Refinement*) [Cla03]. Для более подробного знакомства с методом мы рекомендуем обзор [Ман13], содержащий обширную библиографию по этой теме.

Метод CEGAR применяется для проверки достижимости определенной точки программы. Основная идея состоит в следующем. Проверяется модель программы: если анализ выявляет, что модель корректна (заданная точка недостижима), то корректна и исходная программа (ошибки второго рода исключены); в противном случае для модели строится *контрпример* — путь в заданную точку: если этот путь реализуем и в исходной программе, программа некорректна; иначе негативный результат проверки есть всего лишь ложная тревога, являющаяся следствием абстрагирования от чего-то существенного. От чего именно, выявляется путем анализа контрпримера; модель уточняется, и к ней применяется тот же алгоритм. В результате ряда итераций метод CEGAR либо найдет путь в заданную точку, либо докажет ее недостижимость, либо, есть и такая возможность, упрется в ограничения по ресурсам.

11.3.1. Декартова предикатная абстракция

Напомним, что *состоянием данных* программы, или просто *состоянием*, называется отображение, сопоставляющее переменным программы их значения, — так называемая *функция означивания*. Пусть p — предикат, заданный на множестве состояний S. Обозначим множество состояний, в которых предикат p истинен, как $[\![p]\!]$, т.е. $[\![p]\!] = \{s \in S \mid s \vDash p\}$. Введем специальные предикаты $false$ и $true$:

$$[\![false]\!] = \emptyset \quad \text{и} \quad [\![true]\!] = S.$$

Пусть задано конечное множество предикатов $\{p_1, \dots, p_n\}$. *Регионом* называется предикат вида $\bigwedge_{k=1}^{m} p_{i_k}$, где $1 \le i_1 < \dots < i_m \le n$, либо $false$. Заметим: если $m = 0$, регион совпадает с $true$.

Метод абстракции, состоящий в «замене» конкретных состояний программы на формулы, описывающие множества состояний, называется *предикатной абстракцией*. Если формулы имеют вид регионов (для некоторого заданного множества предикатов), метод называется *декартовой предикатной абстракцией*.

Пример 11.6

Пусть $p_1 \equiv (x \geq 0)$ и $p_2 \equiv (x \leq 0)$. Выпишем все возможные регионы для множества предикатов $\{p_1, p_2\}$:

- $r_1 \equiv true$;
- $r_2 \equiv p_1 \equiv (x \geq 0)$;
- $r_3 \equiv p_2 \equiv (x \leq 0)$;
- $r_4 \equiv p_1 \wedge p_2 \equiv (x \geq 0) \wedge (x \leq 0) \equiv (x = 0)$;
- $r_5 \equiv false$.

□

Конфигурацией программы называется пара $\langle l, s \rangle$, где l — состояние управления (метка оператора), а s — состояние данных. Пара $\langle l, r \rangle$, где l — состояние управления, а r — регион, называется *абстрактной конфигурацией*.

Для заданной программы P рассмотрим произвольный оператор op. Зафиксируем регион r и определим регион r', в который перейдет регион r под действием оператора op: для всех $s \in S$ и $s' \in M[\![op]\!](s)$ из того, что $s \in [\![r]\!]$, следует, что $s' \in [\![r']\!]$[79]. Если оператор op помечен меткой l, а меткой следующего оператора является l', то op переводит абстрактную конфигурацию $\langle l, r \rangle$ в $\langle l', r' \rangle$.

Пусть $sp(op, r)$ — сильнейшее постусловие оператора op относительно предусловия r (см. лекцию 2). Тогда, r' — подходящий регион, если (и только если) общезначима импликация $sp(op, r) \rightarrow r'$. Очевидно, что этому условию удовлетворяет регион $true$, однако интерес

[79] Обратите внимание: регион аппроксимирует множество возможных состояний *сверху*, т.е. множество возможных состояний является подмножеством множества состояний региона.

представляет «наименьший» из возможных регионов. Воспользуемся следующими тождествами:

$$\left(p \to \bigwedge_{i=1}^{m} p_{i_k}\right) \equiv \left(\neg p \vee \bigwedge_{i=1}^{m} p_{i_k}\right) \equiv \left(\bigwedge_{i=1}^{m} (\neg p \vee p_{i_k})\right);$$

$$\neg\left(p \to \bigwedge_{i=1}^{m} p_{i_k}\right) \equiv \left(\bigvee_{i=1}^{m} (p \wedge \neg p_{i_k})\right).$$

Соответственно, формула $sp(op, r) \to \bigwedge_{k=1}^{m} p_{i_k}$ общезначима тогда и только тогда, когда невыполнима (UNSAT) ни одна из конъюнкций $sp(op, r) \wedge \neg p_{i_k}$. Исходя из вышесказанного, регион r' вычисляется по следующему правилу:

- если формула $sp(op, r)$ невыполнима, то $r' = false$;
- в противном случае $r' = \bigwedge_{k=1}^{m} p_{i_k}$, где

$$\{p_{i_1}, \dots, p_{i_m}\} = \{p_i \mid i \in \{1, \dots, n\} \wedge (sp(op, r) \wedge \neg p_i) \text{ UNSAT}\}.$$

Напомним: построение сильнейшего предусловия базируется на *SSA-представлении* программы (*Static Single Assignment*), в котором в каждую переменную может быть не более одного присваивания [Aho06]. В SSA-представлении используются так называемые *версии* переменных — номер версии инкрементируется при каждом присваивании.

Сильнейшее постусловие программы P относительно предусловия φ определяется следующим образом:

$$sp(P, \varphi) = \big(\varphi[\text{оригиналы} \coloneqq \text{версии}_\text{н}] \wedge ssa(P)\big)[\text{версии}_\text{к} \coloneqq \text{оригиналы}].$$

Здесь $ssa(P)$ — SSA-представление программы P (в виде логической формулы, в которой, в частности, присваивания $x \coloneqq t$ заменены на равенства $x = t$), оригиналы — оригинальные переменные программы P, $\text{версии}_\text{н}$ и $\text{версии}_\text{к}$ — соответственно начальные и конечные версии переменных.

Обратите внимание: множество регионов образуют полное частично упорядоченное множество, а описанное правило вычисления регионов обладает свойством монотонности, что позволяет считать декартову предикатную абстракцию разновидностью абстрактной интерпретации (см. лекцию 7).

11.3.2. Уточнение абстракции по контрпримерам (CEGAR)

Сразу выбрать подходящий уровень абстракции модели — задача непростая. В последнее время получил распространение адаптивный подход, уже упомянутый CEGAR, когда, начиная с грубого приближения, модель шаг за шагом уточняется на основе получаемых контрпримеров. Ниже представлена схема метода применительно к декартовой предикатной абстракции. Для определенности считается, что проверяемое свойство — это недостижимость оператора, помеченного меткой $error$**.**

0. Текущее множество предикатов полагается пустым.
1. Строится абстрактная модель программы для текущего множества предикатов.
2. Проверяется корректность абстрактной модели (недостижимость точки $error$):
    a. Если модель корректна, корректна и исходная программа.
    b. Иначе исполняется шаг 3.
3. Строится контрпример π — вычисление из начальной конфигурации в конфигурацию вида $\langle error, r\rangle$, где r — регион, отличный от $false$.

    Конструируется сильнейшее постусловие $sp(op(\pi), true)$[80], где $op(\pi)$ — цепочка операторов SSA-представления, соответствующая вычислению π.
4. Проверяется выполнимость формулы $sp(op(\pi), true)$:
    a. Если формула выполнима, исходная программа некорректна и π — контрпример.
    b. Иначе исполняется шаг 5.
5. Уточняется текущее множество предикатов: во множество добавляются предикаты, построенные на основе цепочки операторов $op(\pi)$.

    Осуществляется переход к шагу 1.

Поясним, как уточняется множество предикатов, по которым строится модель. Для логики предикатов первого порядка известна следующая теорема, доказанная в 1957 г. Вильямом

[80] Здесь в качестве предусловия используется $true$, поскольку считается, что начальное состояние может быть любым. Если начальное состояние известно, берется регион, содержащий все истинные в этом состоянии предикаты.

Крейгом (William Craig, 1918-2016) [Cra57]. Если общезначима импликация $\varphi \to \psi$, то существует формула ρ, все свободные переменные которой (также как и все неинтерпретируемые функциональные и предикатные символы) являются общими для φ и ψ, такая что общезначимы $\varphi \to \rho$ и $\rho \to \psi$. Формула ρ называется *интерполянтом Крейга* или просто *интерполянтом*[81].

Для удобства далее под $sp(P, \varphi)$ понимается сильнейшее постусловие, выраженное в терминах версий переменных, т.е. $\big(\varphi[\text{оригиналы} \coloneqq \text{версии}_{\text{н}}] \wedge ssa(P)\big)$.

Пусть $op(\pi) = \{op_i\}_{i=1}^{n}$, где π — контрпример модели, но не исходной программы (шаг 4.b). То, что π не является контрпримером исходной программы, выражается как невыполнимость $sp(op(\pi), true)$. Постусловие цепочки операторов представимо в виде конъюнкции $(\psi_1 \wedge \ldots \wedge \psi_n)$, где для каждого $i \in \{1, \ldots, n\}$ подформула ψ_i соответствует оператору op_i. Невыполнимость формулы $(\psi_1 \wedge \ldots \wedge \psi_n)$ равносильна общезначимости ее отрицания, т.е. импликации

$$\psi_1 \to \neg(\psi_2 \wedge \ldots \wedge \psi_n).$$

Из теоремы Крейга следует, что для этой импликации существует интерполянт ρ_1, причем ρ_1 содержит только общие переменные формул ψ_1 и $(\psi_2 \wedge \ldots \wedge \psi_n)$. В частности это означает, что для каждой переменной в ρ_1 входит не более одной SSA-версии этой переменной. Обозначим через p_1' предикат, полученный из ρ_1 заменой SSA-версий на оригинальные переменные. Таким образом, $\rho_1 = p_1'\theta_1$, где θ_1 — подстановка, заменяющая переменные их версиями, используемыми в ρ_1.

Нетрудно показать, что если множество предикатов, по которым строится абстракция, содержит предикат p_1', то p_1' содержится в регионе r_1', в который переходит регион $r_0' = true$

---

[81] Мы не рассматриваем здесь процедуру построения интерполянта двух заданных формул. Желающие могут обратиться, например, к книге [Har09]. Отметим лишь, что интерполянт можно построить путем резолютивного вывода опровержения формулы $\varphi \wedge \neg\psi$, отрицания $\varphi \to \psi$ (см. лекцию 8). Каждой клаузе C клаузальной формы φ сопоставляется формула $\rho(C) = g(C)$ — подклауза C (возможно, пустая), содержащая все общие переменные φ и ψ и не содержащая других переменных. Каждой клаузе C клаузальной формы $\neg\psi$ сопоставляется $\rho(C) = true$ (для общих клауз допустимы оба варианта). Если резолюция C_1 и C_2 осуществляется по внутренней переменной φ, то $\rho\big(Res(C_1, C_2)\big) = \rho(C_1) \vee \rho(C_2)$, иначе (резолюция осуществляется по общей переменной или локальной переменной ψ) $\rho\big(Res(C_1, C_2)\big) = \rho(C_1) \wedge \rho(C_2)$.

при исполнении оператора op_1, т.е. $r_1' = (p_1' \wedge r_1)$, где r_1 — некоторая конъюнкция предикатов (возможно, пустая). В самом деле, теорема Крейга утверждает общезначимость импликации $\psi_1 \to \rho_1$, что, принимая в расчет равенства $\psi_1 = sp(op_1, true)$ и $\rho_1 = p_1'\theta_1$, эквивалентно невыполнимости формулы $sp(op_1, true) \wedge \neg p_1'\theta_1$. В свою очередь, это означает, что предикат p_1' входит в регион r_1' (см. правило вычисления регионов).

Рассмотрим оставшиеся операторы: $op\left(\pi^{(2)}\right) = \{op_i\}_{i=2}^{n}$. Условие $sp\left(op\left(\pi^{(2)}\right), r_1'\right)$ невыполнимо, поскольку

$$sp\left(op\left(\pi^{(2)}\right), r_1'\right) = r_1'\theta_1 \wedge sp\left(op\left(\pi^{(2)}\right), true\right) =$$
$$(p_1' \wedge r_1)\theta_1 \wedge (\psi_2 \wedge \ldots \wedge \psi_n) = \rho_1 \wedge (\psi_2 \wedge \ldots \wedge \psi_n) \wedge r_1\theta_1,$$

а теорема Крейга утверждает, что импликация $\rho_1 \to \neg(\psi_2 \wedge \ldots \wedge \psi_n)$ общезначима, или, что равносильно, конъюнкция $\rho_1 \wedge (\psi_2 \wedge \ldots \wedge \psi_n)$ невыполнима.

Применив к $sp\left(op\left(\pi^{(2)}\right), r_1'\right)$ тот же подход, что и к $sp(op(\pi), true)$, получим предикат p_2'. Обозначим через r_2' регион, в который переходит r_1' при исполнении op_2. Если r_2' не есть $false$, процесс продолжается.

Итак, в цикле ($k = 1, 2, \ldots$) для очередной формулы

$$sp\left(op\left(\pi^{(k)}\right), r_{k-1}'\right) = r_{k-1}'\theta_{k-1} \wedge (\psi_k \wedge \ldots \wedge \psi_n)$$

строится интерполянт ρ_k и предикат p_k'. Процесс завершается, когда регион r_k' становится равным $false$. Рано или поздно (не позднее $k = n$) это случится. Действительно, формула $sp\left(op\left(\pi^{(n)}\right), r_{n-1}'\right) = sp(op_n, r_{n-1}')$ невыполнима (как и все предшествующие формулы этого вида). По правилу вычисления региона имеем $r_n' = false$.

Уточнение множества предикатов состоит в добавлении в это множество построенных предикатов $p_1', \ldots, p_k'$. Для уточненной абстракции вычисление π уже не является контрпримером.

11.4. Вопросы и упражнения

1. При верификации моделей программ могут возникать ошибки первого и второго рода. Дайте определение этим типам ошибок. В чем причины их возникновения?
2. Какой уровень абстракции модели программы можно считать адекватным?

3. Опишите общую схему метода адаптивного уточнения абстракции по контрпримерам (CEGAR, Counter Example Guided Abstraction Refinement).

4. При построении асинхронной композиции процессов возникает явление, известное как комбинаторный взрыв числа состояний. Предложите способы борьбы с этим явлением.

5. Постройте структуру Крипке для программы, реализующей алгоритм Евклида (программы P из лекции 1), для некоторых фиксированных значений входных переменных, например, 6 и 9.

6. Постройте структуру Крипке асинхронной композиции двух процессов P_0 и P_1 (определенных ниже) и проверьте свойство взаимного исключения [Кар10]:

```
while true do
  NCSᵢ: /* некритическая секция */
  SETᵢ: (yᵢ, s) := (1, i); /* атомарное присваивание */
  TSTᵢ: while (y₁₋ᵢ ≠ 0) ∧ (s = i) do skip end;
  CRSᵢ: /* критическая секция */
  RSTᵢ: yᵢ := 0
end
```

7. Постройте множество всех возможных регионов для предикатов $x \geq 5$ и $x \leq 10$.

8. Используя декартову предикатную абстракцию, постройте модель для следующей программы [Ман13]:

```
if x ≥ 5 then
  if x ≤ 10 then
    x = x + 1;
    if ¬(x ≥ 5) then
error: skip
    end
  end
end
```

   В качестве предикатов используйте $x \geq 5$ и $x \leq 10$.

9. Предложите множество предикатов и постройте декартову предикатную абстракцию программы, реализующей алгоритм Евклида (программы P из лекции 1). Представьте абстрактную модель в форме структуры Крипке (состояния структуры — абстрактные конфигурации, пометки — множества истинных предикатов).

10. Переведите следующую while-программу (уточнение примера, приведенного в лекции 7) на язык PROMELA и с помощью инструмента SPIN попытайтесь проверить, достижима ли в ней точка $error$:

```
i = 2;
j = 0;
while 2 * j ≤ i - 2 do
  if i ≤ 3 * j then
    i := i + 4
  else
    i := i + 2;
    j := j + 1
  end
end;
error: skip
```

Предложите множество предикатов и постройте декартову предикатную абстракцию этой программы. Представьте абстрактную модель на языке PROMELA и проверьте ее с помощью SPIN.

11. Напишите программы (на удобном вам языке программирования), реализующие поиск в ширину и в глубину в ориентированном графе с выделенной начальной вершиной.

Лекция 12. Автоматы Бюхи и ω-регулярные языки

Поскольку время бесконечно, до настоящего момента уже протекла бесконечность, т.е. всякое возможное развитие должно уже было осуществиться. Следовательно, наблюдаемое развитие должно быть повторением.

Ф. Ницше.
Воля к власти

Рассматриваются ω-языки — языки с бесконечными словами — и один из способов их описания — автоматы Бюхи. Структурно автоматы Бюхи идентичны конечным автоматам, но отличаются от них условием распознавания слов. Показывается, как для языков, описываемых с помощью конечных автоматов и автоматов Бюхи, решаются задачи нахождения пересечения и проверки на пустоту. Если множество всех возможных траекторий системы и множество всех ошибочных траекторий системы могут быть представлены в форме автоматов Бюхи, то решение указанных задач дает ключ к проверке модели: если пересечение указанных множеств не пусто, модель ошибочна. Множество траекторий структуры Крипке тривиальным образом описывается автоматом Бюхи. Вопрос о возможности представления автоматами Бюхи формул LTL рассматривается в следующей лекции.

12.1. Конечные автоматы и регулярные языки

Условия корректности программы, выраженные в логике LTL, проверяются на вычислениях или траекториях структуры Крипке, моделирующей программу (см. лекцию 9). Говорят, что формула φ истинна на структуре Крипке M (обозначается: $M \vDash \varphi$), если φ истинна на любом вычислении M. Обратно — формула φ ложна на структуре M (обозначается: $M \nvDash \varphi$), если M имеет хотя бы одно вычисление π, такое что $\pi \nvDash \varphi$ (вычисления такого типа называются *контрпримерами*). В общих словах метод проверки моделей можно рассматривать как метод поиска контрпримеров для заданных структуры Крипке и формулы LTL.

Абстрагируемся на время от способа представления требований. Каким образом проверяется корректность программы при тестировании? Современная практика такова: по спецификации требований генерируется *оракул*, или *монитор*, — компонент тестовой системы, функцией которого является вынесение вердикта о корректности/некорректности поведения программы. Оракул перехватывает стимулы, подаваемые на программу, и выдаваемые реакции и проверяет допустимость наблюдаемого поведения: при обнаружении несоответствия спецификации (например, при получении неправильной реакции на некоторый

стимул), возвращается отрицательный вердикт; если же за все время тестирования ничего «плохого» не произошло, результатом является положительный вердикт.

Адаптируем этот подход к проверке моделей. Поскольку модель программы имеет форму системы переходов (автомата), можно предположить, что и оракул должен иметь подобный вид. Оракул, представленный в форме автомата, будем называть *контрольным автоматом*; он функционирует *параллельно* и *синхронно* с проверяемой моделью: если траектория модели не соответствует требованиям спецификации, контрольный автомат переходит в состояние «ошибка» и остается в нем навсегда [Кар10]. Отметим, что в языке PROMELA контрольные автоматы описываются с помощью **never**-блоков (см. лекцию 10).

Пусть имеется контрольный автомат. Как его использовать для проверки модели? Перебрать все возможные траектории и проверить каждую из них невозможно — траекторий бесконечно много, и каждая из них бесконечна. Можно ограничиться проверкой всех конечных префиксов, но это не гарантирует полноты верификации.

Чтобы решить поставленную задачу следует учесть, что модель конечна, и порождаемое ей множество траекторий *регулярно*; то же справедливо и в отношении контрольного автомата. Здесь есть прямая аналогия с распознаванием регулярных языков автоматами: контрольный автомат — это распознаватель языка, а траектория — слово, принадлежность которого языку (множеству правильных или, наоборот, неправильных поведений) нужно проверить [Кар03]. Принципиальная разница лишь в том, что траектории бесконечны.

Формализуем задачу проверки модели в терминах теории языков. Пусть $\mathcal{L}_M$ — язык модели M (множество всех возможных траекторий M), а $\mathcal{L}_{\neg\varphi}$ — язык, распознаваемый контрольным автоматом (множество всех траекторий, на которых ложна формула φ). Модель M удовлетворяет спецификации φ тогда и только тогда, когда

$$\mathcal{L}_M \cap \mathcal{L}_{\neg\varphi} = \emptyset.$$

Встает вопрос — как представлять языки $\mathcal{L}_M$ и $\mathcal{L}_{\neg\varphi}$ и проверять их пересечение на пустоту? Ситуация усложняется тем, что словами этих языков являются бесконечные траектории. Ответ на возникший вопрос дает теория ω-автоматов и ω-регулярных языков. Прежде чем переходить к этой теме рассмотрим случай «классических» регулярных языков.

12.1.1. Описание языков конечными автоматами

Вспомним основные понятия. *Языком* над конечным и непустым алфавитом Σ называется любое, возможно, бесконечное, множество конечных *цепочек* (*слов*), составленных из *символов* (*букв*) Σ. Множество всех слов в алфавите Σ обозначается Σ^*. Итак, $\mathcal{L}$ — язык над алфавитом Σ, если $\mathcal{L} \subseteq \Sigma^*$.

Известны формализмы, позволяющих описывать бесконечные языки конечным образом. Прежде всего это формальные грамматики и конечные автоматы [Сер12]. Языки, задаваемые конечными автоматами, называются *автоматными языками* [Кар03]. Так как по выразительной силе конечные автоматы эквивалентны регулярным выражениям и регулярным грамматикам, в дальнейшем мы будем считать термины «автоматный язык» и «регулярный язык» синонимами.

Детерминированным конечным автоматом (*ДКА*) над алфавитом Σ называется пятерка $\langle S, \Sigma, s_0, \delta, \mathcal{F} \rangle$, где S — множество *состояний*, Σ — *входной алфавит*, $s_0 \in S$ — *начальное состояние*, $\delta: S \times \Sigma \to S$ — *функция переходов*[82], $\mathcal{F} \subseteq S$ — множество *допускающих состояний*. Предполагается, что все множества, указанные в этом определении, конечны.

Недетерминированным конечным автоматом (*НКА*) называется пятерка $\langle S, \Sigma, S_0, \delta, \mathcal{F} \rangle$, где $S_0 \subseteq S$ — непустое множество *начальных состояний*, $\delta: S \times \Sigma \to 2^S$ — *функция переходов*[83]. Смысл остальных элементов точно такой же, как в ДКА.

Далее, если не оговорено противное, мы будем рассматривать НКА, называя их *конечными автоматами* или просто *автоматами*.

Говорят, что автомат A *допускает* или *распознает* слово (траекторию конечной длины) $w = x_0 x_1 \dots x_{n-1} \in \Sigma^*$, если w «переводит» A из одного из начальных состояний в одно из допускающих состояний. Формально это определяется следующим образом: существует последовательность состояний (вычисление) $\pi = \{s_i\}_{i=0}^{n}$, такая что

---

[82] Функция переходов, вообще говоря, является частичной, т.е. она не обязана быть определенной для всех комбинаций состояний и символов.

[83] В некоторых работах по теории формальных языков в НКА допускаются переходы по *пустому символу* $\varepsilon \notin \Sigma$: $\delta: S \times (\Sigma \cup \{\varepsilon\}) \to 2^S$. Заметим, что по НКА с ε-переходами можно построить эквивалентный ему НКА без ε-переходов; более того, по НКА можно построить эквивалентный ему ДКА [Сер12].

- $s_0 \in S_0$;
- $s_i \in \delta(s_{i-1}, x_{i-1})$ для всех $1 \leq i \leq n$;
- $s_n \in \mathcal{F}$.

Множество всех слов, допускаемых автоматом A, называется *языком*, *допускаемым* A, и обозначается $\mathcal{L}_A$. Язык называется *регулярным*, если существует допускающий его автомат.

Пример 12.1

На рис. 12.1 изображены два автомата A и B над алфавитом $\{a, b\}$. Начальные состояния помечены ведущими в них стрелками (из «ниоткуда»), а допускающие состояния выделены жирными линиями. Автомат A допускает все слова, заканчивающиеся символом a, автомат B — все слова четной длины, в которых символы a и b чередуются.

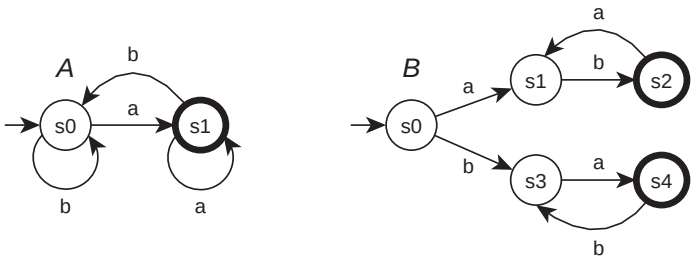


Рисунок 12.1. Примеры конечных автоматов

□

12.1.2. Синхронная композиция конечных автоматов

Синхронной композицией автоматов $A = \langle S_A, \Sigma, S_{A0}, \delta_A, \mathcal{F}_A \rangle$ и $B = \langle S_B, \Sigma, S_{B0}, \delta_B, \mathcal{F}_B \rangle$ над общим алфавитом Σ называется автомат

$$A \otimes B = \langle S_A \times S_B, \Sigma, S_{A0} \times S_{B0}, \delta_{AB}, \mathcal{F}_A \times \mathcal{F}_B \rangle,$$

функция переходов которого определяется следующим равенством:

$$\delta_{AB}\big((s_A, s_B), x\big) = \delta_A(s_A, x) \times \delta_B(s_B, x).$$

Говоря неформально, синхронная композиция двух автоматов — это автомат, моделирующий их параллельную и синхронную работу: входной символ подается на оба автомата;

каждый из них переходит в некоторое состояние в соответствии со своей функцией переходов. Начальными состояниями синхронной композиции являются пары, составленные из начальных состояний композируемых автоматов; допускающими состояниями — пары допускающих состояний.

Пример 12.2

На рис. 12.2 изображена синхронная композиция автоматов A и B из предыдущего примера. В автомате $A \otimes B$ имеется одно начальное состояние (s_0, s_0) и одно допускающее состояние (s_1, s_4).

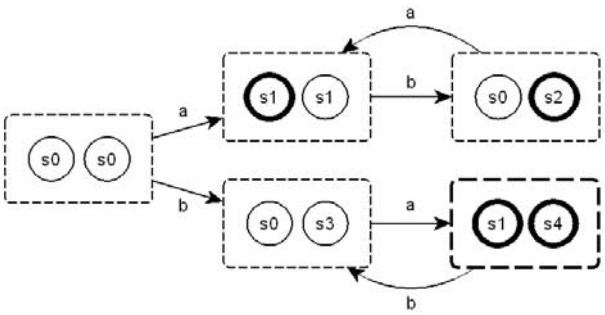


Рисунок 12.2. Синхронная композиция конечных автоматов

Нетрудно видеть, что автомат $A \otimes B$ допускает слова, имеющие вид $(ba)^n$, где $n \geq 1$, т.е. слова, которые допускаются как автоматом A, так и автоматом B.

□

Наблюдение, сделанное в примере, имеет универсальный характер. Оказывается, что класс автоматных языков замкнут относительно операции пересечения — для любой пары конечных автоматов существует конечный автомат, распознающий пересечение их языков. Более того, пересечение языков, распознаваемых автоматами A и B, совпадает с языком, распознаваемым синхронной композицией $A \otimes B$ [Кар10]:

$$\mathcal{L}_A \cap \mathcal{L}_B = \mathcal{L}_{A \otimes B}.$$

12.1.3. Проверка языка, допускаемого конечным автоматом, на пустоту

Вернемся к исходной задаче: для заданной модели M и спецификации φ требуется проверить на пустоту пересечение $\mathcal{L}_M$ и $\mathcal{L}_{\neg\varphi}$. Если модель и оракул являются конечными автоматами (обозначим оракул как $C_{\neg\varphi}$), то, как мы выяснили, это пересечение описывается синхронной композицией $M \otimes C_{\neg\varphi}$. Осталось проверить, пуст ли язык $\mathcal{L}_{M\otimes C_{\neg\varphi}}$.

Проверка автоматного языка на пустоту осуществляется очевидным образом — путем поиска достижимого допускающего состояния. Если в автомате A хотя бы одно допускающее состояние достижимо хотя бы из одного начального состояния, язык $\mathcal{L}_A$ не пуст. В противном случае, т.е. когда ни одно допускающее состояние не достижимо ни из одного начального состояния, язык $\mathcal{L}_A$ пуст. Таким образом, проверка автоматного языка на пустоту сводится к проверке достижимости состояния с заданными свойствами. Эта задача решается поиском в ширину или в глубину (см. лекцию 11).

Пример 12.3

Язык, допускаемый конечным автоматом $A \otimes B$ из рассмотренного ранее примера, не пуст. В этом нетрудно убедиться, построив путь из единственного начального состояния (s_0, s_0) в единственное допускающее состояние (s_1, s_4): $\{(s_0, s_0), (s_0, s_3), (s_1, s_4)\}$.

□

Обратите внимание, что процесс проверки языка на пустоту можно организовать так, что в случае если язык не пуст, предъявляется контрпример — путь из начального состояния в допускающее. Для автомата $M \otimes C_{\neg\varphi}$ этот путь является вычислением модели M, в котором нарушается свойство φ.

12.2. Автоматы Бюхи и ω-регулярные языки

Классическая теория формальных языков построена для слов конечной длины. Вычисления реагирующих систем и моделирующих их структур Крипке являются *бесконечными* (потен-

циально такие системы могут работать бесконечно долго). Соответственно, для их верификации необходимы инструменты, позволяющие оперировать с бесконечными цепочками и их множествами: ω-*словами* и ω-*языками* соответственно[84] [Кар10].

Множество всех ω-слов над алфавитом Σ обозначается Σ^{ω}. $\mathcal{L}$ — ω-язык над алфавитом Σ, если $\mathcal{L} \subseteq \Sigma^{\omega}$.

Аналогом конечных автоматов для ω-языков являются так называемые ω-*автоматы*. К ним, в частности, относятся автоматы Бюхи[85], предложенные в 1960 г. Юлиусом Бюхи (Julius Richard Büchi, 1924-1984) [Buc60].

12.2.1. Описание ω-языков автоматами Бюхи

Структурно автоматы Бюхи абсолютно идентичны обычным конечным автоматам. *Детерминированным автоматом Бюхи* называется пятерка $\langle S, \Sigma, s_0, \delta, \mathcal{F} \rangle$; *недетерминированным* — $\langle S, \Sigma, S_0, \delta, \mathcal{F} \rangle$ (смысл обозначений такой же, как в ДКА и НКА). Далее, если не оговорено противное, мы будем рассматривать недетерминированные автоматы Бюхи, называя их *автоматами Бюхи* или просто *автоматами*.

Отличие автоматов Бюхи от конечных автоматов состоит в *условии допустимости* слова. Обозначим через $\inf(\pi)$ множество всех элементов последовательности π, встречающихся в ней бесконечное число раз. Говорят, что ω-слово $w = x_0 x_1 \ldots \in \Sigma^{\omega}$ *допускается* автоматом $\langle S, \Sigma, S_0, \delta, \mathcal{F} \rangle$, если в нем возможно вычисление с траекторией w, т.е. существует последовательность состояний $\pi = \{s_i\}_{i=0}^{\infty}$, такая что

- $s_0 \in S_0$;
- $s_i \in \delta(s_{i-1}, x_{i-1})$ для всех $1 \leq i \leq n$;
- $\inf(\pi) \cap \mathcal{F} \neq \emptyset$:
  хотя бы одно допускающее состояние встречается бесконечное число раз.

---

[84] Вместо терминов «ω-слово» и «ω-язык» также используются термины «сверхслово» и «сверхязык».

[85] Другими типами ω-автоматов являются, например, *автоматы Мюллера* (David Eugene Muller, 1924-2008), 1963 г. [Mul63] и *автоматы Рабина* (Michael Oser Rabin, род. в 1931 г.), 1969 г. [Rab69]. Все типы ω-автоматов имеют одинаковую структуру, совпадающую со структурой конечных автоматов, а различаются только условием допустимости ω-слова. Все эти формализмы имеют одинаковую выразительную силу [Tho96].

Множество всех ω-слов, допускаемых автоматом A, называется ω-*языком*, *допускаемым* A, и обозначается $\mathcal{L}_A$; ω-язык называется ω-*регулярным языком*, или *регулярным* ω-*языком* (что правильнее), если существует допускающий его *недетерминированный* автомат Бюхи. В отличие от конечных автоматов не для всякого недетерминированного автомата Бюхи существует эквивалентный ему детерминированный автомат[86], поэтому в приведенном определении условие недетерминированности является существенным.

Пример 12.4

Рассмотрим автоматы A и B, изображенные на рис. 12.1, однако на этот раз будем считать, что это не конечные автоматы, а автоматы Бюхи. Какие языки распознают эти автоматы? Язык, допускаемый A, — все ω-слова в алфавите $\{a, b\}$, не заканчивающиеся бесконечной последовательностью b^ω. Язык, допускаемый B, состоит всего из двух цепочек: $(ab)^\omega$ и $(ba)^\omega$.

□

12.2.2. Синхронная композиция автоматов Бюхи

Класс регулярных языков замкнут относительно операции пересечения. То же справедливо и в отношении ω-регулярных языков, однако здесь есть небольшая тонкость. Пересечение языков автоматов A и B распознается синхронной композицией $A \otimes B$; допускающими состояниями являются пары, состоящие из допускающих состояний A и B. Пересечение языков $\mathcal{L}_A$ и $\mathcal{L}_B$, где A и B — автоматы Бюхи, также распознается синхронной композицией $A \otimes B$ (определение будет дано ниже). Цепочка допускается $A \otimes B$, если она допускается как автоматом A, так и автоматом B, т.е. существует вычисление, которое бесконечное число раз проходит через допускающие состояния автомата A и бесконечное число раз проходит через допускающие состояния автомата B. Заметим, что от вычисления не требуется, чтобы оно проходило через допускающие состояния A и B одновременно.

[86] Для ω-автоматов Мюллера и Рабина это не так — классы детерминированных и недетерминированных автоматов имеют одинаковую выразительную силу [Tho96].

Введем более общий тип автоматов. *Обобщенным автоматом Бюхи* называется пятерка $\langle S, \Sigma, S_0, \delta, \boldsymbol{\mathcal{F}} \rangle$, где $\boldsymbol{\mathcal{F}} = \{\mathcal{F}_1, \dots, \mathcal{F}_k\}$ — множество множеств допускающих состояний. Говорят, что ω-слово $w = x_0 x_1 \dots \in \Sigma^\omega$ *допускается* обобщенным автоматом Бюхи, если существует последовательность состояний $\pi = \{s_i\}_{i=0}^{\infty}$, такая что

- $s_0 \in S_0$;
- $s_i \in \delta(s_{i-1}, x_{i-1})$ для всех $1 \le i \le n$;
- $\inf(\pi) \cap \mathcal{F}_j \neq \emptyset$ для всех $1 \le j \le k$.

Обычный автомат Бюхи можно считать обобщенным автоматом Бюхи, у которого множество множеств допускающих состояний состоит из одного множества.

Синхронной композицией обобщенных автоматов Бюхи

$$A = \langle S_A, \Sigma, S_{A0}, \delta_A, \boldsymbol{\mathcal{F}}_A = \{\mathcal{F}_{A1}, \dots, \mathcal{F}_{Ak}\} \rangle \quad \text{и} \quad B = \langle S_B, \Sigma, S_{B0}, \delta_B, \boldsymbol{\mathcal{F}}_B = \{\mathcal{F}_{B1}, \dots, \mathcal{F}_{Bm}\} \rangle$$

над общим алфавитом Σ называется обобщенный автомат Бюхи

$$A \otimes B = \langle S_A \times S_B, \Sigma, S_{A0} \times S_{B0}, \delta_{AB}, \boldsymbol{\mathcal{F}}_{AB} \rangle,$$

где $\boldsymbol{\mathcal{F}}_{AB} = \{(\mathcal{F}_{A1} \times S_B), \dots, (\mathcal{F}_{Ak} \times S_B), (S_A \times \mathcal{F}_{B1}), \dots, (S_A \times \mathcal{F}_{Bm})\}$.

Справедливо следующее утверждение: для любого обобщенного автомата Бюхи существует эквивалентный ему обычный автомат Бюхи.

Пусть $G = \langle S, \Sigma, S_0, \delta, \boldsymbol{\mathcal{F}} = \{\mathcal{F}_1, \dots, \mathcal{F}_k\} \rangle$ — обобщенный автомат, в котором $k \ge 1$ (очевидно, что если $k = 0$, то, заменив пустое множество $\boldsymbol{\mathcal{F}}$ на $\{S\}$, получим эквивалентный обобщенный автомат Бюхи). Построим k «копий» автомата G. Множества начальных и допускающих состояний имеются только в 1-ой копии: S_0 и $\mathcal{F}_1$ соответственно. Из состояний множества $\mathcal{F}_i$, относящегося к i-ой копии, где $1 \le i < k$, переходы ведут в соответствующие состояния $(i+1)$-ой копии: если в исходном автомате $s' = \delta(s, x)$, то в конструируемом автомате $(s', i+1) = \delta\big((s, i), x\big)$, где индекс в паре с состоянием — это номер копии; для k-ой копии переходы ведут в 1-ую копию (см. рис. 12.3).

Легко убедиться, что построенный таким образом автомат (обозначим его B) эквивалентен обобщенному автомату G: допускающее состояние автомата B, посещаемое бесконечное число раз, существует тогда и только тогда, когда в каждом из допускающих множеств $\mathcal{F}_i$ есть состояние, которое посещается бесконечное число раз.

Формально автомат B определяется следующим образом:

$$B = \langle S \times \{1, \dots, k\}, \Sigma, S_0 \times \{1\}, \delta_B, \mathcal{F}_1 \times \{1\} \rangle,$$

где функция переходов δ_B имеет следующий вид:

- $\delta_B\big((s,i),x\big) = \{(s',i) \mid s' \in \delta(s,x)\}$, если $s \notin \mathcal{F}_i$;
- $\delta_B\big((s,i),x\big) = \{(s',(i \,\%\, k)+1) \mid s' \in \delta(s,x)\}$, иначе.

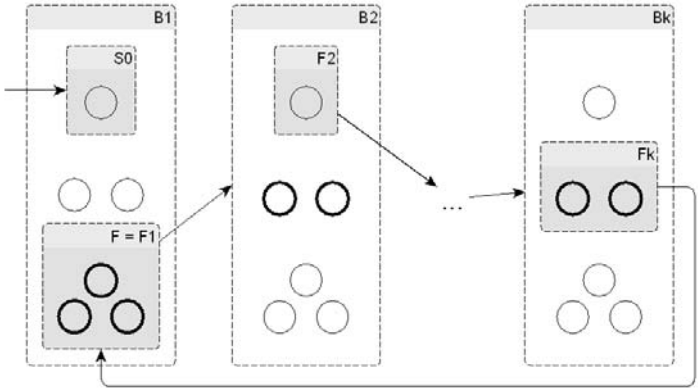


Рисунок 12.3 Построение автомата Бюхи по обобщенному автомату Бюхи

12.2.3. Проверка языка, допускаемого автоматом Бюхи, на пустоту

Проверка языка, допускаемого автоматом Бюхи, на пустоту основана на следующем утверждении: язык не пуст тогда и только тогда, когда в графе состояний автомата есть цикл, проходящий через некоторое допускающее состояние и достижимый из одного из начальных состояний. Такой цикл будем называть *допускающим*.

В самом деле, пусть π_C — допускающий цикл, а s_F — допускающее состояние, через которое он проходит. Тогда вычисление $\pi = \pi' \cdot (\pi_C)^\omega$, где π' — путь из одного из начальных состояний в состояние s_F, включает бесконечное число вхождений s_F. Соответствующее ω-слово допускается автоматом, а значит, его язык не пуст. С другой стороны, если существует вычисление π, в котором некоторое допускающее состояние s_F встречается бесконечное число раз, то, очевидно, существует и допускающий цикл — в таком качестве можно взять путь между двумя первыми вхождениями s_F в π.

Есть два основных подхода к поиску допускающих циклов. Первый из них основан на построении компонент сильной связности. *Компонентой сильной связности* ориентированного графа называется максимальный по включению вершин подграф, в котором любая пара вершин соединена путем. Известны эффективные алгоритмы построения множества достижимых компонент сильной связности. Один из них — алгоритм Тарьяна (Robert Endre Tarjan, род. в 1948 г.), 1972 г. — основан на поиске в глубину и имеет вычислительную сложность $O(m)$, где m — число дуг графа [Tar72], [Кас03]. Сформулированное выше условие пустоты языка можно переформулировать эквивалентным образом: язык, допускаемый автоматом Бюхи, пуст тогда и только тогда, когда ни одна достижимая нетривиальная (отличная от одиночной вершины без петель) компонента сильной связности не содержит допускающих состояний.

Второй подход основан на *вложенном поиске в глубину* (*nested depth-first search*). Именно этот метод используется в инструменте SPIN [Hol03]. Его главное преимущество по сравнению с алгоритмом Тарьяна —возможность верификации *на лету* (*on the fly*): допускающие циклы ищутся не после построения всего пространства состояний, а в процессе обхода. Метод оперирует с парами (s, i), где s — состояние автомата, а i — признак вложенного поиска: 0 — основной поиск; 1 — вложенный. Каждое состояние s хранится в одном экземпляре, но вместе с ним хранятся два флага $toggle_0$ и $toggle_1$: $toggle_i = true$, если пространство состояний содержит (s, i). Таким образом, метод требует дополнительно лишь два бита на состояние по сравнению с обычным поиском в глубину (см. лекцию 11).

Сам метод состоит в следующем. На фазе основного поиска, когда пройдены все последователи текущего состояния (перед тем, как вытолкнуть это состояние из стека), проверяется, является ли оно допускающим: если да, запускается вложенный поиск, осуществляющий проверку достижимости состояния из самого себя. Признаком достижимости является попадание в состояние, присутствующее в стеке основного поиска. Еще раз подчеркнем, что вложенный поиск — это тот же поиск в глубину, с той лишь разницей, что пространство состояний заполняется парами вида $(s, 1)$, а не $(s, 0)$. Алгоритм гарантирует, что при наличии допускающих циклов, по крайней мере один из них будет найден.

В таблице 12.1 представлен псевдокод алгоритма, в котором для наглядности предполагается, что имеется только одно начальное состояние s_0. Множество S содержит посещенные

состояния — пары вида (s, i). Предикат $accepting$ проверяет, является ли состояние допускающим; функция $choose$ выбирает произвольный элемент непустого множества. Обозначим через $succ(s)$ множество всех непосредственных последователей состояния s. Пусть $succ(s, i) = \{(s', i) \mid s' \in succ(s)\}$.

Таблица 12.1. Вложенный поиск в глубину

Основной поиск	Вложенный поиск: вызов $ndfs(s_c)$

`S := {(s₀,0)};`
`dfs_stack := {s₀};`
`while dfs_stack ≠ ∅ do`
`  s := dfs_stack.top();`
`  if (succ(s,0) \ S) = ∅ then`
`  if accepting(s) then`
`  if ndfs(s) then`
`  return true /* цикл */`
`  end`
`  end;`
`  dfs_stack.pop()`
`  else`
`  (s',0) := choose(succ(s,0) \ S);`
`  S := S ∪ {(s',0)};`
`  dfs_stack.push(s')`
`  end`
`end;`
`return false /* цикл не найден */`
	`S := S ∪ {(s꜀,1)};`
	`ndfs_stack := {s꜀};`
	`while ndfs_stack ≠ ∅ do`
	`  s := ndfs_stack.top();`
	`  if succ(s) ∩ dfs_stack ≠ ∅ then`
	`  return true /* цикл */`
	`  end;`
	`  if (succ(s,1) \ S) = ∅ then`
	`  ndfs_stack.pop()`
	`  else`
	`  (s',1) := choose(succ(s,1) \ S);`
	`  S := S ∪ {(s',1)};`
	`  ndfs_stack.push(s')`
	`  end`
	`end;`
	`return false /* цикл не найден */`

12.3. Теоретико-автоматный подход к проверке моделей

Траектории структуры Крипке над множеством элементарных высказываний AP образуют ω-язык над алфавитом 2^{AP}. Этот язык является ω-регулярным — для него существует распознающий его автомат Бюхи. Для заданной структуры Крипке $M = \langle S, S_0, R, L\rangle$ автомат Бюхи B_M, допускающий все возможные траектории модели M (и только их), имеет вид $B_M = \langle S, 2^{AP}, S_0, \delta, S\rangle$, где функция переходов δ определена только на парах вида $\big(s, L(s)\big)$:

$$\delta\big(s, L(s)\big) = \{s' \mid (s, s') \in R\}.$$

Заметим, что в автомате Бюхи B_M все состояния являются допускающими.

Пример 12.5

На рис. 12.4 приведена структура Крипке M и соответствующий автомат Бюхи B [Кар10]. Начальные состояния помечены ведущими в них стрелками (из «ниоткуда»); допускающие состояния выделены жирными линиями.

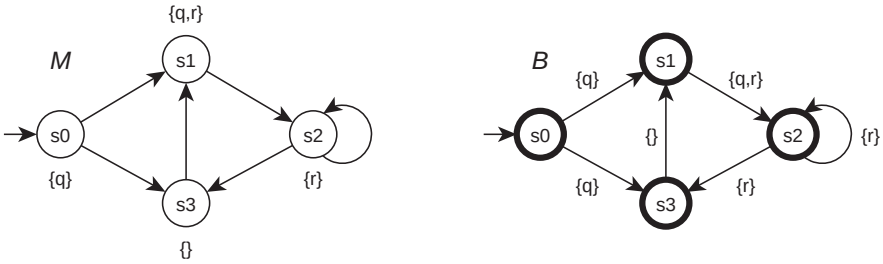


Рисунок 12.4. Структура Крипке и соответствующий автомат Бюхи

□

Более сложной задачей является построение автомата Бюхи по формуле LTL: требуется по формуле φ построить автомат B_φ, допускающий те и только те траектории, на которых формула φ истинна (если формула описывает условия корректности модели, то автомат Бюхи, играющий роль контрольного автомата, строится для ее отрицания). Алгоритм построения автомата Бюхи по формуле LTL рассматривается в следующей лекции, а пока мы ограничимся несколькими примерами.

Пример 12.6

Рассмотрим следующие формулы (формулы такого типа часто используются на практике):

- $\mathbf{G}p$ — свойство *безопасности* (*safety*):
  нечто плохое никогда не произойдет;
- $\mathbf{F}p$ — свойство *живости* (*liveness*):
  нечто хорошее когда-нибудь произойдет;
- $\mathbf{GF}p$ — свойство *справедливости* (*fairness*):
  нечто должно возникать периодически;
- $\mathbf{FG}p$ — свойство *стабилизации* (*stabilization*):
  нечто должно стать постоянным.

Построим для этих формул автоматы Бюхи, распознающие траектории, на которых формулы истинны, в предположении, что $AP = \{p\}$ и, соответственно, $\Sigma = 2^{AP} = \{\emptyset, \{p\}\}$.

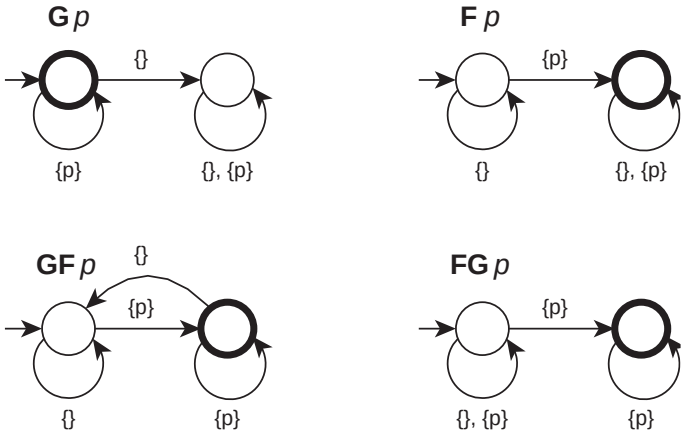


Рисунок 12.5. Автоматы Бюхи для некоторых формул LTL

□

Теперь мы готовы рассмотреть общую схему проверки моделей для логики LTL. Описываемый подход называется *теоретико-автоматным подходом*, он был предложен в 1985 г. Мойше Варди (Moshe Vardi, род. в 1954 г.) и Пьером Вольпером (Pierre Wolper, род. в 1955 г.) [Var86]. Итак, мы имеем модель M и спецификацию φ — формулу LTL над тем же множеством элементарных высказываний, что и модель M. Проверка модели M на соответствие спецификации φ, т.е. проверка того, что структура Крипке M является моделью формулы φ, осуществляется следующим образом:

1. строится автомат B_M, распознающий все траектории модели M (и только их);
2. строится автомат $B_{\neg\varphi}$, распознающий все траектории, на которых истинна формула $\neg\varphi$ (и только их);
3. строится синхронная композиция $B_M \otimes B_{\neg\varphi}$;
4. в автомате $B_M \otimes B_{\neg\varphi}$ проверяется наличие допускающего цикла:
    a. если допускающий цикл существует, модель ошибочна ($M \not\models \varphi$):
        i. контрпример — путь до цикла плюс бесконечное повторение цикла;
    b. в противном случае модель корректна ($M \models \varphi$).

Вообще говоря, автомат $B_M \otimes B_{\neg\varphi}$ является обобщенным автоматом Бюхи, однако из-за того, что в B_M все состояния являются допускающими, его можно воспринимать как обычный автомат Бюхи, в котором множество допускающих состояний совпадает с множеством

допускающих состояний $B_{\neg\varphi}$: состояние $(s_M, s_{\neg\varphi})$ является допускающим тогда и только тогда, когда допускающим является $s_{\neg\varphi}$.

Вычислительная сложность проверки моделей для LTL оценивается величиной $O(|S| \cdot 2^{|\varphi|})$, где $|S|$ — число состояний в модели M, а $|\varphi|$ – число символов в формуле φ. Происхождение множителя $2^{|\varphi|}$ станет понятным из следующей лекции — это оценка числа состояний в автомате $B_{\neg\varphi}$.

12.4. Вопросы и упражнения

1. Докажите, что для конечных автоматов имеет место равенство $L_A \cap L_B = L_{A \otimes B}$.
2. В чем отличие автоматов Бюхи от классических конечных автоматов?
3. Определите автомат Бюхи и обобщенный автомат Бюхи. Опишите схему преобразования обобщенного автомата Бюхи к обычному.
4. Известно, что для любого НКА существует эквивалентный ему (допускающий тот же язык) ДКА. Верно ли следующее утверждение: для любого недетерминированного автомата Бюхи существует эквивалентный ему детерминированный автомат Бюхи?
5. Докажите, что класс ω-регулярных языков замкнут относительно операций объединения и дополнения.
6. Опишите способ построения по заданной структуре Крипке M (с конечным числом состояний и полным отношением переходов) автомата Бюхи, распознающего все возможные траектории M и только их.
7. Постройте минимальный конечный автомат над алфавитом $\{a, b\}$, допускающий язык b^*a^*. Какой язык допускает этот автомат, если его рассматривать как автомат Бюхи [Кар10]? Выразите этот язык формулой LTL (считайте a и b взаимоисключающими высказываниями).
8. Постройте автомат Бюхи над алфавитом $\{a, b\}$, допускающий все цепочки с конечным числом вхождений символа a и бесконечным числом вхождений символа b. Выразите указанный язык формулой LTL.
9. Постройте автомат Бюхи над алфавитом $\{a, b\}$, допускающий все цепочки с нечетным числом вхождений символа a и бесконечным числом вхождений символа b. Можно ли

выразить указанный язык формулой LTL? Выразите этот язык, введя вспомогательные высказывания (удаляемые из допускаемых траекторий).

10. Приведите пример ω-языка, не являющегося ω-регулярным.
11. Постройте автомат Бюхи, допускающий все траектории, на которых выполняются следующие формулы LTL ($AP = \{p, q\}$):
    a. $\mathbf{X}\{p \vee q\}$;
    b. $\mathbf{G}\{\neg(p \wedge q)\}$;
    c. $p\ \mathbf{U}\ q$;
    d. $\mathbf{G}\{p \rightarrow \mathbf{X}q\}$.
12. Предложите алгоритм отыскания компонент сильной связности ориентированного графа. Оцените его сложность.
13. Докажите корректность алгоритма вложенного поиска в глубину.
14. Напишите программу (на удобном вам языке программирования), реализующую вложенный поиск в глубину.

Лекция 13. Синтез автомата Бюхи по формуле LTL

Вместе с тем можно указать два требования, предъявление которых к выбираемому или создаваемому метаязыку весьма естественно. Первое требование отражает, так сказать, интересы заказчика и относится к выразительности метаязыка ⟨…⟩ Второе требование соответствует точке зрения исполнителя. Оно заключается в том, чтобы существовал достаточно удобный алгоритм для решения проблемы синтеза применительно к данному метаязыку. Эти два требования в некотором смысле антагонистичны.

Б.А. Трахтенброт, Я.М. Барздинь.
Конечные автоматы (поведение и синтез)

Рассматривается ключевой этап теоретико-автоматного подхода к проверке моделей для логики LTL — построение автомата Бюхи, допускающего все те и только те траектории, на которых истинна заданная формула LTL. Описываемая процедура может быть использована и для проверки выполнимости формулы LTL: если язык, допускаемый соответствующим автоматом Бюхи, не пуст, формула выполнима — для нее существует траектория, на которой она истинна. Имеется несколько алгоритмов построения автоматов Бюхи по формулам LTL; все они схожи по сложностным характеристикам: результатом является недетерминированный автомат Бюхи, число состояний которого экспоненциально зависит от размера формулы — числа символов в ее записи. Вопросы оптимизации размера автомата выходят за рамки учебного пособия.

13.1. Постановка задачи и предварительные замечания

Для простоты мы будем рассматривать формулы LTL, в которых используются две логические связки, $\neg$ (отрицание) и $\vee$ (дизъюнкция), и два темпоральных оператора, $\mathbf{X}$ (*neXt time*) и $\mathbf{U}$ (*Until*). Это не ограничивает общности рассуждений, поскольку указанные связки и операторы образуют базис LTL (см. лекцию 9).

Пусть задано множество элементарных высказываний AP и LTL-формула φ над множеством AP. Требуется по формуле φ построить недетерминированный автомат Бюхи B_φ над алфавитом 2^{AP}, допускающий все траектории, на которых истинна формула φ, и только их. Сразу отметим, что задача имеет решение: для любой формулы LTL такой автомат существует. Обратное, вообще говоря, неверно: не для каждого автомата Бюхи найдется эквивалентная формула LTL.

Рассматриваемый алгоритм построения автомата Бюхи основан на метафоре *реализации обязательств* [Кар10], [Bai08]. Поясним, что имеется в виду. Если в формуле нет темпоральных операторов, она «реализует» свои обязательства в настоящем — обязательств в будущем у нее нет. Если же в формуле есть темпоральный оператор, у нее могут быть отложенные обязательства. Например, формула $\mathbf{F}\psi$ обязуется *когда-нибудь в будущем* удовлетворить свойство ψ (если, конечно, она не удовлетворяет его сейчас), а формула $\mathbf{G}\psi$ обязуется *всегда* удовлетворять свойство ψ. Обязательства сложной формулы есть обязательства ее подформул; реализация обязательств сложной формулы есть *согласованная* реализация обязательств подформул (при исполнении обязательств формулы ее подформулы, преследуя свои «локальные интересы», не должны конфликтовать друг с другом).

Множество всех подформул формулы φ, включая саму формулу φ, вместе с их отрицаниями[87] называется *замыканием* формулы φ и обозначается $closure(\varphi)$.

Пример 13.1

Выпишем замыкание формулы $\varphi \equiv (p \vee q)\ \mathbf{U}\ (p \wedge q)$:

$$closure(\varphi) = \{p, q, (p \vee q), (p \wedge q), \varphi, \neg p, \neg q, \neg(p \vee q), \neg(p \wedge q), \neg\varphi\}.$$

□

Далее для краткости мы будем опускать «негативные» подформулы в записи замыкания. Например, для формулы, рассмотренной выше, будем писать

$$closure(\varphi) = \{p, q, (p \vee q), (p \wedge q), \varphi\}.$$

Перед описанием алгоритма построения автомата Бюхи по формуле LTL рассмотрим следующий пример.

Пример 13.2

Попробуем определить истинность формулы $\varphi \equiv (p \vee q)\ \mathbf{U}\ (p \wedge q)$ на вычислении с траекторией $(\{p\}\{q\}\{p, q\}\{q\}\emptyset)^{\omega}$ [Кар10]. Разметим каждое состояние вычисления теми подформулами φ, которые в нем истинны: начнем с элементарных высказываний и постепенно будем добавлять в разметку все более сложные подформулы, как показано в таблице 13.1.

[87] Формулы $\neg\neg\psi$ и ψ считаются совпадающими.

Таблица 13.1. Разметка состояний траектории истинными подформулами

Состояние вычисления: s	Пометка состояния: $L(s)$	Разметка подформулами φ
s_0	$\{p\}$	$\{p, (p \vee q), \varphi\}$
s_1	$\{q\}$	$\{q, (p \vee q), \varphi\}$
s_2	$\{p, q\}$	$\{p, q, (p \vee q), (p \wedge q), \varphi\}$
s_3	$\{q\}$	$\{q, (p \vee q)\}$
s_4	$\emptyset$	$\emptyset$

Разметка состояний подформулами без темпоральных операторов тривиальна — для этого достаточно знать пометку состояния. Истинность подформул, содержащих темпоральные операторы, устанавливать сложнее — для этого приходится «заглядывать в будущее». Например, находясь в состоянии s_0 или s_1 и не зная будущего, нельзя сказать, истинна формула φ или ложна. В состоянии s_2 необходимая информация появляется: $s_2 \vDash (p \wedge q)$. Поскольку разметка каждого состояния, начиная с s_0 (включительно) и заканчивая s_2 (не включительно), содержит $(p \vee q)$, формула φ добавляется в разметку всех этих состояний, включая s_2 (см. определение оператора $\mathbf{U}$ в лекции 9). Так как формула φ входит в разметку начального состояния, она истинна на траектории:

$$(\{p\}\,\{q\}\,\{p, q\}\,\{q\}\,\emptyset)^{\omega} \vDash (p \vee q)\ \mathbf{U}\ (p \wedge q).$$

□

Описанный в примере метод разметки намекает на то, как построить автомат Бюхи по заданной формуле LTL. Состояниями такого автомата будут «согласованные» разметки (множества подформул), а переходы определяются по правилам реализации обязательств.

13.2. Состояния автомата Бюхи

Теперь мы готовы описать алгоритм построения автомата Бюхи B_{φ} по формуле φ. Результатом алгоритма является обобщенный автомат Бюхи, который может быть преобразован в эквивалентный ему автомат Бюхи (см. лекцию 12). Для краткости мы будем называть его автоматом Бюхи или просто автоматом. Приводимое здесь описание метода базируется на книгах [Кар10] и [Bai08], а также работе [Wol02].

Состояниями автомата B_{φ} являются множества подформул формулы φ, возможно с отрицаниями, т.е. $S \subseteq 2^{closure(\varphi)}$. Подформулы, составляющие одно состояние, должны быть *локально согласованными* (*непротиворечивыми*) — противореча друг другу, подформулы не могут реализовать обязательства исходной формулы. Сверх того, состояния должны

быть *максимально насыщенными* по включению подформул — это конкретизирует способ реализации обязательств[88].

Множество $s \subseteq closure(\varphi)$ называется *элементарным множеством*, или *атомом*, если выполнены следующие свойства:

1. для всех $\chi \in closure(\varphi)$:

   $\chi \in s$ тогда и только тогда, когда $\neg\chi \notin s$;

2. для всех $(\chi \vee \psi) \in closure(\varphi)$:

   $(\chi \vee \psi) \in s$ тогда и только тогда, когда $\chi \in s$ или $\psi \in s$;

3. для всех $(\chi \mathbf{U} \psi) \in closure(\varphi)$:

   a. если $(\chi \mathbf{U} \psi) \in s$ и $\psi \notin s$, то $\chi \in s$;

   b. если $\psi \in s$, то $(\chi \mathbf{U} \psi) \in s$.

Перечисленные требования называются *условиями локальной согласованности* (и *максимальной насыщенности*). Множество всех элементарных множеств замыкания формулы φ обозначается $atoms(\varphi)$. Определение можно очевидным образом расширить для других логических связок и темпоральных операторов.

Почему в условиях локальной согласованности не упоминается темпоральный оператор $\mathbf{X}$? Потому что у формул вида $\mathbf{X}\psi$ нет обязательств в настоящем — они сосредоточены в следующем моменте времени.

Построение элементарных множеств для заданной формулы LTL удобно организовать в форме следующей процедуры:

1. выписать входящие в формулу элементарные высказывания и подформулы вида $\mathbf{X}\psi$, предварительно их упростив:

   $$\mathbf{X}\neg\psi \equiv \neg\mathbf{X}\psi \quad \text{и} \quad \mathbf{X}(\chi \vee \psi) \equiv \mathbf{X}\chi \vee \mathbf{X}\psi;$$

[88] Такие состояния тесно связаны с множествами Хинтикки (Jaakko Hintikka, 1929-2015), т.е. максимальными согласованными множествами формул, используемыми в табличных методах проверки выполнимости (в контексте LTL они обретают вид структур [Ben12]).

2. перебрать все комбинации значений истинности выписанных условий и для каждой из них построить множество истинных подформул исходной формулы, полученных из этих условий применением классических логических связок;
3. для каждой подформулы вида $\chi \mathbf{U} \psi$ (в порядке увеличения сложности формул) проверить вхождение ψ в каждое из построенных на шаге 2 множеств s:
    a. если $\psi \in s$, добавить подформулу $\chi \mathbf{U} \psi$ в множество s;
    b. иначе:
        i. если $\chi \in s$, построить копию s с добавленной подформулой $\chi \mathbf{U} \psi$;
        ii. в противном случае оставить s без изменений.

Итак, состояниями автомата B_φ являются элементарные множества замыкания φ. Интуитивно понятно, что начальными состояниями автомата должны быть такие элементарные множества, которые содержат саму формулу φ — именно в них задаются обязательства по ее удовлетворению.

Пример 13.3

Для формулы $\varphi \equiv (p \vee q) \mathbf{U} (p \wedge q)$ существуют шесть элементарных множеств подформул: они показаны в таблице 13.2 (как мы договаривались, подформулы с отрицаниями не указываются, но подразумеваются).

Таблица 13.2. Элементарные множества подформул формулы $(p \vee q) \boldsymbol{U} (p \wedge q)$

Значения истинности		Насыщение (классические связки)	Насыщение (темпоральные операторы)
p	q		
$false$	$false$	$\emptyset$	$s_1 = \emptyset$
$true$	$false$	$\{p, (p \vee q)\}$	$s_2 = \{p, (p \vee q)\}$
			$s_3 = \{p, (p \vee q), \varphi\}$
$false$	$true$	$\{q, (p \vee q)\}$	$s_4 = \{q, (p \vee q)\}$
			$s_5 = \{q, (p \vee q), \varphi\}$
$true$	$true$	$\{p, q, (p \vee q), (p \wedge q)\}$	$s_6 = \{p, q, (p \vee q), (p \wedge q), \varphi\}$

Роль начальных состояний автомата B_φ выполняют элементарные множества s_3, s_5 и s_6.

□

13.3. Переходы автомата Бюхи

Если формула, входящая в некоторое состояние автомата, не содержит темпоральных операторов, у нее нет обязательств в будущем — она никак не ограничивает переходы. Для

темпоральных формул ситуация иная: если состояние автомата содержит формулу $\mathbf{X}\psi$, то следующее состояние обязано содержать ψ; если состояние автомата содержит формулу $\chi\ \mathbf{U}\ \psi$, то либо оно содержит ψ, либо в нем есть χ, а в следующем — $\chi\ \mathbf{U}\ \psi$. Указанные правила легко формализуются: соседние состояния автомата (обозначим их s и s') должны удовлетворять следующим ограничениям:

1. для всех $\mathbf{X}\psi \in closure(\varphi)$:

   $\mathbf{X}\psi \in s$ тогда и только тогда, когда $\psi \in s'$;

2. для всех $\chi\ \mathbf{U}\ \psi \in closure(\varphi)$:

   $\chi\ \mathbf{U}\ \psi \in s$ тогда и только тогда, когда $\psi \in s$ или же $\chi \in s$ и $\chi\ \mathbf{U}\ \psi \in s'$.

Перечисленные требования называются *условиями реализации обязательств* (обозначение: $s \to s'$). Их можно определить и для других темпоральных операторов.

Обратите внимание на связку «тогда и только тогда»: в частности, если в $closure(\varphi)$ есть подформула вида $\mathbf{X}\psi$, то переход в состояние, содержащее ψ, возможен *только* из состояний, содержащих $\mathbf{X}\psi$.

Каким образом в элементарном множестве может появиться формула вида $\mathbf{X}\psi$, ведь темпоральный оператор $\mathbf{X}$ не упоминается в условиях локальной согласованности? Именно из-за того, что условия локальной согласованности не накладывают ограничений на вхождение формул вида $\mathbf{X}\psi$ в элементарные множества: есть множества, в которые они входят, а есть множества, в которые — нет (входят, но с отрицаниями):

$$\{\psi, \mathbf{X}\psi\}, \qquad \{\psi, \neg\mathbf{X}\psi\}, \qquad \{\neg\psi, \mathbf{X}\psi\}, \qquad \{\neg\psi, \neg\mathbf{X}\psi\}.$$

Пример 13.4

Рассмотрим формулу $\varphi \equiv (p \vee q)\ \mathbf{U}\ (p \wedge q)$ и элементарные множества $s_2 = \{p, (p \vee q)\}$ и $s_3 = \{p, (p \vee q), \varphi\}$ (см. пример 13.3).

Определим множества $\delta(s_2) = \{s_2' \mid s_2 \to s_2'\}$ и $\delta(s_3) = \{s_3' \mid s_3 \to s_3'\}$. Выпишем условие реализации обязательств (условие одно, поскольку формула содержит один темпоральный оператор — $\mathbf{U}$):

$\varphi \in s$ тогда и только тогда, когда $(p \wedge q) \in s$ или $(p \vee q) \in s$ и $\varphi \in s'$.

Для состояний s_2 и s_2' имеем: поскольку $\varphi \notin s_2$, то $(p \wedge q) \notin s_2$ и, кроме того, $(p \vee q) \notin s_2$ или $\varphi \notin s_2'$, что равносильно $\varphi \notin s_2'$.

- $\delta(s_2) = \{s_1, s_2, s_4\}$ (все состояния, не содержащие φ), где
  - $s_1 = \emptyset$;
  - $s_2 = \{p, (p \vee q)\}$;
  - $s_4 = \{q, (p \vee q)\}$.

Для состояний s_3 и s_3' имеем: поскольку $\varphi \in s_3$, то $(p \wedge q) \in s_3$ или $(p \vee q) \in s_3$ и $\varphi \in s_3'$, что равносильно $\varphi \in s_3'$.

- $\delta(s_3) = \{s_3, s_5, s_6\}$ (все состояния, содержащие φ), где
  - $s_3 = \{p, (p \vee q), \varphi\}$;
  - $s_5 = \{q, (p \vee q), \varphi\}$;
  - $s_6 = \{p, q, (p \vee q), (p \wedge q), \varphi\}$.

□

13.4. Допускающие состояния автомата Бюхи

Для определения того, какие состояния автомата являются допускающими, а какие — нет, используется следующая идея. Искомый автомат является обобщенным автоматом Бюхи с несколькими множествами допускающих состояний. Каждое такое множество соответствуют определенной подформуле исходной формулы φ. Если $closure(\varphi) = \{\psi_1, \dots, \psi_k\}$, то автомат B_φ содержит k допускающих множеств: $\boldsymbol{\mathcal{F}} = \{\mathcal{F}_{\psi_1}, \dots, \mathcal{F}_{\psi_k}\}$. Состояние s является допускающим для подформулы ψ, т.е. $s \in \mathcal{F}_\psi$, если в s реализованы все обязательства ψ.

Для подформул, не содержащих темпоральных операторов, обязательства реализуются непосредственно в состояниях, их включающих, поэтому в искомом автомате нет состояний, в которых эти обязательства не выполнены (либо нет подформулы, либо ее обязательства исполняются «на месте»). Похожая ситуация имеет место с подформулами вида $\mathbf{X}\psi$: отличие лишь в том, что за реализацию их обязательств отвечают не состояния, включающие $\mathbf{X}\psi$, а состояния, следующие за ними. Таким образом, подформулы без темпоральных операторов и подформулы вида $\mathbf{X}\psi$ могут не учитываться при определении допускающих состояний.

Для каждой подформулы вида $\chi \mathbf{U} \psi$ множество допускающих состояний включает все состояния, в которых нет $\chi \mathbf{U} \psi$ или есть ψ:

$$s \in \mathcal{F}_{\chi \mathbf{U} \psi} \text{ тогда и только тогда, когда } (\chi \mathbf{U} \psi) \notin s \text{ или } \psi \in s.$$

Пример 13.5

Допускающие состояния автомата Бюхи B_φ для формулы $\varphi \equiv (p \vee q) \mathbf{U} (p \wedge q)$ (см. пример 13.3) приведены в таблице 13.3.

Обратите внимание: поскольку в φ один оператор $\mathbf{U}$, в B_φ имеется ровно одно допускающее множество состояний.

Таблица 13.3. Допускающие состояния автомата Бюхи B_φ для формулы $\varphi \equiv (p \vee q) \boldsymbol{U} (p \wedge q)$

Допускающее состояние	Комментарий
$s_1 = \emptyset$	$\varphi \notin s_1$
$s_2 = \{p, (p \vee q)\}$	$\varphi \notin s_2$
$s_4 = \{q, (p \vee q)\}$	$\varphi \notin s_4$
$s_6 = \{p, q, (p \vee q), (p \wedge q), \varphi\}$	$\varphi \in s_6$ и $(p \wedge q) \in s_6$

□

13.5. Алгоритм построения автомата Бюхи

Подведем итог. Пусть AP — множество элементарных высказываний, а φ — формула LTL над множеством AP. Обобщенный автомат Бюхи $B_\varphi = \langle S, \Sigma, S_0, \delta, \boldsymbol{\mathcal{F}} \rangle$, допускающий те и только те траектории, на которых истинна формула φ, определяется следующим образом:

- $S = atoms(\varphi)$;
- $\Sigma = 2^{AP}$;
- $S_0 = \{s \in S \mid \varphi \in s\}$;
- $\delta: S \times \Sigma \to 2^S$ определяется следующим образом:

$$\delta(s, x) = \begin{cases} \{ s' \mid s \to s' \}, & \text{если } x = (s \cap AP); \\ \emptyset, & \text{иначе;} \end{cases}$$

- $\boldsymbol{\mathcal{F}} = \{\mathcal{F}_{\chi \mathbf{U} \psi} \mid \chi \mathbf{U} \psi \in closure(\varphi)\}$, где $\mathcal{F}_{\chi \mathbf{U} \psi} = \{s \in S \mid \big((\chi \mathbf{U} \psi) \notin s\big) \vee (\psi \in s)\}$.

Очевидно, что описанный метод не гарантирует минимальности автомата B_φ: вопросы оптимизации выходят за рамки пособия. Нетрудно показать, что число состояний построенного автомата оценивается величиной $O(2^{|\varphi|})$, где $|\varphi|$ — длина формулы φ, т.е. число символов в ее записи.

Доказательство корректности алгоритма можно найти, например, в книге [Bai08].

Пример 13.6

Пусть $AP = \{p, q\}$. Построим автомат Бюхи $B_\varphi = \langle S, \Sigma, S_0, \delta, \mathcal{F} \rangle$ для уже рассмотренной формулы $\varphi \equiv (p \vee q)\ \mathbf{U}\ (p \wedge q)$.

- $S = atoms(\varphi) = \{s_1, \ldots, s_6\}$, где[89]
  - $s_1 = \emptyset$;
  - $s_2 = \{p, (p \vee q)\}$;
  - $s_3 = \{p, (p \vee q), \varphi\}$;
  - $s_4 = \{q, (p \vee q)\}$;
  - $s_5 = \{q, (p \vee q), \varphi\}$;
  - $s_6 = \{p, q, (p \vee q), (p \wedge q), \varphi\}$.
- $\Sigma = 2^{AP} = \big\{\emptyset, \{p\}, \{q\}, \{p, q\}\big\}$;
- $S_0 = \{\, s \in S \mid \varphi \in s \,\} = \{s_3, s_5, s_6\}$;
- $\mathcal{F} = \mathcal{F}_\varphi = \big\{\, s \in S \mid (\varphi \notin s) \vee \big((p \wedge q) \in s\big) \big\} = \{s_1, s_2, s_4, s_6\}$[90];
- δ определяется следующим образом (для наглядности входной символ не указан):
  - $\delta(s_1) = \{s_1, \ldots, s_6\}$ — все состояния;
  - $\delta(s_2) = \{s_1, s_2, s_4\}$ — нет переходов в состояния, содержащие φ;
  - $\delta(s_3) = \{s_3, s_5, s_6\}$ — нет переходов в состояния, не содержащие φ;
  - $\delta(s_4) = \{s_1, s_2, s_4\}$;
  - $\delta(s_5) = \{s_3, s_5, s_6\}$;
  - $\delta(s_6) = \{s_1, \ldots, s_6\}$.

[89] Еще раз подчеркнем, что подформулы с отрицаниями опущены.

[90] Заметим, что в автомате B_φ имеется одно множество допускающих состояний, т.е. это обычный автомат Бюхи.

Граф состояний построенного автомата Бюхи показан на рис. 13.1.

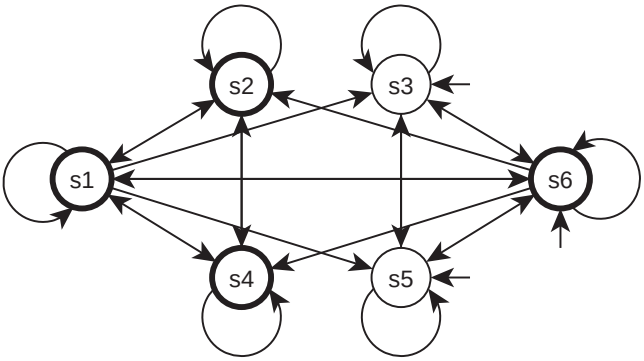


Рисунок 13.1. Автомат Бюхи B_φ для формулы $\varphi \equiv (p \vee q)\ \mathbf{U}\ (p \wedge q)$

□

Пример 13.7

Пусть, как и прежде, $AP = \{p, q\}$. Построим автомат Бюхи $B_\varphi = \langle S, \Sigma, S_0, \delta, \mathcal{F} \rangle$ для формулы $\varphi \equiv \mathbf{G}\{p \to \mathbf{X}q\}$. Прежде всего, заметим, что[91]:

$$\varphi \equiv \mathbf{G}\{p \to \mathbf{X}q\} \equiv \neg\mathbf{F}\neg(p \to \mathbf{X}q) \equiv \neg\big(true\ \mathbf{U}\ \neg(p \to \mathbf{X}q)\big) \equiv \neg\big(true\ \mathbf{U}\ (p \wedge \neg\mathbf{X}q)\big);$$

$$\neg\varphi \equiv true\ \mathbf{U}\ (p \wedge \neg\mathbf{X}q).$$

- $S = atoms(\varphi) = \{s_1, \ldots, s_{14}\}$ — см. таблицу 13.4[92].

Таблица 13.4. Состояния автомата Бюхи B_φ для формулы $\varphi \equiv \boldsymbol{G}\{p \to \boldsymbol{X}q\}$

Значение истинности			Насыщение (классические связки)	Насыщение (темпоральные операторы)
p	q	$\mathbf{X}q$		
$false$	$false$	$false$	$\{true\}$	$s_1 = \{true\}$
				$s_2 = \{true, \neg\varphi\}$
$true$	$false$	$false$	$\{true, p, (p \wedge \neg\mathbf{X}q)\}$	$s_3 = \{true, p, (p \wedge \neg\mathbf{X}q), \neg\varphi\}$
$false$	$true$	$false$	$\{true, q\}$	$s_4 = \{true, q\}$
				$s_5 = \{true, q, \neg\varphi\}$
$false$	$false$	$true$	$\{true, \mathbf{X}q\}$	$s_6 = \{true, \mathbf{X}q\}$
				$s_7 = \{true, \mathbf{X}q, \neg\varphi\}$
$true$	$true$	$false$	$\{true, p, q, (p \wedge \neg\mathbf{X}q)\}$	$s_8 = \{true, p, q, (p \wedge \neg\mathbf{X}q), \neg\varphi\}$

[91] Поскольку нами не были определены условия локальной согласованности и реализации обязательств для темпорального оператора $\mathbf{G}$, формула преобразуется к эквивалентной, содержащей только $\mathbf{X}$ и $\mathbf{U}$.

[92] Обратите внимание, что во все состояния входит подформула $true$ — это следствие условий локальной согласованности.

$true$	$false$	$true$	$\{true, p, \mathbf{X}q\}$	$s_9 = \{true, p, \mathbf{X}q\}$
				$s_{10} = \{true, p, \mathbf{X}q, \neg\varphi\}$
$false$	$true$	$true$	$\{true, q, \mathbf{X}q\}$	$s_{11} = \{true, q, \mathbf{X}q\}$
				$s_{12} = \{true, q, \mathbf{X}q, \neg\varphi\}$
$true$	$true$	$true$	$\{true, p, q, \mathbf{X}q\}$	$s_{13} = \{true, p, q, \mathbf{X}q\}$
				$s_{14} = \{true, p, q, \mathbf{X}q, \neg\varphi\}$

- $\Sigma = 2^{AP} = \{\emptyset, \{p\}, \{q\}, \{p, q\}\}$;
- $S_0 = \{ s \in S \mid \varphi \in s \} = \{s_1, s_4, s_6, s_9, s_{11}, s_{13}\}$ (состояния, которые не содержат $\neg\varphi$);
- $\mathcal{F} = \mathcal{F}_{\neg\varphi} = \{ s \in S \mid (\neg\varphi \notin s) \vee ((p \wedge \neg\mathbf{X}q) \in s) \} = \{s_1, s_3, s_4, s_6, s_8, s_9, s_{11}, s_{13}\}$;
- δ определяется следующим образом (для наглядности входной символ не указан):
  - $\delta(s_1) = \{s_1, s_6, s_9\}$;
  - $\delta(s_2) = \{s_2, s_3, s_7, s_{10}\}$ — состояния, содержащие $\neg\varphi$, но не содержащие q;
  - $\delta(s_3) = \{s_1, s_2, s_3, s_6, s_7, s_9, s_{10}\}$ — нет переходов в состояния, содержащие q;
  - $\delta(s_4) = \delta(s_1)$;
  - $\delta(s_5) = \delta(s_2)$;
  - $\delta(s_6) = \{s_4, s_{11}, s_{13}\}$ — состояния, содержащие q, но не содержащие $\neg\varphi$;
  - $\delta(s_7) = \{s_5, s_8, s_{12}, s_{14}\}$ — состояния, содержащие $\neg\varphi$ и q;
  - $\delta(s_8) = \delta(s_3)$;
  - $\delta(s_9) = \delta(s_6)$;
  - $\delta(s_{10}) = \delta(s_7)$;
  - $\delta(s_{11}) = \delta(s_6)$;
  - $\delta(s_{12}) = \delta(s_7)$;
  - $\delta(s_{13}) = \delta(s_6)$;
  - $\delta(s_{14}) = \delta(s_7)$.

Граф состояний построенного автомата Бюхи показан на рис. 13.2.

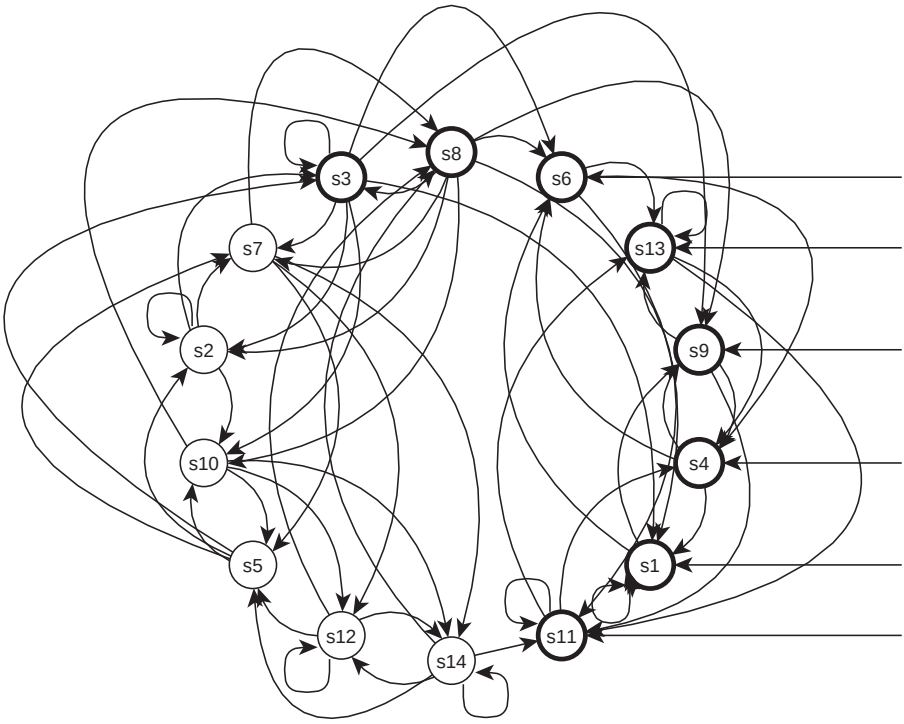


Рисунок 13.2. Автомат Бюхи B_φ для формулы $\varphi \equiv \mathbf{G}\{p \to \mathbf{X}q\}$

□

13.6. Общая схема проверки моделей для логики LTL

Синтез автомата Бюхи по формуле LTL — это ключевой этап теоретико-автоматного подхода к проверке моделей. Общая схема метода приведена на рис. 13.3. Входными данными для инструмента проверки моделей являются модель системы (например, представленная на языке PROMELA) и модель требований — формула LTL. По модели и по отрицанию формулы конструируются автоматы Бюхи. Строится их синхронная композиция — тоже автомат Бюхи. В композиции проверяется наличие допускающего цикла: если допускающего цикла нет, модель корректна относительно заданной формулы; в противном случае модель некорректна (в этом случае предъявляется контрпример).

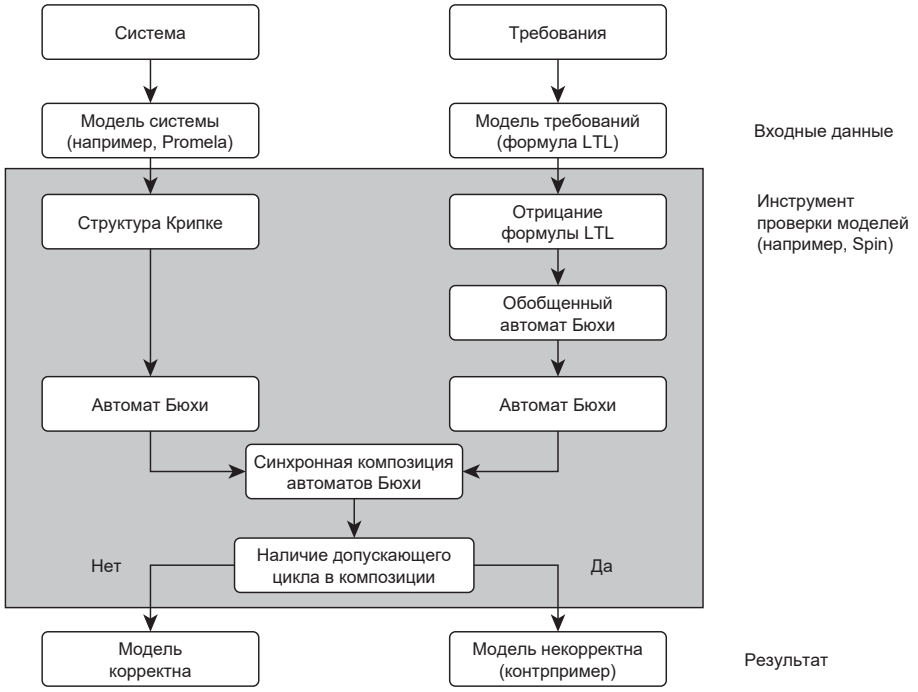


Рисунок 13.3. Общая схема проверки моделей для логики LTL

13.7. Вопросы и упражнения

1. Дайте определения замыканию формулы и элементарному множеству подформул.
2. Выпишите условия реализации обязательств для темпоральных операторов $\mathbf{X}$ и $\mathbf{U}$.
3. Для темпоральных операторов $\mathbf{F}$ и $\mathbf{G}$ определите условия локальной согласованности, условия реализации обязательств и условия на допускающие состояния.
4. Пусть φ — некоторая LTL формула, а $\neg\varphi$ — ее отрицание. Рассмотрите по шагам построение автоматов Бюхи B_{φ} и $B_{\neg\varphi}$. В чем различие между этими автоматами?
5. Предъявите пример автомата Бюхи, для которого не существует эквивалентной формулы LTL.
6. В предположении, что $AP = \{p, q\}$ выпишите элементарные множества подформул для следующих формул LTL:
   a. $\mathbf{X}p$;
   b. $\mathbf{G}p$;

c. $\mathbf{F}p$;

d. $\mathbf{GF}p$;

e. $\mathbf{FG}p$;

a. $\mathbf{G}\{p \rightarrow \mathbf{F}q\}$;

b. $p\ \mathbf{U}\ (\mathbf{X}q)$;

c. $\mathbf{G}\{p \rightarrow (p\ \mathbf{U}\ q)\}$;

d. $(\mathbf{FG}p) \rightarrow (\mathbf{GF}q)$.

7. Постройте автоматы Бюхи для следующих формул LTL ($AP = \{p, q\}$):

   a. $p \wedge (\mathbf{XG}\neg q)$;

   b. $\mathbf{F}\{p \rightarrow \mathbf{X}\neg q\}$;

   c. $\mathbf{GF}\{\neg p \wedge \mathbf{X}q\}$;

   d. $\mathbf{FG}\{\mathbf{X}p \wedge q\}$;

   e. $(p \wedge \neg\mathbf{X}q)\mathbf{U}\ (\neg p \wedge q)$;

   f. $(\mathbf{FG}p) \rightarrow (\mathbf{X}q)$;

   g. $\mathbf{G}\{p \rightarrow p\ \mathbf{U}\ q\}$.

8. Предложите способы оптимизации автомата Бюхи, конструируемого по формуле LTL.

9. Напишите программу (на удобном вам языке программирования), синтезирующую автомат Бюхи по заданной формуле LTL.

10. Напишите программу, реализующую теоретико-автоматный подход к проверке моделей для логики LTL. Входные данные: структура Крипке (в некотором формате) и формула LTL. Выходные данные — вердикт о корректности.

Лекция 14. Символическая проверка моделей

Любая схема представляется в виде системы уравнений, составленных из символов, соответствующих различным реле и переключателям схемы. Разрабатывается аппарат для преобразования этих уравнений с помощью простых математических приемов (...). Показывается, что этот аппарат в точности аналогичен исчислению высказываний символической логики.

К. Шеннон.
Символический анализ релейных и переключательных схем

Рассматривается символическая проверка моделей для логики LTL. Отличие от классического подхода, рассмотренного ранее, состоит в том, что множество состояний и отношение переходов модели, а также автомат Бюхи, построенный по формуле LTL, представляются неявно, в символической форме. Общая идея метода остается прежней, но алгоритмы поиска в графах состояний заменяются на манипуляции с их символическими представлениями, что частично решает проблему комбинаторного взрыва. Ключевую роль здесь играют способы символического представления множеств и отношений. Рассматривается один из таких способов, позволяющий создавать эффективные символические алгоритмы, — двоичные решающие диаграммы (BDD, Binary Decision Diagrams).

14.1. Символические методы верификации

Традиционные методы проверки моделей, включая рассмотренный ранее теоретико-автоматный подход (см. лекции 12 и 13), называются *методами с явным представлением состояний*: каждое состояние модели (если быть точным, синхронной композиции модельного и контрольного автоматов) некоторым образом кодируется и сохраняется в памяти компьютера. Из-за комбинаторного взрыва число состояний может превысить любые мыслимые значения, например, число атомов во Вселенной[93]. Даже если допустить, что компьютер располагает достаточным объемом памяти, то верификация может занять время, превышающее возраст Вселенной[94].

Представители «классической» школы проверки моделей прибегают к разного рода «трюкам», оптимизирующим представление состояний в памяти. В инструменте Spin, например,

---

[93] По современным представлениям, число атомов в наблюдаемой части Вселенной — величина порядка $10^{79} - 10^{81}$. Для сравнения: минимальное количество неповторяющихся шахматных партий оценивается как 10^{118} (число Шеннона). Число состояний в моделях программ может достигать 10^{1000}, и это не предел.

[94] Считается, что возраст Вселенной составляет около 10^{10} лет.

используется сжатие данных (с потерями и без): *хеширование* (*bitstate hashing*) [Hol98], *рекурсивное индексирование* (*recursive indexing*) [Hol97] и др., однако кардинально ситуацию это не меняет. Применяются техники параллельного исследования графов состояний [Бур11], что позволяет сократить время верификации на 2-3 порядка (в зависимости от числа компьютеров), но этого не всегда достаточно; кроме того, для этого требуются большие вычислительные мощности.

Альтернативу методам с явным представлением состояний составляют *методы с неявным представлением состояний — символические методы*. Суть в том, что множество состояний представляется не путем перечисления его элементов, а как решение некоторой системы ограничений. Пространство состояний системы, как правило, обладает некоторой регулярностью, что позволяет задать его в компактной форме[95]. Аналогично могут быть представлены и требования.

Пусть χ_M — описание пространства состояний модели M, а χ_φ — описание требований. Для простоты будем считать, что φ — свойство безопасности: φ должно быть истинным в любом состоянии M. Тогда проверка модели есть проверка общезначимости импликации $\chi_M \to \chi_\varphi$ или, что то же самое, невыполнимости конъюнкции $\chi_M \wedge \neg\chi_\varphi$ (см. лекцию 8).

Символическая проверка моделей была разработана в конце 1980-х — начале 90-х [Bur90]. Метод описан в докторской диссертации Кеннета Макмиллана (Kenneth McMillan) [Mcm92] и реализован в инструментах SMV (Symbolic Model Verifier), NuSMV и nuXmv [Cav14]. Изначально символическая проверка применялась для логики CTL, мы же, следуя теоретико-автоматному подходу, рассмотрим ее разновидность для LTL [Roz10].

14.1.1. Символическое представление множеств и отношений

Пусть S — не более чем счетное непустое множество. Чтобы символически представить подмножества S, можно ввести *нумерацию*, т.е. сопоставить каждому элементу S уникаль-

[95] К слову сказать, код программы — неявное и компактное представление ее пространства состояний.

ное натуральное число. Тогда любая арифметическая формула φ с одной свободной переменной x, например, алгебраическое уравнение с одной неизвестной, является описанием некоторого подмножества $X \subseteq S$[96]:

$$x \in X \text{ тогда и только тогда, когда } \vDash \varphi([\![x]\!]),$$

где $[\![x]\!]$ — номер x (предполагается, что $x \in S$).

Если множество S конечно, можно поступить проще: *закодировать* его элементы двоичными векторами — элементами множества $\mathbb{B}^n$, где $\mathbb{B} = \{0,1\}$, а $n = \lceil \log_2 |S| \rceil$. (Не ограничивая общности рассуждений, можно считать, что $|S|$ является степенью двойки.) В этом случае подмножество $X \subseteq S$ может быть задано *булевой функцией* χ_X, принимающей значение 1 на тех и только тех двоичных векторах, которые кодируют элементы X. Такие функции называются *характеристическими*.

Операции над множествами могут быть определены через *булевы операции* над их характеристическими функциями. Пусть на множестве S задана двоичная кодировка, X и Y — подмножества S, χ_X и χ_Y — их характеристические функции в заданной кодировке. В таблице 14.1 показано соответствие между операциями над множествами и операциями над характеристическими функциями.

Таблица 14.1. Выражение операций над множествами через характеристические функции

Описание	Операция над множествами	Операция над функциями
Константы		
Полное множество	S	$\chi_S \equiv 1$
Пустое множество	$\emptyset$	$\chi_\emptyset \equiv 0$
Унарные операции		
Дополнение множества	$\overline{X}$	$\chi_{\bar{X}} \equiv \neg\chi_X$
Бинарные операции		
Пересечение множеств	$X \cap Y$	$\chi_{X\cap Y} \equiv \chi_X \wedge \chi_Y$
Объединение множеств	$X \cup Y$	$\chi_{X\cup Y} \equiv \chi_X \vee \chi_Y$
Разность множеств	$X \setminus Y$	$\chi_{X\setminus Y} \equiv \chi_X \wedge \neg\chi_Y$

[96] Множество натуральных чисел, которое может быть описано формулой в языке арифметики первого порядка, называется *арифметическим множеством*.

Отношения также могут быть представлены с помощью характеристических функций. Пусть $R \subseteq S \times S$ — бинарное отношение. Если двоичная кодировка элементов S требует n битов, то χ_R — характеристическая функция R — есть булева функция от $2n$ переменных: $\vec{x} = (x_1, \dots, x_n)$ и $\vec{x}' = (x'_1, \dots, x'_n)$.

Некоторые операции над бинарными отношениями представлены в таблице 14.2. Здесь X и Y — подмножества S, а χ_X и χ_Y — их характеристические функции.

Таблица 14.2. Выражение операций над отношениями через характеристические функции

Описание	Операция над отношениями	Операция над функциями
Образ и домен отношения		
Образ отношения	$Im(R) = \{ y \in S \mid \exists x((x,y) \in R) \}$	$\chi_{Im(R)}(\vec{x}') = \exists \vec{x}(\chi_R(\vec{x}, \vec{x}')) = \bigvee_{\mathbf{x} \in \mathbb{B}^n} \chi_R(\vec{x}, \vec{x}')$
Домен отношения	$Dom(R) = \{ x \in S \mid \exists y((x,y) \in R) \}$	$\chi_{Dom(R)}(\vec{x}) = \exists \vec{x}'(\chi_R(\vec{x}, \vec{x}')) = \bigvee_{\mathbf{x}' \in \mathbb{B}^n} \chi_R(\vec{x}, \vec{x}')$
Ограничение образа и домена		
Ограничение домена	$X \lhd R = \{(x,y) \in R \mid x \in X\}$	$\chi_{X \lhd R}(\vec{x}, \vec{x}') = \chi_X(\vec{x}) \wedge \chi_R(\vec{x}, \vec{x}')$
Ограничение образа	$R \rhd Y = \{(x,y) \in R \mid y \in Y\}$	$\chi_{R \rhd Y}(\vec{x}, \vec{x}') = \chi_R(\vec{x}, \vec{x}') \wedge \chi_Y(\vec{x}')$
Образ множества относительно отношения		
Прямой образ	$R(X) = Im(X \lhd R)$	$\chi_{R(X)}(\vec{x}') = \exists \vec{x}(\chi_X(\vec{x}) \wedge \chi_R(\vec{x}, \vec{x}'))$
Обратный образ	$R^{-1}(Y) = Dom(R \rhd Y)$	$\chi_{R^{-1}(Y)}(\vec{x}) = \exists \vec{x}'(\chi_R(\vec{x}, \vec{x}') \wedge \chi_Y(\vec{x}'))$

14.1.2. Символическое представление структур Крипке

Основным формализмом, используемом в проверке моделей, является структура Крипке. В очередной раз напомним, что структурой Крипке для заданного множества элементарных высказываний AP называется четверка $\langle S, S_0, R, L \rangle$, где S — множество состояний, $S_0 \subseteq S$ — множество начальных состояний, $R \subseteq S \times S$ — отношение переходов, $L: S \to 2^{AP}$ — функция разметки.

В символической проверке моделей структура Крипке задается символически. Для этого выбирается кодировка $code: S \to \mathbb{B}^{\log |S|}$ и определяются характеристические функции множества S_0 и отношения R: χ_0 и χ_R соответственно; для описания функции L используется следующий подход: для каждого $p \in AP$ задается характеристическая функция χ_p множества $[\![p]\!] = \{s \in S \mid p \in L(s)\}$. Таким образом, символическим представлением структуры Крипке $\langle S, S_0, R, L \rangle$ является четверка $\langle code, \chi_0, \chi_R, \{\chi_p\}_{p \in AP} \rangle$.

Пример 14.1

Пусть $AP = \{p, q, r\}$. Построим символическое представление простой явно заданной структуры Крипке $M = \langle S, S_0, R, L \rangle$, где

- $S = \{s_0, s_1, s_2\}$;
- $S_0 = \{s_0\}$;
- $R = \{(s_0, s_1), (s_0, s_2), (s_1, s_0), (s_1, s_2), (s_2, s_2)\}$;
- $L = \{s_0 \mapsto \{p, q\}, s_1 \mapsto \{q, r\}, s_2 \mapsto \{r\}\}$ [Кар10].

На рис. 14.1 приведено графическое изображение этой структуры.

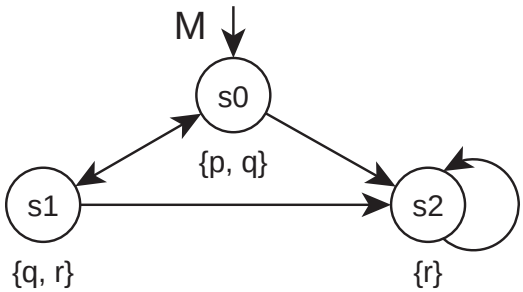


Рисунок 14.1. Пример структуры Крипке

Для представления состояний достаточно двух булевых переменных (назовем их x_1 и x_2). Возможная кодировка приведена в таблице 14.3.

Таблица 14.3. Двоичная кодировка состояний

Состояние — s	Двоичная кодировка $code(s)$		Характеристическая функция χ_s
	x_1	x_2	
s_0	0	0	$\neg x_1 \wedge \neg x_2$
s_1	0	1	$\neg x_1 \wedge x_2$
s_2	1	0	$x_1 \wedge \neg x_2$

Характеристическая функция $\chi_0(x_1, x_2)$ множества начальных состояний S_0 определяется формулой $\neg x_1 \wedge \neg x_2$.

Характеристическая функция $\chi_R(x_1, x_2; x_1', x_2')$ отношения переходов R определяется, как показано в таблице 14.4.

Таблица 14.4. Характеристическая функция отношения переходов

Таблица истинности χ_R				
x_1	x_2	x_1'	x_2'	χ_R
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Символическое представление χ_R	
Совершенная ДНФ	Упрощенная формула
$\neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge x_2'$	$\neg x_1 \wedge \neg x_2 \wedge (x_1' \oplus x_2')$
$\neg x_1 \wedge \neg x_2 \wedge x_1' \wedge \neg x_2'$	
$\neg x_1 \wedge x_2 \wedge \neg x_1' \wedge \neg x_2'$	$\neg x_1 \wedge x_2 \wedge \neg x_2'$
$\neg x_1 \wedge x_2 \wedge x_1' \wedge \neg x_2'$	
$x_1 \wedge \neg x_2 \wedge x_1' \wedge \neg x_2'$	$x_1 \wedge \neg x_2 \wedge x_1' \wedge \neg x_2'$

Характеристические функции, задающие разметку L, определяются согласно таблице 14.5.

Таблица 14.5. Характеристические функции, задающие разметку

Элементарное высказывание — φ	Множество $[\![\varphi]\!]$	Характеристическая функция χ_φ
p	$\{s_0\}$	$\neg x_1 \wedge \neg x_2$
q	$\{s_0, s_1\}$	$\neg x_1$
r	$\{s_1, s_2\}$	$x_1 \oplus x_2$

□

14.1.3. Символические алгоритмы

В основе символических методов проверки моделей лежат так называемые *символические алгоритмы* — алгоритмы, оперирующие не с явно заданными множествами объектов и отношениями между ними, а с их символическими представлениями. Пока нам не важен способ представления множеств и отношений — это могут быть формулы, схемы и т.п.

Проиллюстрируем понятие символического алгоритма на примере анализа достижимости состояний в графе [Кар10]. Пусть $\langle S, S_0, R\rangle$ — ориентированный граф (система переходов, моделирующая программу): S — множество состояний, $S_0 \subseteq S$ —множество начальных состояний, $R \subseteq S \times S$ — отношение переходов. Задача состоит в построении множества всех

состояний графа, достижимых из начальных состояний, т.е. множества $R^*(S_0)$, где R^* — рефлексивное и транзитивное замыкание отношения R.

Алгоритм решения этой задачи показан в таблице 14.6. Он основан на вычислении неподвижной точки отображения R[97] (см. лекцию 7). Алгоритм этот носит теоретический характер — при явном представлении графа разумнее использовать поиск в глубину или в ширину (см. лекцию 11).

Таблица 14.6. Обычный и символический алгоритмы построения множества достижимых состояний

Обычный алгоритм	Символический алгоритм

S_{old} := S_0;
S_{new} := $S_{old} \cup R(S_{old})$;
while $S_{new} \neq S_{old}$ **do**
  S_{old} := S_{new};
  S_{new} := $S_{old} \cup R(S_{old})$
end
	χ_{old} := χ_0;
	χ_{new} := $\chi_{old} \vee (\exists \mathbf{x}(\chi_{old} \wedge \chi_R))[\mathbf{x}' := \mathbf{x}]$;
	while $\chi_{new} \neq \chi_{old}$ **do**
	  χ_{old} := χ_{new};
	  χ_{new} := $\chi_{old} \vee (\exists \mathbf{x}(\chi_{old} \wedge \chi_R))[\mathbf{x}' := \mathbf{x}]$
	end

В левом столбце представлен обычный алгоритм, в котором множества состояний и отношения переходов представлены явно; в правом — его символический аналог, где операции над множествами и отношениями заменены на операции над их характеристическими функциями. Подстановка $[\vec{x}' \coloneqq \vec{x}]$ используется для того, чтобы привести представление образа множества к исходным переменным. Результаты возвращаются в переменных S_{new} и χ_{new} соответственно.

Существует упрощенный вариант символической проверки моделей, в котором строится не замыкание отношения переходов, а итерация ограниченной глубины:

$$R^{(k)} = \bigcup_{i=0}^{k} R^i.$$

Подход известен как *ограниченная проверка модели* (*BMC*, *Bounded Model Checking*) [Bie03]. Этот метод не является полным, но позволяет эффективно обнаруживать ошибки. Пусть для простоты проверяемое свойство имеет вид $\mathbf{G}\varphi$. На каждом шаге $i \in \{0, \ldots, k\}$ проверяется выполнимость формулы $\chi_i \wedge \neg\chi_\varphi$, где χ_i — характеристическая функция множества состояний, достижимых из начальных за i переходов, а χ_φ — представление свойства

[97] Отношение $R \subseteq S \times S$ можно представить как отображение $R: 2^S \to 2^S$, определяемое равенством $R(X) = Im(X \lhd R)$.

φ через переменные, кодирующие состояния. Если формула оказывается выполнимой, в модели есть ошибка.

Заметим, что сравнение $\chi_{new} \neq \chi_{old}$ в символическом алгоритме построения множества достижимых состояний— это проверка эквивалентности двух символических представлений, например, двух логических формул. Вообще говоря, это нетривиальная задача, которая для логики высказываний является NP-полной, а в общем случае — алгоритмически неразрешимой (см. лекцию 8). Ниже мы рассмотрим способ представления булевых функций, упрощающий эту процедуру.

14.2. Двоичные решающие диаграммы

Способ, о котором идет речь, связан с *двоичными решающими диаграммами* (*BDD*, *Binary Decision Diagrams*). Булева функция представляется в виде корневого ориентированного ациклического графа, т.е. дерева со «склеенными» вершинами. Нелистовые вершины графа помечены переменными; из каждой такой вершины исходят две дуги: одна соответствует значению 0 этой переменной, другая — значению 1; листовых вершин две: $\boxed{0}$ и $\boxed{1}$. Такие графы и называются двоичными решающими диаграммами.

Двоичные решающие диаграммы являются чрезвычайно популярной структурой данных. Первой работой по BDD считается статья [Lee59], датируемая 1959 г., однако потенциал BDD раскрыл в 1986 г. Рэндел Брайант (Randal Bryant, род. в 1952 г.) [Bry86] — он предложил фиксировать порядок переменных и повторно использовать общие подграфы. Важной вехой стала работа 1978 г. Шелдона Акерса (Sheldon Akers) [Ake78]. Следует отметить и русскоязычную статью Раймунда Убара (Raimund Ubar, род. в 1941 г.), датируемую 1976 г. [Уба76], в которой описана схожая структура данных, названная *альтернативным графом*, и ее применение к тестированию цифровых устройств.

14.2.1. Синтаксис и семантика диаграмм

Пусть V — множество переменных. *Двоичной решающей диаграммой*, или просто *диаграммой*, называется пятерка $\langle N \cup \{\boxed{0}, \boxed{1}\}, n_0, L, low, high\rangle$, где $N \cup \{\boxed{0}, \boxed{1}\}$ — множество вершин (причем $N \cap \{\boxed{0}, \boxed{1}\} = \emptyset$), $n_0 \in N \cup \{\boxed{0}, \boxed{1}\}$ — корень, $L: N \to V$ — функция разметки, $low: N \to N \cup \{\boxed{0}, \boxed{1}\}$ и $high: N \to N \cup \{\boxed{0}, \boxed{1}\}$ — отображения задающие дуги (так называемые low- и $high$-дуги).

Для нелистовой вершины n введем следующие обозначения: $then(n)$ и $else(n)$ — поддиаграммы, корнями которых являются вершины $high(n)$ и $low(n)$ соответственно. Для диаграммы F с корнем n_0 будем считать:

- $root(F) \equiv n_0$ и, если $n_0 \notin \{\boxed{0}, \boxed{1}\}$,
  - $L(F) \equiv L(n_0)$;
  - $then(F) \equiv then(n_0)$;
  - $else(F) \equiv else(n_0)$.

Диаграммы, корнями которых являются $\boxed{0}$ или $\boxed{1}$, будем называть *тривиальными* и обозначать $ZERO$ и ONE соответственно.

Семантика диаграммы F, т.е. представляемая ей булева функция (обозначается: $[\![F]\!]$), определяется следующим образом:

- Если $root(F) = \boxed{0}$, то $[\![F]\!] \equiv 0$.
- Если $root(F) = \boxed{1}$, то $[\![F]\!] \equiv 1$.
- Если $root(F) \notin \{\boxed{0}, \boxed{1}\}$, то $[\![F]\!] \equiv (L(F) \wedge [\![then(F)]\!]) \vee (\neg L(F) \wedge [\![else(F)]\!])$.

Таким образом, представление булевых функций с помощью двоичных решающих диаграмм имеет в своей основе *разложение Шеннона* (Claude Shannon, 1916-2001):

$$f(x_1, \dots, x_n) = \big(x_i \wedge f(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n)\big) \vee \big(\neg x_i \wedge f(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n)\big).$$

При изображении диаграмм, чтобы различать low- и $high$-дуги, первые изображаются пунктиром. Листовые вершины изображаются квадратами, нелистовые — окружностями. Пометки вершин рисуются внутри соответствующих фигур.

Пример 14.2

Диаграмма, показанная на рис. 14.2 [Yua06], соответствует следующей ДНФ:

$$(\neg x_1 \wedge \neg y_1 \wedge z_1) \vee (\neg x_1 \wedge y_1 \wedge z_1) \vee (x_1 \wedge \neg y_2 \wedge z_1) \vee (x_1 \wedge y_2 \wedge \neg z_2) \vee (x_1 \wedge y_2 \wedge z_2).$$

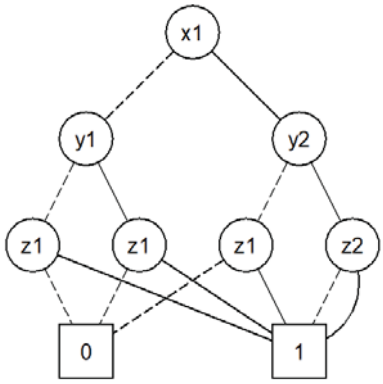


Рисунок 14.2. Пример двоичной решающей диаграммы

□

14.2.2. Сокращенные упорядоченные диаграммы

Данное выше определение допускает неоднозначность в представлении функций диаграммами: возможны ситуации, когда вдоль разных путей от корня до листовой вершины используются разные порядки переменных, и даже ситуации, когда одна и та же переменная используется вдоль одного пути несколько раз. Эта неоднозначность затрудняет проверку эквивалентности характеристических функций.

В этом разделе мы рассмотрим усовершенствованный вид двоичных решающий диаграмм, лишенный подобных недостатков.

Диаграмма называется *упорядоченной*, если на множестве переменных задан линейный порядок $\prec$ и для любых двух вершин v и u справедливо соотношение $L(v) \prec L(u)$, если вершина u — потомок вершины v. Диаграмма называется *сокращенной*, если в ней отсутствуют изоморфные подграфы, а также вершины с одним последователем, т.е. вершины, в которых последователи по low- и $high$-дугам совпадают.

Любая диаграмма может быть приведена к сокращенной форме путем применения следующих правил (пока хотя бы одно из них применимо):

1. Если есть два изоморфных подграфа, удалить один из них и перенаправить ведущие в него дуги в соответствующие вершины оставшегося подграфа.

2. Если есть вершина с одним последователем, удалить ее и перенаправить ведущие в нее дуги в этого последователя.

Когда идет речь о двоичных решающих диаграммах, обычно имеются в виду с*окращенные упорядоченные двоичные решающие диаграммы* (*ROBDD*, *Reduced Ordered Binary Decision Diagrams*); для краткости мы будем называть их просто диаграммами. ROBDD обладают важным свойством — представление функций с их помощью является *каноническим*: две функции эквивалентны тогда и только тогда, когда их диаграммы совпадают. Это свойство может также использоваться для проверки выполнимости и общезначимости логических формул (см. лекцию 8).

Пример 14.3

Применим правила сокращения 1 и 2 к диаграмме из предыдущего примера.

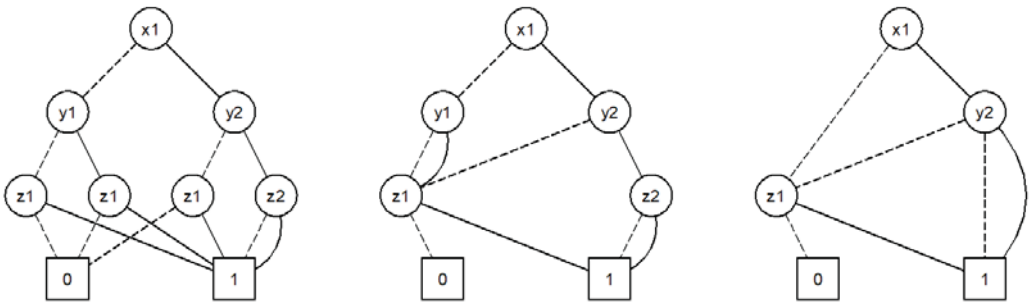


Рисунок 14.3. Построение сокращенной диаграммы: слева — исходная диаграмма; в центре — результат применения правила 1; справа — результат применения правила 2

□

Следует отметить, что размер диаграммы существенно зависит от порядка переменных. Известны эвристики для выбора «хорошего» порядка, однако их рассмотрение выходит за рамки пособия.

14.2.3. Манипуляции с диаграммами

Манипуляции с двоичными решающими диаграммами базируются на процедуре $Apply$, основанной, в свою очередь, на разложении Шеннона [Bry86]. Введем следующие обозначения:

- $Reduce(F)$ — результат применения правил сокращения к диаграмме F;
- $Compose(x, T, E)$ — результат композиции диаграмм T и E по переменной x, т.е. диаграмма F, в которой
  - $L(F) = x$;
  - $then(F) = T$;
  - $else(F) = E$.

Пусть f и g — булевы функции, а $\circ$ — булева операция. Применяя разложение Шеннона, получим

$$f \circ g = \left(x \wedge (f \circ g)_{|x=1}\right) \vee \left(\neg x \wedge (f \circ g)_{|x=0}\right) =$$
$$\left(x \wedge \left(f_{|x=1} \circ g_{|x=1}\right)\right) \vee \left(\neg x \wedge \left(f_{|x=0} \circ g_{|x=0}\right)\right).$$

Применяя это равенство, определим процедуру $Apply$, конструирующую диаграмму $F \circ G$ (функции $f \circ g$) по диаграммам F и G (функций f и g соответственно):

$$Apply(F \circ G) = Reduce\left(Compose\left(x, Apply\left(F_{|x=1} \circ G_{|x=1}\right), Apply\left(F_{|x=0} \circ G_{|x=0}\right)\right)\right).$$

Запись $F_{|x=\sigma}$ означает диаграмму, полученную из F путем следующего преобразования: для каждой вершины n, такой что $L(n) = x$, дуги, входящие в n, перенаправляются в $low(n)$, если $\sigma = 0$, или $high(n)$, если $\sigma = 1$.

В определении $Apply$ заложена неоднозначность выбора переменной x, по которой производится разложение. В целях оптимизации в качестве x используют «минимальную» из переменных, помечающих корни F и G. В этом случае имеет место равенство

$$F_{|x=\sigma} = \begin{cases} then(F), & \text{если } x = L(F) \text{ и } \sigma = 1; \\ else(F), & \text{если } x = L(F) \text{ и } \sigma = 0; \\ F, & \text{если } x \prec L(F). \end{cases}$$

Заключительные ветви рекурсии в $Apply(F \circ G)$ должны учитывать семантику операции $\circ$. Например, для конъюнкции можно использовать следующие проверки:

- если $root(F) = \boxed{0}$ или $root(G) = \boxed{0}$, возвратить $ZERO$;
- если $root(F) = \boxed{1}$, возвратить G;
- если $root(G) = \boxed{1}$, возвратить F.

Для построения диаграммы отрицания некоторой функции можно воспользоваться равенством $\neg f = f \oplus 1$ (очевидно, что это можно сделать и более простым способом). Отметим также, что операции квантификации, используемые для построения прямого и обратного образа бинарного отношения допускают эффективную реализацию с помощью двоичных решающих диаграмм.

Существует несколько свободно распространяемых библиотек для работы с двоичными решающими диаграммами. Одна из них — CUDD [Cudd], используемая в инструменте проверки моделей nuXmv [Cav14].

14.3. Символическая проверка моделей для логики LTL

Метод символической проверки моделей для логики LTL соответствует рассмотренному ранее теоретико-автоматному подходу (см. лекции 12 и 13), но оперирует не с явными представлениями автоматов, а с символическими. На вход подаются формальное описание модели (M) и формула LTL (φ). Метод состоит из следующих шагов:

1. построение символического представления $\mathcal{R}(B_M)$ автомата B_M;
2. построение символического представления $\mathcal{R}(B_{\neg\varphi})$ автомата $B_{\neg\varphi}$;
3. построение символического представления $\mathcal{R}(B_M \otimes B_{\neg\varphi})$ автомата $B_M \otimes B_{\neg\varphi}$ и проверка языка $\mathcal{L}_{B_M \otimes B_{\neg\varphi}}$ на пустоту.

14.3.1. Символическое представление модели программы

По большому счету, модель M, подаваемая на вход, уже является символическим представлением[98]. Задача состоит в том, чтобы из одного представления построить другое, более подходящее для верификации. Таким представлением является описание автомата (множества начальных состояний и отношения переходов) через характеристические функции.

[98] Инструменты проверки моделей работают не со структурами Крипке, представленными явно, а с описаниями моделей на специальных языках, например, PROMELA (см. лекцию 10).

Пусть $V = \{x_1, \ldots, x_n\}$ — множество переменных. Напомним, что *конфигурацией* программы называется пара $\langle l, s \rangle$, где l — состояние управления (метка текущего оператора), а s — состояние данных (функция означивания переменных). Пусть задана начальная метка l_0 и предусловие χ_0: начальным состоянием данных может быть любое состояние s, такое что $s \vDash \chi_0$.

Отношение переходов модели можно представить символически, в виде логических формул. Например, при исполнении оператора присваивания $x_{i_0} \coloneqq t$ конфигурация $\langle l, s \rangle$ перейдет в $\langle l', s' \rangle$, где l' — метка следующего оператора, а $s' = s[x_{i_0} \coloneqq s(t)]$. Переход $\langle l, s \rangle \to \langle l', s' \rangle$ описывается формулой

$$(pc = l) \wedge \left(\bigwedge_{i=1}^{n} \big(x_i = s(x_i)\big) \right) \wedge (pc' = l') \wedge \big(x'_{i_0} = t\big) \wedge same(V \setminus \{x_{i_0}\}),$$

где pc — специальная переменная, хранящая метку текущего оператора (от англ. *program counter* — счетчик программы); запись x', где x — некоторая переменная, означает очередную версию переменной x; $same(V') \equiv \bigwedge_{x \in V'} (x' = x)$, где $V' \subseteq V$ [Кар10].

В таблице 14.7 приведены правила вычисления символического представления отношения переходов для программ на языке while. $\mathcal{R}(l\colon P, l')$ обозначает символическое представление отношения переходов программы P с начальной меткой l и конечной меткой l' (считается, что в конце исходной программы есть метка l_ω).

Таблица 14.7. Правила построения символического представления while-программ

(1)	$\mathcal{R}(l\colon \mathbf{skip}, l')$	$(pc = l) \wedge (pc' = l') \wedge same(V)$
(2)	$\mathcal{R}(l\colon x \coloneqq t, l')$	$(pc = l) \wedge (pc' = l') \wedge (x' = t) \wedge same(V \setminus \{x\})$
(3)	$\mathcal{R}(l_1\colon P_1; \ldots; l_n\colon P_n, l_{n+1})$	$\bigvee_{i=1}^{n} \mathcal{R}\,(l_i\colon P_i, l_{i+1})$
(4)	$\mathcal{R}(l\colon \mathbf{if}\ B\ \mathbf{then}\ l_1\colon P_1\ \mathbf{else}\ l_2\colon P_2\ \mathbf{end}, l')$	$\big((pc = l) \wedge B \wedge (pc' = l_1) \wedge same(V)\big) \vee \mathcal{R}(l_1\colon P_1, l') \vee$ $\big((pc = l) \wedge \neg B \wedge (pc' = l_2) \wedge same(V)\big) \vee \mathcal{R}(l_2\colon P_2, l')$
(5)	$\mathcal{R}(l\colon \mathbf{while}\ B\ \mathbf{do}\ l_1\colon P_1\ \mathbf{end}, l')$	$\big((pc = l) \wedge B \wedge (pc' = l_1) \wedge same(V)\big) \vee \mathcal{R}(l_1\colon P_1, l) \vee$ $\big((pc = l) \wedge \neg B \wedge (pc' = l') \wedge same(V)\big)$

Правила эти интуитивно понятны и не требуют особых пояснений. Заметим лишь, что в них нет аналогов конъюнкции $\bigwedge_{i=1}^{n} \big(x_i = s(x_i)\big)$, имеющейся в формуле, приведенной перед

таблицей: дело в том, что точная идентификация состояния — это как раз то, от чего в символических методах хотят уйти. Достаточно иметь ограничение на начальную конфигурацию $(pc = l_0) \wedge \chi_0$ и формулы, описывающие отношение переходов.

Отметим также, что символическое представление отношение переходов может быть пополнено следующим условием:

$\big((pc = l_\omega) \wedge (pc' = l_\omega) \wedge same(V)\big)$, где l_ω — завершающая метка программы[99].

Если все переменные имеют конечные домены (чем меньше, тем лучше), можно воспользоваться двоичной кодировкой состояний. Самый простой способ состоит в упорядочивании переменных и конкатенировании соответствующих им двоичных векторов. Для задания характеристических функций могут использоваться двоичные решающие диаграммы.

Пример 14.4

Проиллюстрируем сказанное на примере алгоритма Петерсона взаимного исключения двух процессов (см. лекцию 9). Для простоты критическая и некритическая секции состоит из одного оператора **skip**.

```
BGNᵢ: while true do
  NCSᵢ: skip; /* некритическая секция */
  SETᵢ: flagᵢ := 1; SET'ᵢ: turn := i;
  TSTᵢ: while (flag₁₋ᵢ = 1) ∧ (turn = i) do SKPᵢ: skip end;
  CRSᵢ: skip; /* критическая секция */
  RSTᵢ: flagᵢ := 0
end
ENDᵢ:
```

Выпишем переменные программы и их домены ($i \in \{0,1\}$ — идентификатор процесса):

- $Dom(pc_i) = \{BGN_i, NCS_i, SET_i, SET_i', TST_i, SKP_i, CRS_i, RST_i, END_i\}$;
- $Dom(flag_i) = \{0,1\}$;
- $Dom(turn) = \{0,1\}$.

Построим символическое представление отдельных процессов в соответствии правилами, приведенными в таблице 14.7. Результат показан в таблице 14.8.

---

[99] Если не добавить это условие, то при синхронной композиции процессов завершение какого-то одного процесса будет «вызывать» завершение всех процессов.

Таблица 14.8. Символическое представление процессов

Метка	Символическое представление
BGN_i	$\big((pc_i = BGN_i) \wedge true \wedge (pc_i' = NCS_i) \wedge same(V)\big) \vee$
NCS_i	$\big((pc_i = NCS_i) \wedge (pc_i' = SET_i) \wedge same(V)\big) \vee$
SET_i	$\big((pc_i = SET_i) \wedge (pc_i' = SET_i') \wedge (flag_i = 1) \wedge same(V \setminus \{flag_i\})\big) \vee$
SET_i'	$\big((pc_i = SET_i') \wedge (pc_i' = TST_i) \wedge (turn = i) \wedge same(V \setminus \{turn\})\big) \vee$
TST_i	$\Big((pc_i = TST_i) \wedge \big((flag_{1-i} = 1) \wedge (turn = i)\big) \wedge (pc_i' = SKP_i) \wedge same(V)\Big) \vee$
SKP_i	$\big((pc_i = SKP_i) \wedge (pc_i' = TST_i) \wedge same(V)\big) \vee$
	$\Big((pc_i = TST_i) \wedge \neg\big((flag_{1-i} = 1) \wedge (turn = i)\big) \wedge (pc_i' = CRS_i) \wedge same(V)\Big)$
CRS_i	$\big((pc_i = CRS_i) \wedge (pc_i' = RST_i) \wedge same(V)\big) \vee$
RST_i	$\big((pc_i = RST_i) \wedge (pc_i' = BGN_i) \wedge (flag_i = 0) \wedge same(V \setminus \{flag_i\})\big) \vee$
	$\big((pc_i = BGN_i) \wedge \neg true \wedge (pc_i' = END_i) \wedge same(V)\big) \vee$
END_i	$\big((pc = END_i) \wedge (pc' = END_i) \wedge same(V)\big)$

Определив двоичную кодировку значений переменных, можно построить более низкоуровневое описание — характеристическую функцию отношения переходов — и задать ее в форме двоичных решающих диаграмм.

□

Пусть $P = P_1 \parallel \cdots \parallel P_n$ — параллельная программа, в которой для каждого процесса P_i, где $i \in \{1, \ldots, n\}$, определено символическое представление отношения переходов $\mathcal{R}(P_i)$. Как построить $\mathcal{R}(P)$ — символическое представление отношения переходов программы P? Ответ зависит от используемой модели вычислений (см. лекцию 9). В первом приближении:

- $\mathcal{R}(P) = \bigwedge_{i=1}^{n} \mathcal{R}(P_i)$ для синхронной композиции;
- $\mathcal{R}(P) = \bigvee_{i=1}^{n} \mathcal{R}(P_i)$ для асинхронной композиции.

Для синхронной композиции имеется нюанс, связанный с условиями вида $same(V')$. Поясним на примере. В том виде, как правила определены сейчас, синхронное исполнение операторов $x_1 := x_1 + 1$ и $x_2 := x_2 + 1$ описывается заведомо ложной формулой

$$\big((x_1' = x_1 + 1) \wedge (x_2' = x_2)\big) \wedge \big((x_2' = x_2 + 1) \wedge (x_1' = x_1)\big),$$

Дело в том, что условия $same(V')$, заданные в правилах, не рассчитаны на действительно параллельное исполнение операторов. Чтобы избежать этой аномалии можно использовать более общий подход. Условия $same(V')$ удаляются из описания операторов. Вместо этого для каждой переменной x делается следующее:

- составляется условие $def(x)$, сигнализирующее о присваивании в переменную x (обычно такая формула имеет вид $\left(pc_{i_1} = l_{i_1}\right) \vee \ldots \vee \left(pc_{i_k} = l_{i_k}\right)$);
- к символическому представлению отношения переходов программы добавляется конъюнктивный член $\bigwedge_{x \in V}\left(def(x) \vee (x = x')\right)$.

14.3.2. Символическое представление модели требований

На прошлой лекции был рассмотрен алгоритм построения автомата Бюхи для заданной формулы LTL. Вместо построения автомата в явном виде можно построить его символическое представление. Суть алгоритма от этого не меняется.

14.3.3. Проверка соответствия моделей программы и требований

Символическое представление синхронной композиции $B_M \otimes B_{\neg\varphi}$ делается по правилам, указанным выше.

Для проверки языка $\mathcal{L}_{B_M \otimes B_{\neg\varphi}}$ на пустоту, т.е. проверки отсутствия допускающих циклов, можно использовать алгоритмы вычисления *оболочки компонент сильной связности*, или *SCC-оболочки* (от англ. *SCC — Strongly Connected Component*).

SCC-оболочкой системы переходов называется множество состояний нетривиальных компонент сильной связности, достижимых из начальных состояний. Нас интересуют только те компоненты, которые содержат допускающие состояния, — так называемые *допускающие компоненты*[100]. Часто вычисляется не сама оболочка, а ее аппроксимация: если множество пусто, допускающих компонент нет; в противном случае допускающие компоненты есть, однако вычисленное множество может содержать лишние состояния [Roz10]. Алгоритм, описанный ниже, базируется на работе [Kes98], но игнорирует условия справедливости.

Пусть S — множество состояний синхронной композиции $B_M \otimes B_{\neg\varphi}$, а $R \subseteq S \times S$ — отношение переходов. Для удобства будем считать, что синхронная композиция является не обобщенным, а обычным автоматом Бюхи, т.е. содержит одно множество допускающих

[100] В общем случае речь идет о *справедливых* компонентах сильной связности — компонентах, содержащих допускающие циклы, удовлетворяющие к тому же всем имеющимся условиям справедливости.

состояний $\mathcal{F}$. Алгоритм описан так, как если бы он выполнялся над множествами представленными явно (перейти к символическому варианту не составляет труда). В конце работы алгоритма множество S_{new} содержит все достижимые допускающие состояния, лежащие в нетривиальных компонентах сильной связности.

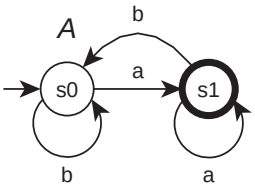
```
S_old := ∅;
S_new := R*(S_0);       /* множество достижимых состояний */
while S_new ≠ S_old do /* пока процесс не стабилизируется */
  S_old := S_new;
  S_new := R*(S_new ∩ F);  /* состояния, достижимые из
                              допускающих состояний */
  while S_new ≠ (S_new ∩ R(S_new)) do  /* оставляются состояния, ведущие
    S_new := S_new ∩ R(S_new)              в то же множество состояний */
  end
end;
verdict := (S_new = ∅)  /* если множество пусто,
                           допускающих циклов нет */
```

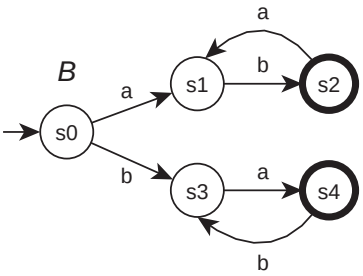
Для построения контрпримера может использоваться следующий алгоритм, находящий кратчайший путь между состояниями из множества S_{source} и состояниями из множества S_{target} (предполагается, что такой путь существует).

```
π := ∅;          /* конструируемый путь сначала пуст */
S_start := S_source;  /* множество начальных состояний */
while (S_start ∩ S_target) = ∅ do
  S_end := R^-1(S_target)  /* множество предшественников целевых состояний */
  while (S_start ∩ S_end) = ∅ do
    S_end := R^-1(S_end)  /* множество предшественников */
  end;
  s := choose(S_start ∩ S_end); /* выбирается некоторое общее состояние s */
  π := π · {s};  /* состояние s добавляется в путь */
  S_start := R(s)  /* начальными состояниями становятся последователи s */
end
```

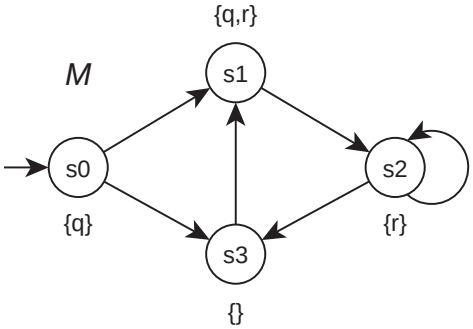
14.4. Вопросы и упражнения

1. В чем отличие символических методов проверки моделей от классических?
2. Постройте символические представления для следующих конечных автоматов (под символическим представлением автомата понимаются логические формулы, характеризующие множество состояний, множество начальных состояний, множество завершающих состояний, функцию переходов):

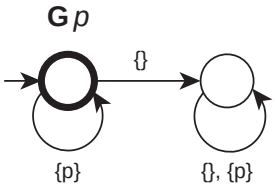


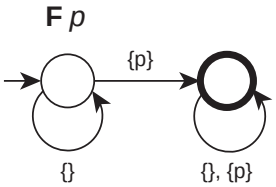


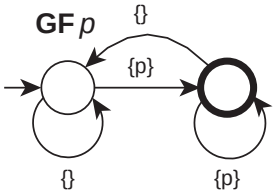
3. Постройте символическое представления для следующей структуры Крипке:

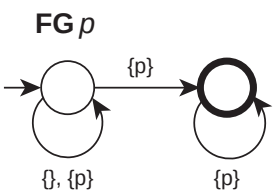


4. Постройте символические представления для следующих автоматов Бюхи:









5. Постройте символическое представление программы, реализующей алгоритм Евклида (программы P из лекции 1).

6. Постройте символическое представление (асинхронной) параллельной программы $P = P_1 \parallel P_2$, где процессы P_1 и P_2 определяются, как показано ниже [Кар10]:

Процесс P_1	Процесс P_2

while true **do**
  A0: x := 0;
  A1: y := 1
end
	while true **do**
	  B0: x := 1;
	  B1: y := 0
	end

7. В предыдущей задаче поменяйте порядок операторов, помеченных метками $A0$ и $A1$, и постройте символическое представление синхронной композиции процессов P_1 и P_2.
8. Задайте двоичную кодировку состояний для алгоритма Петерсона (число процессов равно двум) и постройте характеристическую функцию отношения переходов.
9. Постройте двоичные решающие диаграммы (ROBDD) для всех возможных булевых функций от двух переменных для всех возможных порядков переменных.
10. Как построить двоичную решающую диаграмму, представляющую отрицание функции, по диаграмме этой функции?
11. Предложите эффективный способ реализации с помощью двоичных решающих диаграмм операций квантификации булевых функций:
    a. $\exists x_i f(x_1, \dots, x_n) = f(x_1, \dots, x_{i-1}, 0, x_{i+1}, x_n) \vee f(x_1, \dots, x_{i-1}, 1, x_{i+1}, x_n)$;
    b. $\forall x_i f(x_1, \dots, x_n) = f(x_1, \dots, x_{i-1}, 0, x_{i+1}, x_n) \wedge f(x_1, \dots, x_{i-1}, 1, x_{i+1}, x_n)$.
12. Определите рекурсивно процедуру $Apply$ для следующих булевых операций:
    a. $\oplus$ (сумма по модулю 2);
    b. $\rightarrow$ (импликация).
13. Постройте ROBDD функции $(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$ для следующих порядков переменных [Bry86]:
    a. $x_1 \prec y_1 \prec x_2 \prec y_2 \prec x_3 \prec y_3$;
    b. $x_1 \prec x_2 \prec x_3 \prec y_1 \prec y_2 \prec y_3$.
14. Постройте ROBDD функции $\neg(x_1 \oplus y_1) \wedge \neg(x_2 \oplus y_2) \wedge \neg(x_3 \oplus y_3)$ для двух разных порядков переменных (см. предыдущую задачу).
15. Постройте двоичную решающую диаграмму для 3-х битного сумматора.
16. Докажите, что алгоритмическая сложность $Apply(F \circ G)$ составляет $O(|F| \cdot |G|)$.

Лекция 15. Использование формальных методов в тестировании

Верификация программ не означает создания программ, лишенных ошибок. И здесь нет чудес. Математические доказательства тоже могут быть ошибочными. Поэтому хотя верификация может облегчить тестирование, она не может отменить его.

Ф. Брукс.
Серебряной пули нет: сущность и акциденция в программной инженерии

Рассматриваются вопросы применения формальных методов к тестированию программ. Подходы, в которых формальные модели и техники используются для решения задач тестирования называются методами тестирования на основе моделей. Для них характерны автоматическая проверка правильности поведения программ в ответ на тестовые воздействия, наличие формализованных критериев полноты тестирования, нацеленность генерации тестов на тестовое покрытие. Описываются некоторые распространенные подходы: тестирование на основе программных контрактов, символическое исполнение и конколическое (конкретно-символическое) тестирование, генерация тестов с помощью обхода графа состояний и др.

15.1. Тестирование vs формальная верификация

Напомним, что *тестированием* называется метод верификации, основанный на анализе результатов исполнения программы в некоторых ситуациях. Процедуры, описывающие процесс создания этих ситуаций, и проверки, которые необходимо выполнить над полученными результатами, называются тестами [Кул08]. В отличие от формальных методов, в которых анализ программы производится для всех возможных вычислений, тестирование всегда имеет дело с ограниченным множеством вариантов; собственно, одной из главных задач тестирования является создание репрезентативной выборки таких вариантов — *тестового набора*. С другой стороны, тестирование имеет дело с реальной системой, а не ее моделью, которая, как известно, может быть неадекватной (см. лекцию 11). Достоинства и недостатки методов тестирования и формальной верификации сведены в таблицу 15.1. Более детальное сравнение методов делается в обзоре [Кул08].

Таблица 15.1. Сравнение методов тестирования и формальной верификации

Тестирование	Формальная верификация
+ Проверяется реальная система	+ Проверяются все возможные сценарии
+ Проверка осуществляется в реальных условиях	+ Проверяется класс возможных реализаций
– Выполняется на поздних этапах разработки	– Проверяется модель, а не реальная система
– Проверяется ограниченный набор сценариев	– Проверяется ограниченный набор свойств
– Высокая трудоемкость разработки тестов	– Высокая вычислительная сложность
– Необходимость большого штата тестировщиков	– Высокие требования к квалификации верификаторов

Методы тестирования и формальной верификации не являются взаимоисключающими, а напротив, дополняют друг друга и могут использоваться совместно. Возможна следующая организация процесса создания ПО, в целом соответствующая концепции *разработки на основе моделей* (*model-driven development / engineering*) [Rai12]:

1. создание модели системы;
2. формальная верификация модели;
3. реализация системы (на основе модели);
4. тестирование системы (на основе модели).

Более того, формальные техники могут применяться для решения задач тестирования: генерации тестовых воздействий (последовательностей стимулов); проверки корректности поведения (реакций, выдаваемых в ответ на стимулы); оценки полноты тестирования (репрезентативности набора тестов). Методы тестирования, в которых задействуются формальные модели и техники их анализа, называются *методами тестирования на основе моделей (MBT, Model-Based Testing*) [Utt12]. Перед тем как рассмотреть некоторые основополагающие практики этого типа, сделаем небольшой экскурс в историю.

Тестирование на основе моделей в том или ином виду существует с 1970-ых гг., однако первоначально применялось лишь в таких областях, как проектирование аппаратуры и телекоммуникационных протоколов [Pet15]. Постепенно пришло осознание, что модели (формальные спецификации) могут использоваться для тестирования более широкого класса систем. В конце 1980-ых гг. стали появляться работы, предлагающие методы генерации тестов по спецификациям, вроде статьи [Ost88]. Это были полуавтоматические подходы, позволяющие генерировать описания для дальнейшей ручной разработки тестов по ним. Полноценные инструменты стали зарождаться в середине 1990-ых гг. Одними из первых тех-

нологий тестирования на основе моделей стали KVEST [Bur99] и UniTESK [Бур03b], разработанные в ИСП РАН. Оба подхода базировались на программных контрактах и техниках обхода модельных графов состояний. Позже появился близкий по возможностям инструмент SpecExplorer [Bla05] от Microsoft Research, основанный на машинах абстрактных состояний (ASM, Abstract State Machines).

15.2. Тестирование программ без состояния

Начнем со случая последовательных программ. Для начала ограничимся *программами преобразования данных*, поведение которых зависит только от значений входных переменных (см. лекцию 2). Такие программы часто называют *программами без состояния* — их поведение не зависит от предыстории запусков. Техники, описанные ниже, могут использоваться для автоматизации тестирования отдельных функций (методов).

15.2.1. Тестирование на основе программных контрактов

Для спецификации условий корректности программ преобразования данных используются *программные контракты — пред-* и *постусловия* (см. лекцию 3). Контракты позволяют решить следующие задачи:

- *проверка допустимости тестового воздействия*: если предусловие ложно, тестовое воздействие недопустимо и не должно подаваться на программу;
- *проверки корректности результата*: если для допустимого тестового воздействия программа завершилась, но постусловие ложно, результат считается ошибочным.

Это открывает возможность для использования автоматических генераторов тестовых воздействий. Самым простым подходом такого типа является *вероятностное тестирование*. Генерируются случайные значения входных переменных: если значения не удовлетворяет предусловию, воздействие отбрасывается; в противном случае оно подается на программу. По завершению программы (если, конечно, она не зацикливается) проверяется соответствие результата постусловию: если постусловие ложно, фиксируется ошибка; в противном случае процесс повторяется. Пусть P — тестируемая программа, $\langle \varphi, \psi \rangle$ — ее спецификация, N — ограничение на число воздействий.

```
verdict := true;                /* вердикт */
i := 0;                         /* счетчик тестовых воздействий */
```

```
while verdict ∧ i < N do
  x := random()                /* генерация случайного воздействия */
  if φ then                    /* проверка предусловия */
    P;                         /* вызов тестируемой программы */
    if ¬ψ then                 /* проверка постусловия */
      verdict := false         /* фиксация ошибки */
    end;
    i := i + 1
  end
end
```

Неоспоримым достоинством вероятностного тестирования является его простота[101]. Недостатки очевидны. С одной стороны, многие тестовые воздействия приводят к одинаковому поведению тестируемой программы, т.е. к исполнению одних и тех же путей в графе потока управления. С другой стороны, вероятность реализации некоторых ситуаций, так называемых *граничных ситуаций*, может быть крайне низкой (к таким ситуациям относятся, в частности, проходы по путям, содержащим условия вида $x = c$, где c— константа).

15.2.2. Тестирование черного ящика

Ранее, в лекции 3, были определили такие характеристики методов верификации, как значимость и полнота: метод называется *значимым*, если из того, что вынесен отрицательный вердикт, следует, что в программа ошибочна; *полным* — если из того, что вынесен положительный вердикт, следует, что программа корректна. В этих терминах знаменитое высказывание Эдсгера Дейкстры (1930-2002) о том, что тестирование может служить для доказательства наличия ошибок, но никогда не докажет их отсутствия, можно перефразировать так: тестирование является значимым методом, но, вообще говоря, неполным. Когда говорят о полноте тестирования, слово «полнота» часто употребляется в ином, менее формальном смысле — как синоним к адекватности, приемлемости и т.п. Формулировка критериев адекватности тестирования, как правило, носит эвристический характер.

Вероятностное тестирование в том виде, в котором оно было представлено выше, является частным случаем *тестирования черного ящика*. Так называется класс методов построения

[101] Некоторые трудности может привнести генерация сложно структурированных данных, но этот вопрос мы здесь опускаем.

тестов, не использующих информацию о структуре программы, т.е. исходный или бинарный код [Bei95]. Ниже описывается еще один подход этого типа, основанный на использовании *функциональных моделей тестового покрытия*.

Пусть задана некоторая программа. *Тестовым покрытием*, или просто *покрытием*, называется конечное множество *тестовых ситуаций* — классов эквивалентности возможных вычислений. *Моделью покрытия* называется способ описания покрытия по программе (реализации) и требованиям к ней (спецификации). Модель покрытия называется *функциональной*, если она использует спецификацию, но не использует реализацию.

На основе программных контрактов можно определить следующие функциональные модели тестового покрытия:

- *покрытие ветвей функциональности* (*branch coverage*);
- *покрытие значений истинности элементарных условий* (*condition coverage*).

Ветви функциональности — это тестовые ситуации, соответствующие различным вариантам поведения программы. В языке ACSL ветвь функциональности задаются с помощью конструкции **behavior** b: **assumes** φ ..., где b — имя ветви, а φ — ее условие (см. лекцию 4). В классическом случае ветви функциональности *разделимы*, т.е. имеют попарно несовместные условия, а их множество *полно* — из истинности предусловия следует истинность условия хотя бы одной ветви. В языке ACSL это требование может быть нарушено: в этом случае можно использовать *кросс-покрытие* — комбинации ветвей.

Пример 15.1

В представленном ниже фрагменте спецификации функции дихотомического поиска в упорядоченном массиве (см. лекцию 4) выделены две ветви: первая имеет тривиальное условие; вторая требует упорядоченности массива.

```
behavior success:
  ...
behavior failure:
  assumes \forall integer i, integer j;
    0 <= i < j <= n-1 ==> x[i] <= x[j];
  ...
```

□

Покрытие значений истинности элементарных условий — это класс моделей покрытия, определяемых в терминах комбинаций значений истинности элементарных логических формул (атомов). Есть несколько моделей этого класса: *покрытие всех возможных комбинаций*; *покрытие попарных комбинаций* (*pairwise testing*); *MC/DC* (*Modified Condition/Decision Coverage*), требующая покрытия всех таких ситуаций, в которых значение истинности атома (condition) влияет на значение истинности формулы в целом (decision)[102].

Пример 15.2

Пусть предусловие имеет вид $\big((y = 0) \to (x = 0)\big) \wedge \big((x > 3) \vee (y < x)\big)$. Формула содержит четыре атома: $(y = 0)$, $(x = 0)$, $(x > 3)$ и $(y < x)$. Всего возможны $2^4 = 16$ комбинации значений истинности; нас устраивают только те из них, для которых предусловие истинно. Получаем 9 вариантов[103], часть из которых нереализуемы из-за семантических взаимосвязей между атомами (такие комбинации зачеркнуты):

1. ~~$(true, true, true, true)$~~;
2. $(true, true, true, false)$;
3. ~~$(true, true, false, true)$~~;
4. ~~$(false, true, true, true)$~~;
5. $(false, true, true, false)$;
6. $(false, true, false, true)$;
7. $(false, false, true, true)$;
8. $(false, false, true, false)$;
9. $(false, false, false, true)$.

□

Покрытие не только задает критерий полноты тестирования, но и может использоваться для *направленной генерации тестовых воздействий* (*coverage-driven verification*). Для

[102] Критерий тестового покрытия MC/DC используется при тестировании авионики (стандарт DO-178B).

[103] По сути, ситуации соответствуют конъюнктивным членам совершенной ДНФ, построенной по формуле предусловия.

этого используются инструменты разрешения ограничений: SAT- и SMT-решатели (см. лекцию 8). Например, для ситуации $(false, false, true, false)$, соответствующей формуле

$$\neg(y = 0) \wedge \neg(x = 0) \wedge (x > 3) \wedge (y < x),$$

с использованием решателя может быть построено воздействие $\{x \mapsto 4, y \mapsto 3\}$. В реальной жизни ограничения могут быть гораздо сложнее и вовлекать указатели, результаты работы с файлами, базами данных и т.п. (рассмотрение этих вопросов выходит за рамки этого пособия).

15.2.3. Тестирование белого ящика

Тестирование белого ящика предполагает использование информации о структуре программы для генерации тестовых воздействий и оценки полноты тестирования [Bei90]. Модели тестового покрытия, определенные на основе реализации (исходного или бинарного кода), называются *структурными*. Наиболее известными моделями этого типа являются:

- *покрытие операторов* (*statement coverage*);
- *покрытие ветвлений* (*branch coverage*);
- *покрытие путей* (*path coverage*).

В терминах блок-схем (см. лекцию 5) покрытие операторов — это покрытие узлов **assign**; покрытие ветвлений — покрытие дуг, исходящих из узлов **test**; покрытие путей — покрытие последовательностей смежных дуг между узлами **start** и **halt**. Для отслеживания возникновения той или иной ситуации в процессе тестирования программа инструментируется.

Рассмотрим, к примеру, покрытие путей. Множество возможных путей в графе потока управления строится с помощью *поиска в глубину* (см. лекцию 11). Если в графе есть циклы, ограничивают число итераций. Для каждого пути с помощью *символического исполнения* может быть построена *формула пути* — условие, истинность которого в начальном состоянии гарантирует исполнение программы по этому пути. Для построения формулы пути может использоваться рассмотренный нами *метод обратных подстановок* (см. лекции 3 и 5), хотя обычно для этих целей применяют SSA-представление программы (см. лекцию 9).

Пример 15.3

Для программы целочисленного деления ограничим число итераций в цикле числом 2 и построим множество возможных путей, используя поиск в глубину (см. рис. 15.1).

```
q := 0;
r := a;
while r ≥ b do
  q := q + 1;
  r := r - b
end
```

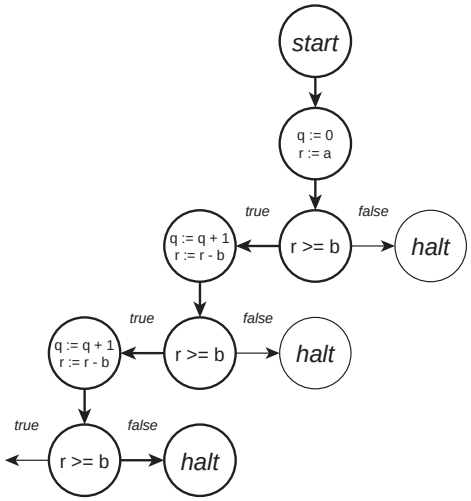


Рисунок 15.1. Ограниченный поиск в глубину в программе целочисленного деления

Рассмотрим путь, в котором исполняются ровно две итерации цикла (на рисунке он выделен жирными дугами). Этот путь можно записать в форме линейной аннотированной программы.

`q := 0;`	
`r := a;`	$(a \geq b) \wedge (a \geq 2b) \wedge (a < 3b)$
`{r ≥ b}`	$(r \geq b) \wedge (r \geq 2b) \wedge (r < 3b)$
`q := q + 1;`	
`r := r - b;`	$\big((r-b) \geq b\big) \wedge \big((r-b) < 2b\big) \equiv (r \geq 2b) \wedge (r < 3b)$
`{r ≥ b}`	$(r \geq b) \wedge (r < 2b)$
`q := q + 1;`	
`r := r - b`	$\big((r-b) < b\big) \equiv (r < 2b)$
`{r < b}`	$(r < b)$

Применяя метод обратных подстановок, получим формулу

$$(a \geq b) \wedge (a \geq 2b) \wedge (a < 3b).$$

Совмещая полученную формулу с предусловием $(a \geq 0) \wedge (b > 0)$, получим

$$(b > 0) \wedge (a \geq 2b) \wedge (a < 3b).$$

По этой формуле SMT-решатель может построить, например, воздействие $\{a \mapsto 12, b \mapsto 5\}$.

□

При символическом исполнении сложных программ могут возникать ограничения, которые не по силам решателям. В последнее время получил распространение подход, называемый *конколическим тестированием* (*concolic testing*), который совмещает конкретное и символическое исполнение программы (*concolic* = *conc*rete + symb*olic*). Концепция подхода была описана в 2005 г. Патрисом Годефруа (Patrice Godefroid), Нильсом Кларлундом (Nils Klarlund) и Каушиком Сеном (Koushik Sen) [God05]. Термин «конколическое тестирование» впервые появился в работе [Sen05].

Для реализации конколического тестирования программа инструментируется таким образом, чтобы в процессе ее исполнения строилась формула пути. В несколько упрощенной форме подход состоит из следующих шагов.

1. Генерируется случайное тестовое воздействие.
2. Тестовое воздействие подается на программу.
3. При исполнении программы строится формула пути.
4. Если путь новый, он запоминается.
5. Выбирается один из пройденных путей:
    a. Одно из условий формулы пути заменяется на его отрицание.
    b. Условия, следующие за измененным (описывающие исполнение программы после соответствующего ветвления), удаляются.
    c. Если полученная формула уже встречалась:
        i. Выбирается другое условие или другой путь (осуществляется переход на шаг 2 или шаг 5).
        ii. Если не удается построить новой формулы, тестирование завершается.
6. Построенное ограничение передается решателю:
    a. Если решатель не справился, генерируется случайное тестовое воздействие.
    b. В противном случае:

i. Если ограничение невыполнимо, выбирается другое условие или путь.

ii. Если ограничение выполнимо, полученный результат используется как тестовое воздействие.

7. Осуществляется переход на шаг 2.

Ограничения, описывающие вычисления программы, по-прежнему могут быть сложными (это определяется самой программой, а не способом ее тестирования), однако рандомизация позволяет преодолевать многие трудности. В конколическом тестировании широко используются методы абстрактной интерпретации (см. лекцию 7), что позволяет строить более простые ограничения, решения которых с большой вероятностью будут решениями исходных ограничений. Если известно предусловие программы, оно используется при генерации тестовых воздействий и построении формул. Если в программе есть циклы, глубина путей ограничивается искусственно.

Пример 15.4

Рассмотрим конколическое тестирование программы целочисленного деления (см. пример выше). Первым делом генерируется случайное тестовое воздействие. Предположим, это $\{a \mapsto 5, b \mapsto 3\}$. Воздействие подается на программу:

`q := 0;`	$q = 0$
`r := a;`	$r = 5$
`{r ≥ b}`	$true$
`q := q + 1;`	$q = 1$
`r := r - b`	$r = 2$
`{r ≥ b}`	$false$

С учетом предусловия получаем следующую формулу пути:

$$(b > 0) \wedge (a \geq b) \wedge (a < 2b).$$

В этой формуле берется последнее условие и заменяется на его отрицание:

$$(b > 0) \wedge (a \geq b) \wedge (a \geq 2b) \equiv (b > 0) \wedge (a \geq 2b).$$

Полученное ограничение решается. В результате получается новое воздействие, например, $\{a \mapsto 123, b \mapsto 45\}$. Процесс продолжается.

□

15.3. Тестирование программ с состоянием

Обратимся теперь к *программам с состоянием*, т.е. к программам, поведение которых зависит не только от значений входных переменных, но и от предыстории запусков (предыстории взаимодействия с окружением). К этому типу относится большинство реальных программ и библиотек, включая абстрактные типы данных, реагирующие системы и др. Техники, описанные ниже, могут использоваться для тестирования систем функций над общими переменными (модулей и классов).

15.3.1. Обход графа состояний программы

Естественным способом тестирования программ с состоянием является исследование их пространств состояний (см. лекцию 11). В отличие от верификации моделей в тестировании реальных программ есть принципиальное ограничение — программу трудно перевести в заданное состояние. В связи с этим исследование пространства состояний осуществляется путем обхода графа состояний — построения маршрута, содержащего все дуги графа (переходы).

Предположим, что тестовые воздействия занумерованы от 1 до n: номер соответствует тестовой ситуации в рамках некоторой модели покрытия. Пусть в каждом состоянии допустимы все воздействия. Работу генератора тестовых воздействий можно представить следующим образом:

```
S := ∅;                          /* множество состояний */
T := ∅;                          /* множество переходов */
F := ∅;                          /* множество пройденных переходов */
while true do                    /* главный цикл генератора тестов */
  s := getState()                /* вычисляется текущее состояние */
  if s ∉ S then                  /* если состояние не встречалось ранее, */
    S := S ∪ {s};                   оно сохраняется */
    T[s] := {1, ..., n};         /* множество исходящих переходов */
    F[s] := ∅                    /* множество пройденных переходов */
  end;
  if F[s] ≠ T[s] then            /* если есть непройденные переходы,
    x := choose(T[s] \ F[s]);       выбирается один из них */
    applyStimulus(x);            /* подается тестовое воздействие */
    F[s] := F[s] ∪ {x}           /* переход отмечается как пройденный */
  else                           /* если нет непройденных переходов,
    p := getPath();                 строится путь в состояние,
                                    где есть непройденные переходы */
    if p = ∅ then                /* если все переходы пройдены,
      break                         тестирование прекращается */
    end;
```

```
      for x ∈ p do                    /* подаются тестовые воздействия,
        applyStimulus(x)                 заданные в пути */
      end
    end
  end
```

По сути, это поиск в глубину, цель которого — найти состояние, в которой нет непройденных переходов. Для такой вершины запускается вложенный поиск ($getPath$), возвращающий путь из текущего состояния в состояние, в котором есть непройденные переходы. Вложенный поиск может быть реализован разными способами: как в глубину, так и в ширину. Для применимости метода достаточно, чтобы граф состояний был *сильно связным* — имел одну компоненту сильной связности (см. лекцию 12) и *детерминированным* — для каждого состояния s и тестового воздействия x должно существовать не более одного перехода вида (s, x, s'), где s' — некоторое состояние [Бур03а].

15.3.2. Абстракция состояний программы

Даже если ограничиться последовательными программами, число состояний может быть огромным. В этом пособии мы сознательно избегали динамических структур данных; между тем их использование делает число возможных состояний необозримым. Для тестирования необходимо искусственно ограничивать пространство поиска. Как и в формальной верификации, для этого используются методы абстракции. Поясним сказанное на простом примере.

Пример 15.5

Рассмотрим очередь, элементами которой являются целые числа. Над очередью определены две операции: $enqueue(x)$ — добавить элемент x в конец очереди; $dequeue()$ — извлечь элемент из головы непустой очереди. Используем следующее отношение эквивалентности для построения абстрактной модели: две очереди эквивалентны тогда и только тогда, когда их размеры совпадают. Зададим дополнительное ограничение — будем считать, что число элементов в очереди не превосходит двух. На рис. 15.2 показан фрагмент графа состояний очереди и граф состояний абстрактной модели.

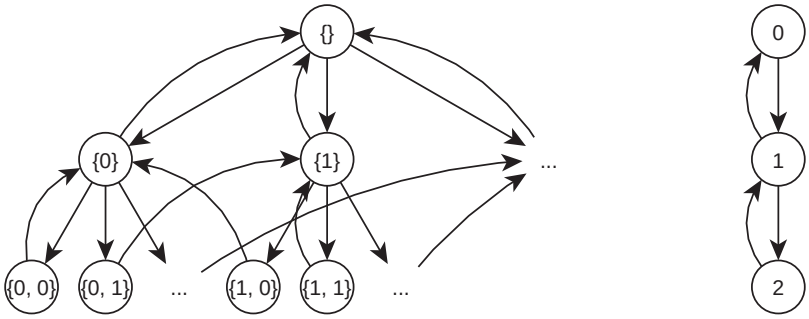


Рисунок 15.2. Фрагмент графа состояний очереди (слева) и его абстракция (справа)

□

15.4. Тестирование реагирующих систем и параллельных программ

Для описания свойств реагирующих систем используется темпоральная логика LTL. На прошлых лекциях мы рассмотрели, как формально проверить истинность формулы LTL на модели программы — структуре Крипке с конечным числом состояний. Можно ли LTL-спецификации использовать для динамической проверки поведения систем? Исследования по этой теме ведутся в рамках такого направления, как *мониторинг времени выполнения* (*runtime monitoring*) [Bau11]. Идея понятна по недетерминированному автомату Бюхи для заданной формулы LTL (см. лекцию 13) строится детерминированный автомат (монитор), в динамике вычисляющий вердикт: $true$, $false$ или $undef$. При построении детерминированных автоматов из недетерминированных применяют методы, близкие алгоритмам построения ДКА по НКА [Сер12].

Методы проверки моделей могут применяться для тестирования параллельных программ. Для этого применяют техники *управляемого выполнения*. Программа инструментируется: в точки, в которых осуществляется взаимодействие процесса с окружением или другими процессами программы (обращение к разделяемой переменной, прием или посылка сообщения и т.п.), добавляется управляющий код, который может приостанавливать выполнение процесса. Решения о приостановке тех или иных процессов принимаются планировщиком, целью которого является обход пространства состояний (систематический перебор возможных вариантов исполнения программы). Примером инструмента, работающего по этой схеме, является Java PathFinder (JPF) — средство проверки Java-программ, работающее

на уровне байт-кода (в JPF используется виртуальная машина JVM со специальным планировщиком) [JPF]. Возможен и другой вариант, когда планировщик не отслеживает состояния процессов, а воспроизводит трассы, полученные при верификации модели системы, например, с помощью инструмента SPIN [Spin].

15.5. Вопросы и упражнения

1. Дайте определение тестирования. Сравните тестирование с формальными методами верификации. В чем достоинства и недостатки каждого из подходов?
2. Что понимается под тестированием на основе моделей? Приведите примеры подходов.
3. Для решения каких задач тестирования можно использовать программные контракты (пред- и постусловия)?
4. Определите следующие понятия: «модель тестового покрытия», «функциональное покрытие», «структурное покрытие». Приведите примеры функциональных и структурных моделей покрытия.
5. В чем разница между тестированием черного ящика и тестированием белого ящика?
6. Опишите схему использования символического исполнения программ для генерации тестов. В чем трудности такого подхода и как их можно преодолеть?
7. Чем различаются методы исследования пространств состояний, используемые в тестировании и проверке моделей?
8. Для решения каких задач тестирования можно использовать LTL-спецификации и методы проверки моделей?
9. Пусть M — число тестовых ситуаций. Предположим, что генератор обеспечивает равномерное распределение тестов по ситуациям, причем тестовые воздействия независимы друг от друга. Оцените величину вероятности достижения 100% покрытия в зависимости от N — длины тестовой последовательности.
10. Для приведенной ниже программы, реализующей алгоритм Евклида, выпишите формулы для всех вычислений, включающих не более двух итераций цикла. Для каждой из них предложите значения входных переменных a и b:

```
x := a;
y := b;
```

```
while x ≠ y do
  if x > y then
    x := x − y
  else
    y := y − x
  end
end;
c := x
```

11. Предложите метод абстракции для такой структуры данных, как двоичное дерево.

12. Напишите программу (на удобном вам языке программирования), строящую обход сильно связного детерминированного ориентированного графа.

Используемые сокращения и обозначения

Сокращения

ДКА	детерминированный конечный автомат
ДНФ	дизъюнктивная нормальная форма
КНФ	конъюнктивная нормальная форма
НКА	недетерминированный конечный автомат
НОД	наибольший общий делитель
НОУ	наибольший общий унификатор
ПНФ	предваренная нормальная форма
ПО	программное обеспечение
ССФ	сколемовская стандартная форма
СУБД	система управления базами данных
ASM	машина абстрактных состояний (abstract state machine)
AD	упрощенная (абстрактная) модель (abstract domain)
AP	элементарные высказывания (atomic propositions)
BDD	двоичная решающая диаграмма (binary decision diagram)
BFS	поиск в ширину (breath-first search)
BMC	ограниченная проверка моделей (bounded model checking)
CDV	верификация, управляемая покрытием (coverage-driven verification)
CEGAR	уточнение абстракции на основе контрпримеров (counter-example-guided abstraction refinement)
CTL	темпоральная логика ветвящегося времени (computational tree logic)
DFS	поиск в глубину (depth-first search)
DPLL	Дэвис-Патнем-Логеман-Лавленд (Davis–Putnam–Logemann–Loveland)[104]
LTL	темпоральная логика линейного времени (linear-time temporal logic)
MBT	тестирование на основе моделей (model-based testing)
MC/DC	модифицированное покрытие условий и решений (modified condition/decision coverage)
NP	недетерминированно полиномиальный (non-deterministic polynomial)[105]
ROBDD	сокращенная упорядоченная двоичная решающая диаграмма (reduced ordered binary decision diagram)

[104] Семейство алгоритмов решения задачи выполнимости КНФ.

[105] Класс задач, допускающих решение за полиномиальное время на недетерминированной машине Тьюринга.

SAT	выполнимость (satisfiability)
SCC	компонента сильной связности (strongly connected component)
SMT	выполнимость с учетом теорий (satisfiability modulo theories)
SP	сильнейшее постусловие (strongest postcondition)
SSA	представление с однократными присваиваниями (static single assignment)
TC	условие завершимости (termination condition)
UNSAT	невыполнимость (unsatisfiability)
VC	условие верификации (verification condition)
WP	слабейшее предусловие (weakest precondition)

Обозначения

Множества и отношения

$\emptyset$ или $\{\}$	пустое множество
$\{x_1, x_2, \dots\}$	множество, состоящее из элементов $x_1, x_2, \dots$
$(x_1, \dots, x_n)$	кортеж (n-ка) из элементов $x_1, \dots, x_n$
$\{x_i\}_{i=1}^{n}$	последовательность из элементов $x_1, \dots, x_n$
$x \in X$ ($x \notin X$)	принадлежность (непринадлежность) элемента x множеству X
$X \subseteq Y$ ($X \nsubseteq Y$)	включение (невключение) множества X во множество Y
$X \subset Y$	строгое включение множества X во множество Y ($X \subseteq Y$ и $X \neq Y$)
$\lvert X \rvert$	мощность конечного множества
$\overline{X}$	дополнение множества X до заданного надмножества
$X \cap Y$	пересечение множеств X и Y
$X \cup Y$	объединение множеств X и Y
$X \setminus Y$	разность множеств X и Y
$X \times Y$	декартово произведение множеств X и Y
$X \to Y$	отображение множества X во множество Y
X^n	n-ая декартова степень множества X (множество всех слов длины n в алфавите X)
2^X	множество всех подмножеств множества X
$\langle X, \prec \rangle$	множество X с заданным на нам отношением частичного порядка $\prec \subseteq X^2$ (частично упорядоченное множество)
R^*	рефлексивное и транзитивное замыкание бинарного отношения R
$Im(R)$	образ бинарного отношения R: $\{ y \in S \mid \exists x((x, y) \in R) \}$
$Dom(R)$	домен бинарного отношения R: $\{ x \in S \mid \exists y((x, y) \in R) \}$
$X \lhd R$	ограничение домена бинарного отношения R: $\{(x, y) \in R \mid x \in X\}$
$R \rhd Y$	ограничение образа бинарного отношения R: $\{(x, y) \in R \mid y \in Y\}$
$R(X)$	прямой образ бинарного отношения R: $Im(X \lhd R)$
$R^{-1}(Y)$	обратный образ бинарного отношения R: $Dom(R \rhd Y)$

Логика

$true$	истина
$false$	ложь
$undef$	неопределенное значение
$\frac{\varphi_1, \ldots, \varphi_n}{\psi}$	правило вывода: вывод формулы ψ из формул $\varphi_1, \ldots, \varphi_n$
$\vdash \varphi$ ($\nvdash \varphi$)	доказуемость (недоказуемость) формулы φ в заданной дедуктивной системе
$\vDash \varphi$ ($\nvDash \varphi$)	общезначимость (необщезначимость) формулы φ
$I \vDash \varphi$ ($I \nvDash \varphi$)	истинность (ложность) формулы φ в интерпретации I
$\neg\varphi$ или $\overline{\varphi}$	отрицание формулы φ
$\varphi \wedge \psi$	конъюнкция формул φ и ψ (И)
$\varphi \vee \psi$	дизъюнкция формул φ и ψ (ИЛИ)
$\varphi \oplus \psi$	сумма по модулю 2 формул φ и ψ (исключающее ИЛИ)
$\varphi \to \psi$	импликация из формулы φ формулы ψ (если φ, то ψ)
$\varphi \leftrightarrow \psi$	эквивалентность формул φ и ψ (φ тогда и только тогда, когда ψ)
$\forall x(\varphi)$	универсальная квантификация формулы φ по (свободной) переменной x
$\exists x(\varphi)$	экзистенциальная квантификация формулы φ по (свободной) переменной x
$\mathbf{X}\varphi$	в следующий момент времени истинна формула φ (neXt time)
$\varphi\ \mathbf{U}\ \psi$	сейчас или когда-нибудь в будущем будет истинна формула ψ; до тех пор, пока ложна формула ψ, истинна формула φ (Until)
$\mathbf{F}\varphi$	сейчас или когда-нибудь в будущем будет истинна формула φ (in the Future)
$\mathbf{G}\varphi$	всегда, начиная с текущего момента времени, истинна формула φ (Globally)
$\varphi\ \mathbf{W}\ \psi$	слабый Until: $(\varphi\ \mathbf{U}\ \psi) \vee \mathbf{G}\varphi$
$\varphi\ \mathbf{R}\ \psi$	освобождение (Release): $\neg(\neg\varphi\ \mathbf{U}\ \neg\psi)$
$\square$	пустая клауза (дизъюнкт)
$Res(C_1, C_2)$	резольвента клауз C_1 и C_2
l^C	литерал контрарный литералу l
$\varphi \approx \psi$	равносильность формул φ и ψ (одновременная выполнимость/невыполнимость)
$[x_1 \coloneqq t_1, \ldots, x_n \coloneqq t_n]$	подстановка термов $t_1, \ldots, t_n$ вместо переменных $x_1, \ldots, x_n$
$\theta_1 \cdot \theta_2$	композиция подстановок θ_1 и θ_2
$\varphi\theta$	применение подстановки θ к формуле φ
НОУ(φ, ψ)	наибольший общий унификатор формул φ и ψ
$closure(\varphi)$	замыкание формулы φ (множество всех подформул и их отрицаний)
$atoms(\varphi)$	множество всех элементарных множеств подформул формулы φ
D_x	домен переменной x (множество возможных значений)
$V(t)$	множество переменных, входящих в терм t

Языки и автоматы

ε	пустое слово
$v \cdot w$	конкатенация конечного слова v и слова w
w^n	n-ая степень конечного слова w: $w^0 = \varepsilon$; если $n > 0$, $w^n = \underbrace{w \cdot w \cdot \ldots \cdot w}_{n \text{ раз}}$
w^*	повторение конечного слова w любое конечное число раз
w^ω	повторение конечного слова w бесконечное число раз
$w^{(n)}$	суффикс $w_n w_{n+1} \ldots$ слова $w = w_0 w_1 \ldots$
X^*	множество всех конечных слов в алфавите X: $\bigcup_{k=0}^{\infty} X^k$
X^ω	множество всех бесконечных слов в алфавите X
$A \otimes B$	синхронная композиция автоматов A и B
$\mathcal{L}_A$	язык, допускаемый автоматом A
B_M	автомат Бюхи, соответствующий структуре Крипке M
B_φ	автомат Бюхи, соответствующий LTL-формуле φ
$\mathcal{L}_M$	язык, соответствующий структуре Крипке M: $\mathcal{L}_{B_M}$ (множество всех возможных траекторий структуры Крипке M)
$\mathcal{L}_\varphi$	язык, соответствующий LTL-формуле φ: $\mathcal{L}_{B_\varphi}$ (множество всех траекторий, на которых формула φ истинна)

Арифметика

$\mathbb{B}$	множество $\{0,1\}$
$\mathbb{N}$	множество натуральных чисел: $\{1, 2, \ldots\}$
$\mathbb{N}_0$	множество натуральных чисел, расширенное 0: $\{0, 1, \ldots\}$
$\mathbb{Z}$	множество целых чисел: $\{0, \pm 1, \pm 2, \ldots\}$
$\mathbb{Q}$	множество рациональных чисел: $\left\{\frac{x}{y} \mid x \in \mathbb{Z}, y \in \mathbb{N}\right\}$
$\mathbb{Q}_+$	множество положительных рациональных чисел: $\left\{\frac{x}{y} \mid x \in \mathbb{N}, y \in \mathbb{N}\right\}$
$[x, y]$	интервал (отрезок) от x до y
$x = y$ ($x \neq y$)	сравнение чисел x и y на равно (не равно)
$x < y$ ($x \geq y$)	сравнение чисел x и y на меньше (больше либо равно)
$x > y$ ($x \leq y$)	сравнение чисел x и y на больше (меньше либо равно)
$x \vdots y$	делимость натурального числа x на натуральное число y
$x + y$	сложение чисел (интервалов) x и y
$x - y$	вычитание числа (интервала) y из числа (интервала) x
$x \cdot y$ или $x * y$	умножение чисел (интервалов) x и y
x/y или $\frac{x}{y}$	деление числа (интервала) x на число (интервал) y
$x \,\%\, y$	остаток от деления целого числа x на целое число y
x^y	возведение числа x в степень y

$\sqrt[y]{x}$	извлечение корня степени y из числа x
$\log x$	(двоичный) логарифм положительного числа x
$\lfloor x \rfloor$	целая часть числа x снизу
$\lceil x \rceil$	целая часть числа x сверху
$\min(x_1, \ldots, x_n)$	минимальное из чисел $x_1, \ldots, x_n$
$\max(x_1, \ldots, x_n)$	максимальное из чисел $x_1, \ldots, x_n$
$\text{НОД}(x, y)$	наибольший общий делитель чисел x и y
$\sim x$	побитовая инверсия двоичного вектора x
$x \,\&\, y$	побитовое И двоичных векторов x и y
$x \mid y$	побитовое ИЛИ двоичных векторов x и y

Двоичные решающие диаграммы

$\boxed{0}$	листовая вершина, соответствующая значению 0 ($false$)
$\boxed{1}$	листовая вершина, соответствующая значению 1 ($true$)
$ZERO$	тривиальная диаграмма с корнем в вершине $\boxed{0}$
ONE	тривиальная диаграмма с корнем в вершине $\boxed{1}$
$root(F)$	корень диаграммы F
$then(F)$	поддиаграмма нетривиальной диаграммы F с корнем в $high\big(root(F)\big)$
$else(F)$	поддиаграмма нетривиальной диаграммы F с корнем в $low\big(root(F)\big)$
$Reduce(F)$	результат применения правил сокращения к диаграмме F
$Compose(x, T, E)$	результат композиции диаграмм T и E по переменной x (корень помечен x, $high(x) = root(T)$ и $low(x) = root(E)$)
$Apply(F \circ G)$	$Reduce\left(Compose\left(x, Apply\big(F_{\lvert x=1} \circ G_{\lvert x=1}\big), Apply\big(F_{\lvert x=0} \circ G_{\lvert x=0}\big)\right)\right)$
$[\![F]\!]$	функция, реализуемая диаграммой F

Программирование

ε	пустая программа
skip	пустой оператор (не изменяет состояния программы)
$x \coloneqq t$	оператор присваивания: x — переменная, t — выражение
$P; Q$	последовательная композиция программ P и Q
if B **then** P **else** Q **end**	условный оператор: B — логическая формула, P и Q — программы
while B **do** P **end**	оператор цикла: B — логическая формула, P — программа
$\{\varphi\}P\{\psi\}$	условие частичной корректности программы P относительно предусловия φ и постусловия ψ

$\langle\varphi\rangle P\langle\psi\rangle$	условие полной корректности программы P относительно предусловия φ и постусловия ψ
$wp(P,\psi)$	слабейшее предусловие программы P относительно постусловия ψ
$sp(P,\varphi)$	слабейшее постусловие программы P относительно предусловия φ
$ssa(P)$	SSA-представление программы P
$\{\varphi\}$	утверждение φ истинно в соответствующей точке программы
$\{\mathbf{inv}\colon \varphi\}$	φ — инвариант соответствующего цикла
$\{\mathbf{bd}\colon u, \langle M,\prec\rangle\}$	u — ограничивающая функция цикла ($\langle M,\prec\rangle$ — фундированное множество)
$M[\![P]\!]$	семантика программы P: $M[\![P]\!]\colon S \to 2^S$, где S — множество состояний
$M^*[\![P]\!]$	отображение, определяемое равенством $M^*[\![P]\!](S) = M[\![P]\!](S) \cup S$
$\to$	отношение переходов на конфигурациях программы
∇	оператор расширения в абстрактной интерпретации
$s(t)$	значение терма t в состоянии s
$[\![p]\!]$	множество состояний программы, в которых истинен предикат p
$\boxed{\mathbf{start}\colon x \coloneqq t}$	начальный оператор блок-схемы: x — переменная, t — выражение
$\boxed{\mathbf{assign}\colon x \coloneqq t}$	оператор присваивания блок-схемы: x — переменная, t — выражение
$\boxed{\mathbf{test}\colon B}$	условный оператор блок-схемы: B — логическая формула
$\boxed{\mathbf{join}}$	оператор соединения блок-схемы
$\boxed{\mathbf{halt}\colon x \coloneqq t}$	завершающий оператор блок-схемы: x — переменная, t — выражение
ϵ	пустая пометка дуги блок-схемы
$P \parallel Q$	параллельная композиция процессов P и Q
$P@l$	пребывание процесса P в точке l
$P@L$	пребывание программы P в одной из точек множества L
χ_X	характеристическая функция множества (отношения) X
$\mathcal{R}(P)$	символическое представление программы P