Александр Камкин

Введение в формальные методы верификации программ

2-е издание, исправленное и дополненное

ДМК Пресс
издательство
Москва, 2025

УДК 004.415.5
ББК 37.972
К18

Камкин А. С.
К18 Введение в формальные методы верификации программ. – 2-е изд., испр. и доп. – М.: ДМК Пресс, 2024. – 304 с.: ил.

ISBN 978-5-93700-340-9

Книга основана на курсах лекций по формальным методам верификации программ, читаемых автором на кафедрах системного программирования МГУ, МФТИ и ВШЭ. Список тем включает: методы формализации семантики языков программирования (операционная и аксиоматическая семантика), методы формальной спецификации требований (программные контракты и темпоральная логика линейного времени), методы доказательства корректности программ (метод индуктивных утверждений и метод фундированных множеств) и методы проверки моделей (теоретико-автоматный подход в явной и символической формах); затрагиваются вопросы абстрактной интерпретации, разрешения ограничений, применения формальных методов в тестировании; даются сведения об инструментах Frama-C/AstraVer, Spin и SMV. Каждая глава сопровождается примерами и упражнениями.

Книга предназначена для студентов и аспирантов программистских специальностей, а также преподавателей и исследователей в области информатики и программной инженерии.

УДК 004.415.5
ББК 37.972

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-5-93700-340-9

© Камкин А. С., 2024
© Оформление, издание, ДМК Пресс, 2024

Содержание

От издательства ... 8

Предисловие ко второму изданию ... 9

Предисловие к первому изданию ... 14

Лекция 1. Введение в формальные методы верификации программ ... 16

1.1. Ошибки в программах ... 17
1.2. Верификация программ ... 19
1.3. Формальная верификация программ ... 20
1.3.1. Проверка эквивалентности ... 22
1.3.2. Проверка модели ... 26
1.3.3. Дедуктивный анализ ... 28
1.4. Вопросы и упражнения ... 30

Лекция 2. Формализация семантики языков программирования ... 32

2.1. Язык программирования while ... 33
2.2. Формальная семантика ... 35
2.2.1. Операционная семантика ... 37
2.2.2. Аксиоматическая семантика ... 39
2.2.3. Доказательное программирование ... 44
2.3. Вопросы и упражнения ... 45

Лекция 3. Дедуктивная верификация последовательных программ ... 47

3.1. Дедуктивные системы и доказательства ... 48
3.1.1. Примеры дедуктивных систем ... 49
3.1.2. Значимость и полнота дедуктивных систем ... 50
3.2. Дедуктивная верификация программ ... 51
3.2.1. Верификация программ на языке while ... 52
3.2.2. Верификация линейных программ ... 56
3.2.3. Верификация ациклических программ ... 57
3.2.4. Верификация циклических программ ... 58
3.3. Вопросы и упражнения ... 60

Лекция 4. Инструменты дедуктивной верификации программ ... 63

4.1. Введение в язык ACSL ... 63
4.1.1. Базовые сведения о языке ... 64
4.1.2. Типы данных ... 66

4.1.3. Вспомогательные определения 66
4.1.4. Контракты функций 68
4.1.5. Утверждения и аннотации циклов 71
4.2. Обзор платформы Frama-C 73
4.2.1. Платформа Frama-C 73
4.2.2. Модули Jessie/AstraVer и платформа Why3 74
4.3. Вопросы и упражнения 75

Лекция 5. Метод индуктивных утверждений 78
5.1. Синтаксис блок-схем 79
5.2. Семантика блок-схем 82
5.3. Метод индуктивных утверждений 83
5.3.1. Метод обратных подстановок 83
5.3.2. Точки сечения, индуктивные утверждения и условия верификации 84
5.3.3. Общая схема метода индуктивных утверждений 86
5.4. Вопросы и упражнения 88

Лекция 6. Метод фундированных множеств 91
6.1. Завершимость программ и полная корректность 91
6.2. Метод фундированных множеств 92
6.2.1. Оценочные функции и условия завершимости 94
6.2.2. Общая схема метода Флойда 94
6.3. Метод счетчиков 96
6.4. Дедуктивная система для доказательства полной корректности 97
6.5. Аннотирование программ для доказательства завершимости 99
6.6. Вопросы и упражнения 100

Лекция 7. Автоматический синтез инвариантов циклов 103
7.1. Автоматизация дедуктивной верификации программ 103
7.2. Эвристические методы синтеза инвариантов циклов 105
7.2.1. Ослабление постусловия 105
7.2.2. Комбинирование пред- и постусловий 107
7.3. Методы синтеза инвариантов на основе абстрактной интерпретации 108
7.3.1. Интервальная арифметика 109
7.3.2. Общие сведения об абстрактной интерпретации 110
7.3.3. Теория выпуклых многогранников 113
7.4. Вопросы и упражнения 116

Лекция 8. Автоматическое доказательство теорем, SAT- и SMT-решатели 118
8.1. Формальные теории и разрешающие процедуры 118
8.1.1. Примеры разрешимых и неразрешимых теорий 120
8.1.2. Задача выполнимости, SAT- и SMT-решатели 122
8.2. Логика высказываний и SAT-решатели 123
8.2.1. Кодировка Цейтина 123

8.2.2. Метод резолюций для логики высказываний 125
8.2.3. Алгоритм DPLL проверки выполнимости 127
8.3. Теории первого порядка и SMT-решатели 130
8.3.1. Метод резолюций для логики первого порядка 131
8.3.2. Теория равенства 133
8.3.3. Комбинирование теорий 135
8.4. Вопросы и упражнения 138

Лекция 9. Параллельные программы и логика LTL 140
9.1. Параллельные программы 140
9.1.1. Параллельные программы над общей памятью 141
9.1.2. Асинхронный параллелизм: семантика чередований 142
9.1.3. Реагирующие системы и справедливость планировщика 144
9.2. Спецификация реагирующих систем 145
9.2.1. Темпоральная логика LTL 146
9.2.2. Структуры Крипке и семантика LTL 149
9.2.3. Аксиоматическое определение LTL 150
9.3. Взаимное исключение процессов 151
9.3.1. Алгоритм Петерсона 151
9.3.2. Справедливость планировщика while-программ 153
9.4. Вопросы и упражнения 153

Лекция 10. Инструмент проверки моделей Spin 156
10.1. Введение в язык Promela 156
10.1.1. Типы данных, переменные и выражения 157
10.1.2. Процессные типы и процессы 161
10.1.3. Средства межпроцессной коммуникации 162
10.1.4. Управляющие операторы 165
10.1.5. Средства спецификации требований 167
10.2. Введение в инструмент Spin 170
10.2.1. Функции инструмента 171
10.2.2. Основные опции 171
10.3. Вопросы и упражнения 172

Лекция 11. Моделирование программ и предикатная абстракция 175
11.1. Представление программ структурами Крипке 176
11.1.1. Обобщенные состояния (абстракция данных) 176
11.1.2. Обобщенные переходы (гранулярность действий) 180
11.2. Модели последовательных и параллельных программ 181
11.2.1. Целочисленное деление 181
11.2.2. Алгоритм Петерсона 183
11.2.3. Построение асинхронной композиции систем переходов 185
11.3. Предикатная абстракция и уточнение абстракции по контрпримерам 186
11.3.1. Декартова предикатная абстракция 186

11.3.2. Уточнение абстракции по контрпримерам (CEGAR) 188
11.4. Вопросы и упражнения 192

Лекция 12. Автоматы Бюхи и регулярные ω-языки 194
12.1. Контрольный автомат 194
12.2. Конечные автоматы и регулярные языки 195
12.2.1. Описание языков конечными автоматами 196
12.2.2. Прямое произведение конечных автоматов 198
12.2.3. Проверка регулярного языка на пустоту 199
12.3. Автоматы Бюхи и регулярные ω-языки 199
12.3.1. Описание ω-языков автоматами Бюхи 200
12.3.2. Прямое произведение автоматов Бюхи 201
12.3.3. Проверка регулярного ω-языка на пустоту 203
12.4. Теоретико-автоматный подход к проверке моделей 205
12.5. Вопросы и упражнения 208

Лекция 13. Синтез автомата Бюхи по формуле LTL 210
13.1. Постановка задачи и предварительные замечания 210
13.2. Состояния автомата 213
13.3. Переходы автомата 215
13.4. Начальные и допускающие состояния автомата 216
13.5. Алгоритм построения автомата 217
13.6. Общая схема проверки моделей для логики LTL 221
13.7. Вопросы и упражнения 222

Лекция 14. Символьная проверка моделей 223
14.1. Символьные методы верификации 224
14.1.1. Символьное представление множеств и отношений 225
14.1.2. Символьное представление структур Крипке 227
14.1.3. Символьные алгоритмы 229
14.1.4. Ограниченная проверка моделей и k-индукция 230
14.2. Символьная проверка моделей для LTL 231
14.2.1. Символьное представление модели программы 232
14.2.2. Символьное представление модели требований 235
14.2.3. Проверка соответствия модели программы и модели требований 235
14.3. Двоичные решающие диаграммы 236
14.3.1. Синтаксис и семантика диаграмм 237
14.3.2. Сокращенные упорядоченные диаграммы 238
14.3.3. Синтез диаграмм и манипуляции с ними 240
14.4. Вопросы и упражнения 243

Лекция 15. Инструменты символьной проверки моделей семейства SMV 246
15.1. Введение в язык SMV 246
15.1.1. Типы данных, переменные и выражения 247

15.1.2. Встроенные функции 251
15.1.3. Модули и их экземпляры 253
15.1.4. Конструкция выбора 257
15.1.5. Средства спецификации требований 258
15.2. Примеры SMV-моделей 258
15.2.1. Алгоритм Петерсона 258
15.2.2. Головоломка «Волк–коза–капуста» 261
15.3. Введение в инструмент NuSMV 263
15.3.1. Функции инструмента 263
15.3.2. Основные опции 263
15.4. Вопросы и упражнения 265

Лекция 16. Использование формальных методов в тестировании ... 269
16.1. Тестирование vs формальная верификация 270
16.2. Тестирование программ без состояния 271
16.2.1. Тестирование на основе программных контрактов 271
16.2.2. Тестирование черного ящика 273
16.2.3. Тестирование белого ящика 276
16.3. Тестирование программ с состоянием 279
16.3.1. Обход графа состояний программы 279
16.3.2. Абстракция состояний программы 280
16.4. Тестирование реагирующих систем и параллельных программ 281
16.5. Заключение 282
16.6. Вопросы и упражнения 282

Используемые сокращения и обозначения 284
Список источников 290
Именной указатель 301

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Предисловие ко второму изданию

Второе издание по большей части совпадает с первым, но есть некоторые отличия. Во-первых, были устранены обнаруженные опечатки и неточности, переработан текст, добавлены иллюстрации и примеры. Во-вторых, была написана новая лекция (глава 15), посвященная SMV-подобным инструментам символьной проверки моделей (теперь теоретический материал лекции 14 подкреплен практикой). Кроме того, в книгу был добавлен именной указатель, а также сноски с краткой информацией о математиках, логиках и программистах, внесших вклад в становление и развитие формальных методов.

Структура книги

Книга состоит из 16 лекций (глав), снабженных вопросами для самопроверки и упражнениями: лекция 1 – вводная; лекции 2–8 посвящены дедуктивному анализу программ; лекции 9–15 – проверке моделей; лекция 16 – использованию формальных методов в тестировании. Более подробная информация о содержании лекций приведена ниже.

В лекции 1 дается представление о корректности программ, т. е. их соответствии требованиям, и верификации – процессе проверки корректности. Делается обзор базовых методов верификации, как формальных (проверка эквивалентности, проверка моделей, дедуктивный анализ), так и неформальных (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.

В лекции 2 на примере простого императивного 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).

В лекции 3 рассматриваются основы дедуктивной верификации последовательных программ: приводятся 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 инварианты циклов не выводимы из структуры программы. Задание инвариантов дает набросок доказательства корректности, проверку которого можно автоматизировать.

В лекции 4 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), а также модулей (plugins) Jessie и AstraVer, автоматизирующих дедуктивную верификацию C-программ на соответствие ACSL-спецификациям. Изложение базируется на примерах (подробное описание языка ACSL и основанных на нем инструментов можно найти в документации). Описанные средства могут использоваться в повседневной программистской практике.

В лекции 5 рассматривается один из первых подходов к формальной верификации программ – метод Флойда. Подход состоит из двух частей: первая (метод индуктивных утверждений) предназначена для доказательства частичной корректности, т. е. корректности без учета завершимости; вторая (метод фундированных множеств) используется для доказательства завершимости и полной корректности (этой части посвящена лекция 6). Метод Флойда имеет дело с блок-схемами и позволяет анализировать программы с общим видом потока управления, в том числе не отвечающие принципам структурного программирования. Подход состоит в рассечении циклов, сопоставлении точкам сечения утверждений, построении условий верификации, связывающих утверждения соседних точек (индуктивных переходов), и их доказательстве.

В лекции 6 рассматривается метод фундированных множеств, применяемый в паре с методом индуктивных утверждений для доказательства завершимости и полной корректности программ. Идея подхода заключается в следующем: каждой точке сечения сопоставляется оценочная функция; значение функции уменьшается при каждом прохождении через эту точку (при условии что истинно индуктивное утверждение), но в то же время не может уменьшаться бесконечно долго в силу характера изменений и ограничений на область значений. Если удается найти такие функции и доказать условия верификации, программа завершается и, более того, является полностью корректной.

В лекции 7 рассматриваются вопросы автоматизации дедуктивной верификации программ, в частности вопросы автоматического синтеза инвариантов циклов. Задача синтеза оценочных функций может быть сведена к построению инвариантов программ, расширенных счетчиками. Описываются два класса методов: эвристические методы и методы, основанные на абстрактной интерпретации, т. е. символьном исполнении в упрощенных предметных областях – абстрактных моделях. Определяются основные понятия абстрактной интерпретации; приводятся примеры для таких областей, как интервальная арифметика и теория выпуклых многогранников.

В лекции 8 рассматриваются основные подходы к автоматическому доказательству теорем, включая метод резолюций для логики высказываний и логики первого порядка. Вводятся понятия разрешающей процедуры и разрешимой теории, приводятся примеры разрешимых и неразрешимых теорий. Большое внимание уделяется базовым принципам работы SAT- и SMT-решателей – средств, составляющих ядро инструментов формальной верификации программ: 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.

В лекции 9 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 (на каждом шаге исполняется оператор одного из процессов, выбранного недетерминированным образом). Особое внимание уделяется так называемым реагирующим системам, которые работают в «бесконечном цикле» и отвечают 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 параллельных программ используется алгоритм Петерсона взаимного исключения двух процессов.

В лекции 10 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 их соответствие LTL-спецификациям, т. е. проверять, является ли заданная модель (программа на языке Promela) «моделью» (в логическом понимании) заданной формулы LTL (истинна ли формула на всех вычислениях модели). Описанные средства могут быть использованы на практике при проектировании протоколов, параллельных программ и реагирующих систем.

В лекции 11 рассматриваются вопросы моделирования программ структурами Крипке, в том числе связанные с выбором уровня абстракции данных и гранулярности действий. Обсуждаются возможные аномалии: появление в модели поведения, не присущего 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 программы.

В лекции 12 рассматриваются ω-языки, т. е. языки с бесконечными словами, и один из способов их описания – автоматы Бюхи. Структурно ав-

томаты Бюхи идентичны конечным автоматам, но отличаются от них условием распознавания слов. Показывается, как для языков, описываемых конечными автоматами и автоматами 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. Показывается, что множество траекторий структуры Крипке тривиальным образом описывается автоматом Бюхи. Вопрос о возможности представления автоматами Бюхи формул LTL рассматривается в следующей лекции.

В лекции 13 рассматривается ключевой этап теоретико-автоматного подхода к проверке моделей для логики LTL – синтез автомата Бюхи, распознающего все траектории, на которых истинна заданная формула LTL (и только их). Результатом синтеза является недетерминированный обобщенный автомат Бюхи, число состояний которого экспоненциально зависит от длины формулы. Описанная процедура может быть использована и для проверки выполнимости формулы: если язык, допускаемый построенным автоматом, не пуст, то формула выполнима (существует траектория, на которой она истинна).

В лекции 14 рассматривается символьная проверка моделей для логики LTL. Отличие от классического подхода, рассмотренного ранее, состоит в том, что модель (множество состояний и отношение переходов), а также автомат Бюхи, построенный по формуле LTL, представляются неявно, в символьной форме. Общая идея метода остается прежней, но алгоритмы поиска в графах состояний заменяются на манипуляции с их символьными представлениями, что частично решает проблему комбинаторного взрыва. Ключевую роль здесь играют способы символьного представления множеств и отношений. Рассматривается один из таких способов, позволяющий создавать эффективные символьные алгоритмы, – двоичные решающие диаграммы (BDD, Binary Decision Diagrams).

В лекции 15 (добавленной в этом издании) рассматриваются основные возможности инструментов семейства SMV (Symbolic Model Verifier) и лежащего в их основе языка моделирования: представление параллельных систем в виде иерархически организованных модулей, спецификация требований на языке темпоральной логики и собственно символьная проверка моделей, базирующаяся на двоичных решающих диаграммах (BDD) и техниках проверки выполнимости (SAT). Описанные средства могут быть использованы на практике для разработки и анализа компьютерных протоколов, моделей программ и цифровой аппаратуры.

В лекции 16 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 (model-based testing). 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, символьное исполнение и конколическое (конкретно-символьное) тестирование, генерация тестов с помощью обхода графа состояний.

Благодарности

Автор хотел бы поблагодарить **А. М. Коцыняка** – коллегу по ИСП РАН, на протяжении многих лет проводящего практические занятия по курсу (без которых невозможно прочувствовать трудность верификации и красоту формальных методов). Все прежние благодарности, конечно, остаются в силе.

Рекомендуемая литература

Для более глубокого погружения в тему формальной верификации программ рекомендуем книги [Apt09], [Bai08], [Lei23], [Вел11], [Кар10] и [Мир23].

Замечания

Автор будет рад любой информации об ошибках и неточностях, встретившихся на страницах книги; приветствуются любые предложения по ее улучшению. Контактная информация не изменилась (см. предисловие к первому изданию).

Предисловие к первому изданию

Корректность программы – соответствие ее *функциональности* (того, *что* программа делает) *требованиям* (тому, что она *должна* делать) – может достигаться либо *безошибочным проектированием* (*correctness by design*), либо путем постепенного выявления и устранения несоответствий, ошибок. Первый подход кажется предпочтительным. В самом деле, зачем тратить усилия на поиск и исправление ошибок, если можно сразу все сделать правильно? Однако не все так просто. Во-первых, не сразу понятно, что значит «правильно»: требования к программе уточняются по мере ее разработки: то, что кажется правильным сегодня, оказывается неправильным завтра. Во-вторых, программа эволюционирует: требования к каждой конкретной версии могут быть заданными, но меняются от версии к версии; «синтез» корректной программы с нуля по обновленным спецификациям требует огромных затрат; если же идти путем доработки имеющейся версии (что обычно и делают), может нарушиться корректность.

Верификацией программ, или, шире, компьютерных систем, называется деятельность, направленная на выяснение их правильности или, наоборот, ошибочности. Деятельность эта может принимать разные формы: *инспекция кода* (просмотр и рецензирование «исходников»); *статический анализ* (поиск типовых ошибок: чтений неинициализированных переменных, разыменований нулевых указателей и т. п.); *тестирование* (запуск программы на примерах и проверка корректности результатов); *имитационное моделирование* (создание исполнимой модели системы и ее исследование в специальном окружении); *формальная верификация* (построение логической модели системы и ее анализ средствами математической логики). Подходов много, однако задача верификации настолько трудна, что ни один из них не годится на роль «серебряной пули», способной гарантировать корректность действительно сложных проектов; лучшие результаты, как показывает практика, достигаются при совместном использовании разных методов.

Предлагаемое вашему вниманию пособие посвящено одному классу методов – *формальным методам*. Их суть состоит в создании математических моделей программ и требований и в логическом анализе соответствия между построенными моделями. В некотором смысле формальные методы – это фундамент, на котором стоит здание программной инженерии. (Аналогичным фундаментом в свое время выступил математический анализ по отношению к физике.) Следует отметить, что это не «голая теория», как часто думают: формальные методы давно вышли за пределы академического сообщества и стали частью индустрии; сегодня это неотъемлемая часть процесса

разработки систем ответственного назначения (микропроцессоров, операционных систем, бортовых комплексов). Пособие преследует две основные цели: способствовать более глубокому пониманию природы программирования и сформировать практические навыки верификации компьютерных систем.

Книга основана на курсах лекций, читаемых автором на факультете ВМК МГУ им. М. В. Ломоносова, ФУПМ МФТИ и ФКН ВШЭ. В ней изложены основы широко известных подходов к верификации: *дедуктивного анализа* и *проверки моделей* (*model checking*). Рассматриваются такие темы, как методы формализации семантики языков программирования (операционная и аксиоматическая семантика), методы формальной спецификации требований (программные контракты и темпоральная логика линейного времени), методы доказательства корректности программ (метод индуктивных утверждений и метод фундированных множеств) и методы проверки моделей (теоретико-автоматный подход в явной и символьной формах). Кроме того, в пособии затрагиваются вопросы абстрактной интерпретации, автоматизированного доказательства теорем и разрешения ограничений, применения формальных методов в тестировании; даются сведения об инструментах Frama-C и Spin.

Пособие написано для студентов и аспирантов программистских специальностей, а также преподавателей и исследователей в области информатики и вычислительной техники. От читателя требуется знание основ дискретной математики и математической логики.

Благодарности

Автор благодарен академику **В. П. Иванникову** (1940–2016) – основателю, первому директору и научному руководителю ИСП РАН, заведующему кафедрами системного программирования МГУ, МФТИ и ВШЭ, – инициировавшему создание учебного курса по верификации и предложившему автору его читать, и профессору **А. К. Петренко** – заведующему отделом Технологий программирования ИСП РАН, – всячески поддерживающему автора в написании этого пособия.

Отдельных слов благодарности заслуживают студенты – слушатели курса.

Замечания

Как и программы, книги содержат ошибки, и эта – не исключение. Автор будет признателен за любую информацию об ошибках и неточностях, встретившихся на страницах пособия, а также за замечания и предложения по его улучшению.

Контактная информация
Камкин Александр Сергеевич, к. ф.-м. н., в. н. с. ИСП РАН
109004, г. Москва, ул. Александра Солженицына, д. 25 (ИСП РАН)
E-mail: kamkin@ispras.ru

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, как формальных (проверка эквивалентности, проверка моделей, дедуктивный анализ), так и неформальных (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 (даже тщательно отлаженной) программе есть хотя бы одна ошибка[1]. Эта шутка появилась не на пустом месте: при высокой сложности компьютерных систем (а это миллиарды транзисторов и миллионы строк кода) избежать ошибок – несоответствий системы требованиям – практически невозможно. Речь идет не только о прикладных решениях, создаваемых в спешке не вполне компетентными специалистами, но и об ответственных (safety-critical) компонентах, включая аппаратуру (микропроцессоры, устройства ввода-вывода, хранения и передачи данных) и системное ПО (операционные системы, компиляторы, драйверы устройств). Системы, при проектировании или реализации которых допущены ошибки, могут в той или иной ситуации повести себя непредсказуемо, приводя к фатальным последствиям. Примеров этому много (см., например, [Адж98] и [Кар10]). Вот некоторые из них, на наш взгляд, достаточно показательные.

- В 1982 году 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 года по январь 1987 года) как минимум два пациента умерли, а несколько человек остались инвалидами[2].
- 2 сентября 1988 года была потеряна связь с советской межпланетной станцией «Фобос-1», запущенной 7 июля того же года для исследования Марса и его спутника Фобоса. За несколько дней до этого, 29 августа, с Земли была передана некорректная команда, ошибочно воспринятая как отключение системы ориентации и стабилизации. Станция перестала направлять солнечные батареи на Солнце, из-за чего ее аккумуляторы разрядились. Через полгода была потеряна связь с «Фобос-2», дублером «Фобос-1». Основная задача обеих миссий – доставка на по-

---

[1] Это высказывание известно как первая аксиома М. Р. Шуры-Буры. 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.

Михаил Романович Шура-Бура (1918–2008) – советский и российский ученый, внесший существенный вклад в становление и развитие программирования в СССР; лауреат Сталинской премии (1955) и Государственной премии СССР (1978).

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

верхность Фобоса долгоживущей автономной станции – осталась невыполненной[1].

- 13 июня 1994 года Томас Найсли (Thomas Nicely), профессор математики из Линчбургского колледжа, обнаружил ошибку в реализации инструкции деления чисел с плавающей точкой в микропроцессоре Pentium фирмы Intel (как выяснилось, в таблице начальных приближений присутствовало несколько неправильных значений). И хотя рядовых пользователей ПК эта ошибка, казалось бы, не должна была беспокоить, компании для сохранения имиджа пришлось организовать бесплатную замену выпущенных микросхем. Затраты на это составили 475 миллионов долларов – более половины прибыли Intel за последний квартал 1994 года.
- 4 июня 1996 года на 39-й секунде после старта взорвалась ракета-носитель «Ариан 5», разработанная Европейским космическим агентством (ЕКА). Бортовая система частично использовала ПО предыдущей версии ракеты, «Ариан 4», в том числе модуль управления инерциальной навигационной системой[2]. При преобразовании 64-битного числа с плавающей точкой (угла наклона ракеты) в 16-битное целое возникло переполнение, не обработанное модулем. При разработке предполагалось, что переполнение невозможно из-за физических ограничений ракеты, но прогресс не стоит на месте – в «Ариан 5» использовались новые двигатели.
- 23 марта 2003 года во время войны в Ираке американский зенитно-ракетный комплекс «Patriot» ошибочно идентифицировал британский истребитель-бомбардировщик «Tornado», летевший близ кувейтской границы, как приближающуюся вражескую ракету и автоматически произвел залп. Погибли два пилота. Спустя полторы недели, 2 апреля, «дружественным огнем» (friendly fire) был уничтожен американский палубный самолет F/A-18 «Hornet». Еще один пилот погиб.

Список можно продолжить... Следует понимать, что ошибки в системах не являются чем-то исключительным: как говорили римляне, errare humanum est – человеку свойственно ошибаться. Согласно статистике, среднее число ошибок на тысячу строк неотлаженного кода колеблется в пределах 10–50[3] [Mcc04]. К сожалению, в некоторых областях наблюдается тенденция к деградации качества проектирования: «С точки зрения способности все за-

---

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

[2] Кстати говоря, ПО в ЕКА разрабатывается на языке Ada, созданном по заданию Министерства обороны США как язык программирования встроенных систем с повышенными требованиями к надежности и безопасности.

[3] Известно, что ошибки распределены в программах неравномерно и имеют тенденцию образовывать скопления [Gla02]. Согласно [Boe01], примерно 80 % ошибок находятся в 20 % модулей (это одно из проявлений принципа Парето).

путать и испортить компьютер с каждым днем становится все больше похож на человека» [Llo07]. По-видимому, это следствие возрастающей сложности систем и неразумной оптимизации расходов на их создание и эксплуатацию. Вместе с тем наша жизнь все сильнее зависит от информационных технологий (онлайн-банкинг, интернет вещей, облачные хранилища данных, медицина, транспорт), поэтому все бóльшее значение приобретает *верификация*.

1.2. Верификация программ

Верификацией называется проверка соответствия программы (или некоторого промежуточного результата проектирования: модели, прототипа и т. п.) предъявляемым к ней требованиям (проектной документации). Если программа соответствует требованиям, она называется *корректной* (*правильной*); в противном случае программа называется *некорректной* (*ошибочной*), а само несоответствие требованиям – *ошибкой*[1]. Таким образом, верификация – это анализ программы на предмет отсутствия или наличия в ней ошибок. Говоря утрированно, результатом верификации является *вердикт*:

- *отрицательный* – в программе найдена хотя бы одна ошибка (продемонстрировано, что в определенной ситуации поведение программы отличается от ожидаемого);
- *положительный* – доказано (причем достаточно убедительно), что ошибок нет;
- *неопределенный* (*unknown*, *inconclusive*) – ошибки не найдены, но их отсутствие не доказано (это весьма распространенная ситуация).

Так обстоит дело в теории, на практике же необходимо учитывать множество нюансов. Во-первых, требования часто представляются *неформально*, на естественном языке, поэтому не всегда понятно, соответствует им программа или нет. Во-вторых, даже если требования *формализованы*, т. е. определены на строгом, машиночитаемом языке, проверить отсутствие ошибок в программе трудно с математической точки зрения (эта работа не вполне поддается автоматизации). В большинстве проектов задача доказательства корректности ПО не ставится[2]. Типичной целью является обеспечение качества, где под *качеством* понимают интегральную характеристику[3], учитывающую в том числе показатели надежности.

---

[1] Слово «ошибка» может пониматься по-разному: и как отклонение поведения программы от ожидаемого (сбой – failure), и как причина такого отклонения (неисправность, ошибка в коде – fault, bug). Обычно значение понятно из контекста.

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

[3] Характеристики качества ПО включают переносимость, надежность, эффективность, удобство в использовании, тестируемость, понятность, модифицируемость [Gla02].

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

1) *формальные методы*, основанные на формализации программ и требований – построении моделей – и математически строгом анализе этих моделей;
2) *методы тестирования*, осуществляющие проверку реального поведения программ на некотором наборе сценариев (*тестов*);
3) *экспертизу*, выполняемую людьми на основе их знаний и опыта непосредственно над результатами проектирования (например, *инспекция кода*).

«Серебряной пули» [Bro95] нет – методы каждой группы имеют свои достоинства и недостатки, свою область применимости. Полноценная верификация систем ответственного назначения невозможна без совместного использования нескольких дополняющих друг друга подходов. Методы верификации, совмещающие разные техники, называются *гибридными*, или *синтетическими* [Кул08]. Примеры таких методов, а именно методы тестирования, использующие элементы формальной верификации (так называемые *методы тестирования на основе моделей*), рассматриваются в лекции 16.

Книга, которую вы держите в руках, посвящена формальным методам верификации программ. Безусловно, эти методы не покрывают всех потребностей индустрии разработки ПО, но как математика является языком науки, так формальные методы являются языком программной инженерии. Приведем слова Джона Маккарти[1]: «Разумно ожидать, что связи между вычислительной техникой и математической логикой окажутся столь же плодотворными в следующем столетии [XXI веке. – *А. К.*], какими были связи между математическим анализом и физикой в столетии предыдущем [XIX веке. – *А. К.*]» [Усп87].

1.3. Формальная верификация программ

Формальная верификация основывается на математическом (формально-логическом) моделировании программ и требований к ним. Идея здесь точно такая, как при использовании моделей в других областях знаний:

- создается формальная модель – идеализированное описание исследуемого объекта или явления;
- модель исследуется с применением математических методов;

[1] Джон Маккарти (John McCarthy; 1927–2011) – американский ученый-информатик; автор термина «искусственный интеллект» (1955), создатель языка Lisp; лауреат премии Тьюринга (1971).

- результаты исследования переносятся на реальный объект или явление.

Правомерность такого подхода зависит от используемых моделей – нужно понимать условия их применимости (адекватности).

Общая схема формальной верификации показана на рис. 1.1:

- создаются формальная модель программы и формальная модель требований;
- формально проверяется соответствие модели программы модели требований;
- на основании результатов проверки делается вывод о соответствии или, наоборот, несоответствии реальной программы реальным требованиям (т. е. об отсутствии или наличии ошибок в программе).

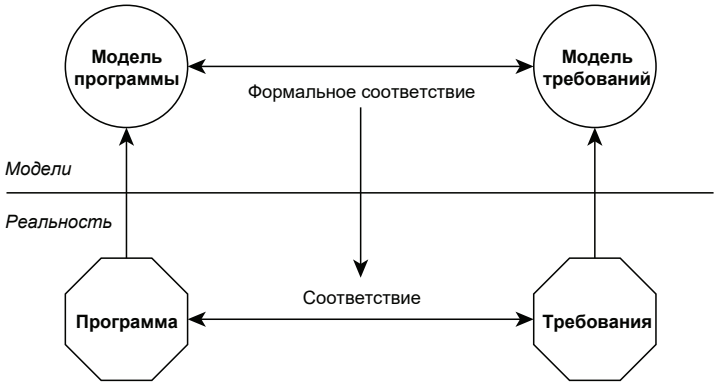


Рис. 1.1 ❖ Общая схема формальной верификации

Для представления моделей программ и моделей требований используются соответственно *языки моделирования программ* и *языки спецификации требований* (примеры таких языков будут даны в последующих лекциях).

Методы формальной верификации, как явствует из схемы, различаются в следующих аспектах:

1) типы моделей программ (схемы программ, конечные автоматы, сети Петри);
2) типы моделей требований (программные контракты, темпоральные утверждения, контрольные автоматы);
3) отношения соответствия (эквивалентность, (би)симуляция, включение трасс, частичная/полная корректность);
4) техники проверки соответствия (исследование пространства состояний, автоматизированное доказательство теорем).

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

1.3.1. Проверка эквивалентности

В *проверке эквивалентности* (*equivalence checking*) модель программы и модель требований однотипны (это могут быть логические схемы, конечные автоматы или схемы программ), а в качестве отношения соответствия используется одно из отношений эквивалентности, определенное для моделей рассматриваемого типа (обычно это функциональная эквивалентность). Модель требований часто называют *эталонной моделью* (*reference model*).

Напомним: отношение эквивалентности – это бинарное отношение (обычно обозначаемое $\sim$), схожее с равенством, т. е. удовлетворяющее свойствам *рефлексивности* ($x \sim x$), *симметричности* (если $x \sim y$, то $y \sim x$) и *транзитивности* (если $x \sim y$ и $y \sim z$, то $x \sim z$).

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

(*Детерминированным*) *конечным автоматом Мили*[1], или просто *конечным автоматом* (для краткости – *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;
- δ: $S \times X \to S$ – функция переходов;
- λ: $S \times X \to Y$ – функция выходов.

Все упомянутые множества полагаются конечными и непустыми.

Семантика автомата определяется отображением цепочек стимулов в цепочки реакций. Два автомата над общими входным и выходным алфавитами называются (*функционально*) *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 входящими в них стрелками (ведущими из «ниоткуда»); первая пометка на дуге – стимул, вторая – реакция:

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

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

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

---

[1] Джордж Мили (George H. Mealy; 1927–2010) – американский математик и информатик; в его честь названы конечные автоматы определенного вида (автоматы Мили), обычно противопоставляемые автоматам Мура (разница в том, что в автоматах Мура выход не зависит от входа, а определяется только состоянием).

где

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

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

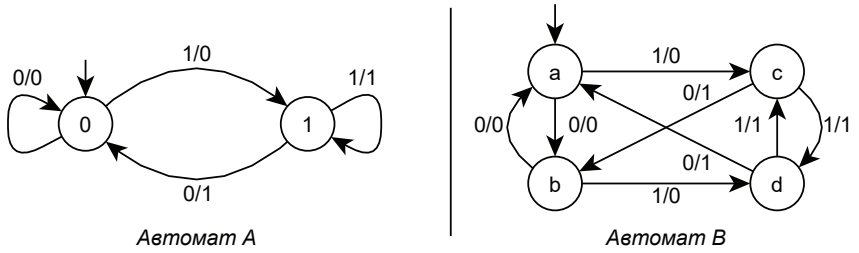


Рис. 1.2 ❖ Пример эквивалентных конечных автоматов

□

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*[1] *о различении состояний автоматов*, определяющая ограничение на длину теста в зависимости от числа состояний сравниваемых автоматов [Але20]. Формулировка: 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[2]. Заметим, что тестирование применимо, даже когда автоматы являются «черными ящиками», т. е. когда их структура неизвестна.

Другим подходом является статический анализ отношений переходов. На этот счет есть другая теорема Мура [Кар03]: два конечных автомата A и B

---

[1] Эдвард Форест Мур (Edward Forrest Moore; 1925–2003) – американский математик и информатик; известен многочисленными результатами в области конечных и клеточных автоматов.

[2] Другими словами, существует последовательность стимулов длины, не превосходящей $n + m - 1$, подача которой на данные автоматы в данных состояниях приводит к разным последовательностям реакций.

эквивалентны тогда и только тогда, когда в любом достижимом состоянии (s_A, s_B) их прямого произведения 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. В общих словах это пара автоматов, синхронно обрабатывающих одни и те же стимулы.

Есть и третий подход – эквивалентное преобразование одного автомата к другому или обоих автоматов к третьему, минимальному (в последнем случае применяют техники объединения неразличимых состояний [Aho74]). Методы этого типа широко распространены на практике, однако следует иметь в виду, что конечной системы (локальных) преобразований (rewriting system), позволяющей любую пару эквивалентных автоматов приводить друг к другу, не существует [Ябл86].

Перейдем теперь к более сложным объектам – программам. Две (детерминированные) последовательные программы с одинаковыми множествами входных и выходных переменных называются (*функционально*) *эквивалентными*, если их области определения совпадают (входные данные, допустимые для одной программы, допустимы и для другой) и на любых допустимых входах их выходы совпадают.

Пример 1.2

Ниже в табл. 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`
`  else`
`  y := y - x`
`  end`
`end;`
`c := x`
	`x := a;`
	`y := b;`
	`while y ≠ 0 do`
	`  z := x;`
	`  x := y;`
	`  y := z % y`
	`end;`
	`c := x`

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

Можно показать, следуя нашей догадке, что обе программы вычисляют НОД(a, b) и записывают результат в переменную c, т. е. для каждой из них

справедливо следующее соотношение между входными и выходными переменными:

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

Как это делается, рассматривается в лекциях 3–6, посвященных дедуктивной верификации. Говоря об автоматической проверке, все же неясно, как заставить компьютер «понять», что делает программа, какой у нее «смысл».

Можно пойти другим путем – не вникая в смысл сравниваемых программ, попытаться трансформировать одну к другой, применяя эквивалентные преобразования. Трансформируем, например, Q к P.

1. Заменим $y := z \,\%\, y$ (третье присваивание в теле цикла) на $y := z \,\%\, x$. Очевидно, это можно сделать, поскольку $x = y$ (см. второе присваивание).
2. Присваивание $y := z \,\%\, x$ заменим на цикл

   ```
   y := z; while y ≥ 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 эквивалентно, на $x \neq y$ (вопрос читателю: почему это можно сделать?).
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:

   ```
   ...
   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 **if–then** (внешнего цикла достаточно для организации вычислений).

6. Внешнее условие $x \neq y$ позволяет объединить два оператора **if–then** с условиями $x > y$ и $y < x$ в один оператор **if–then–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 полностью вникать в суть программ (в преобразованиях участвуют лишь ограниченные фрагменты кода), приведение программ друг к другу в общем случае не 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 рассматриваемых классов программ.

Подробно тема проверки эквивалентности программ в книге не рассматривается; читателям, заинтересовавшимся этой проблематикой, рекомендуем монографию [Кот91] и диссертацию [Зах12].

1.3.2. Проверка модели

В *проверке моделей* (*model checking*) требования к программам представляются логическими формулами того или иного вида, а сами программы – структурами, интерпретирующими такие формулы; отношение соответствия – истинность формулы на структуре (программа удовлетворяет требованиям, если соответствующая структура являются моделью[1] соответствующей формулы) [Cla99].

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*) [Кар10]. Соответственно, для представления программ используются структуры Крипке (размеченные системы переходов).

[1] Слово «модель» здесь используется в формально-логическом смысле: моделью формулы называется всякая интерпретация, в которой эта формула истинна.

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.

В контексте проверки моделей рассматриваются так называемые *реагирующие системы* (*reactive systems*) – системы, работающие в «бесконечном цикле» и постоянно взаимодействующие со своим окружением. Поведение реагирующей системы моделируется *вычислением в структуре Крипке* – бесконечной последовательностью состояний, первое из которых принадлежит множеству S_0, а каждая пара последовательных состояний связана отношением R.

Для описания требований может применяться логика LTL. Помимо классических связок – отрицания (¬), конъюнкции (∧), дизъюнкции (∨), импликации (→), – в ней есть темпоральные операторы **G** (*Globally*), **F** (*in the Future*), **X** (*neXt time*) и **U** (*Until*), интерпретируемые следующим образом:

- $\mathbf{G}\varphi$ – формула φ истинна *всегда*;
- $\mathbf{F}\varphi$ – формула φ истинна *сейчас* или станет истинной *когда-нибудь в будущем*;
- $\mathbf{X}\varphi$ – формула φ станет истинной *в следующий момент времени*;
- $\varphi\ \mathbf{U}\ \psi$ – формула φ истинна *до тех пор, пока* не станет истинной формула ψ (предполагается, что когда-нибудь это произойдет).

Полноценное определение синтаксиса и семантики 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 изображено слева; рядом с каждым состоянием указано множество истинных в нем высказываний). Состояния структуры размечены следующим образом:

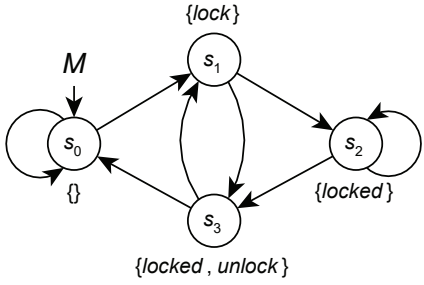


Рис. 1.3 ❖ Структура Крипке, моделирующая дверной замок

- $L(s_0) = \emptyset$ – 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.

Ниже приведены примеры требований к системе, выраженные в LTL:

- $\mathbf{G}\{\neg(lock \wedge unlock)\}$ – нельзя одновременно и открывать, и закрывать дверь;
- $\mathbf{G}\{\neg(locked \wedge lock)\}$ – нельзя закрывать закрытую дверь;
- $\mathbf{G}\{lock \rightarrow \mathbf{X}(locked)\}$ – если дверь закрывают, она станет закрытой в *следующий момент времени*;
- $\mathbf{G}\{locked \rightarrow \neg(\neg unlock\ \mathbf{U}\ \neg locked)\}$ – если дверь закрыта, она *будет* таковой, *пока* ее не попытаются открыть (не произведут действия *unlock*) – результат не гарантирован, но без действия *unlock* дверь точно не откроется.

Темпоральный оператор **G**, предваряющий каждое требование, говорит о том, что требование применимо *всегда* – в каждый момент времени.

□

Теме проверке моделей, а именно теоретико-автоматному подходу к проверке моделей для логики LTL, посвящены лекции 11–13. Пока лишь отметим, что основной техникой установления корректности или некорректности модели является поиск в пространстве состояний (обход графа состояний). В лекции 14 рассматриваются более современные подходы – символьные методы.

1.3.3. Дедуктивный анализ

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

Будем считать, что переменные программы делятся на три группы:

- *входные переменные* – содержат исходные данные и не меняются во время исполнения программы;
- *внутренние переменные* – используются для хранения промежуточных результатов (обычно не видны снаружи);
- *выходные переменные* – предназначены для выдачи полученного решения (допускается использование внутренних переменных в качестве выходных).

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

(предусловие ∧ завершается) ⇒ постусловие.

Говорят, что программа P *полностью корректна* относительно предусловия φ и постусловия ψ (обозначение: $\langle\varphi\rangle P\langle\psi\rangle$ или $\{\varphi\}P\{\varphi\}$, если из контекста ясно, что речь идет о полной корректности), если для всех значений входных переменных, для которых истинно φ, P завершается и для значений выходных переменных истинно ψ:

предусловие ⇒ (завершается ∧ постусловие).

Пример 1.4

Ниже приведена программа *DIV*, у которой две входные целочисленные переменные (a и b) и две выходные целочисленные переменные (q и r). Для программы задано предусловие (символ ≡ означает «есть по определению»)

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

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* полностью корректна относительно предусловия φ и постусловия ψ:

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

Символ ⊨, как это принято в логике, говорит, что стоящая справа от него формула истинна.

□

Теме дедуктивной верификации программ будут посвящены лекции 3–6. Краеугольным камнем этой группы методов является поиск так называемых *инвариантов циклов*. Но обо всем по порядку...

1.4. Вопросы и упражнения

1. Определите следующие понятия: «ошибка в программе», «корректность программы», «верификация программы».
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 (функционирование) 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 доступными вам средствами, 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 останавливает мотор и 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. В первую очередь идентифицируйте «опасные ситуации» в работе стиральной машины, которые никогда не должны возникать. Выразите требования в логике LTL.
11. Выразите в виде формул LTL следующие требования:
    a) если нажать «Пуск» и не нажимать «Стоп» (во время стирки белья), то рано или поздно белье постирается;
    b) если нажать «Пуск», то рано или поздно белье постирается (если не нажимать «Стоп»), при этом во время стирки дверца бака будет заблокирована.
12. Определите программный контракт (пред- и постусловие) для программы, вычисляющей квадратный корень числа с заданной точностью (x – входное число, y – результат, ε – точность вычисления).
13. Определите программный контракт (пред- и постусловие) для программы сортировки числового массива (x – входной массив, N – его размер, x' – выходной массив, N' – его размер).
14. 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

Прежде чем доказывать (или опровергать), что программа – код на языке программирования – делает то, что от нее требуется, нужно понять, как *интерпретируются* конструкции языка, на котором она написана, т. е. как коду сопоставляется *смысл*. Другими словами, должна быть определена *семантика* языка программирования, причем если речь идет о формальной верификации, семантика должна быть *формализована*.

Подходы к формализации семантики языков программирования мы будем рассматривать на примере языка while, простого императивного языка программирования [Apt09]. Сделаем два замечания. Первое: в обычных языках программирования *сигнатура*, т. е. множество *функциональных* и *предикатных символов* (вместе с их арностью)[1], используемых в выражениях ($0^{(0)}$, $1^{(0)}$, $+^{(2)}$, $-^{(2)}$, $=^{(2)}$, $<^{(2)}$, $>^{(2)}$), 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 табл. 2.1.

Таблица 2.1. Грамматика языка программирования while

P ::=	
skip	пустой оператор
| x := t	оператор присваивания
| P; P	последовательная композиция
| **if** B **then** P **else** P **end**	условный оператор
| **while** B **do** P **end**	оператор цикла

Здесь P – цель грамматики (while-программа), x – переменная, t – терм (функциональное выражение), B – формула (логическое выражение, условие).

Термы и формулы определяются привычным образом:

- переменная есть терм;
- константа (функциональный символ арности 0) есть терм;
- если f – функциональный символ арности $k > 0$, а $t_1, \ldots, t_k$ – термы, то $f(t_1, \ldots, t_k)$ есть терм;
- логическая константа (предикатный символ арности 0) – *true* или *false* – есть формула;

[1] Символы констант – это функциональные символы арности 0.

- если p – предикатный символ арности $k > 0$, а $t_1, \ldots, t_k$ – термы, то $p(t_1, \ldots, t_k)$ есть формула[1];
- 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[2].

Никакие другие объекты термами и формулами не являются.

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

Как в языке C, однострочные комментарии будем начинать с //, а многострочные помещать внутрь /* и */.

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: частное от деления сохраняется в переменную q, а остаток – в переменную r (это программа *DIV* из лекции 1).

```
q := 0;  /* инициализация q и r */
r := a;
while r ≥ b do  // цикл
  q := q + 1;
  r := r - b
end
```

□

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

1) *задача спецификации* – описать желаемую семантику программы (определить функциональные требования к программе)[3];
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);

---

[1] Такие формулы, как и логические константы, называются атомарными формулами, или атомами.

[2] Определение формулы можно расширить и на другие логические связки.

[3] В этой и других задачах термины «семантика программы» и «требования к программе» используются как синонимы, хотя, конечно, это не одно и то же: семантика – это то, что у программы *есть*, а требования – то, что *должно быть*. 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.

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.

В формальных методах предполагается, что описание (спецификация) и проверка (верификация) осуществляются формально, с применением методов математической логики.

2.2. Формальная семантика

Интуитивно понятно, что каждая программа (если только она не зацикливается) осуществляет отображение (возможно, недетерминированное) одного состояния, т. е. множества значений переменных, в другое. Таким образом, семантика программы может быть определена заданием отношения между множествами значений входных и выходных переменных. Такой подход называется *декларативным*; он активно используется при описании требований и выводе свойств программ.

Альтернативой является сопоставление программе последовательности действий некоторого исполнителя (автомата, машины). Это так называемый *интерпретационный* подход; его области применения включают разработку компиляторов и интерпретаторов языков программирования[1].

Для формализации семантики языка программирования сначала нужно определить семантику используемых в нем функциональных и предикатных символов. Это может быть сделано двумя основными способами:

1) заданием *дедуктивной системы* (см. лекции 3 и 8) – множества *аксиом* и *правил вывода*, определяющих взаимосвязи между символами сигнатуры (такой способ используется в декларативном подходе);
2) заданием *процедуры*, позволяющей вычислять значения выражений по значениям переменных (такой способ используется в интерпретационном подходе).

---

[1] 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 заданием схемы 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).

Сигнатуры вместе с дедуктивными системами или вычислительными процедурами, используемыми для их смыслового наполнения, будем называть *предметными областями*, или *доменами*[1].

Пример 2.2

Проиллюстрируем понятие дедуктивной системы на примере арифметики Пеано[2] [Pea89]. Сигнатурой этой системы является $\langle\{+^{(2)}, \cdot^{(2)}, 0^{(0)}, 1^{(0)}\}, \{=^{(2)}\}\rangle$; аксиомы и правила вывода приведены ниже:

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. $\dfrac{\varphi(0), \varphi(x) \to \varphi(x+1)}{\varphi(x)}$.

Здесь и далее $x \neq y$ есть сокращение от $\neg(x = y)$.

Запись $\dfrac{\varphi_1, \ldots, \varphi_n}{\varphi}$, 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 (7), связаны кванторами 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 называют *функциями означивания*, или просто *означиваниями*. Например, $\{x \mapsto 0, y \mapsto 1\}$ – состояние программы, в котором переменной x сопоставлено значение 0, а переменной y – значение 1.

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

---

[1] Термин «домен» часто используется в другом значении – область определения (бинарного отношения, отображения).

[2] Джузеппе Пеано (Giuseppe Peano; 1858–1932) – итальянский математик; внес значительный вклад в математическую логику и философию математики; наиболее известен как автор стандартной аксиоматизации арифметики натуральных чисел.

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 варианты исполнения одной и той же конструкции, из-за чего программа может выдавать разные результаты (особенно это характерно для параллельных программ). Для *детерминированных* программ $M[\![P]\!](s)$ является синглетоном (множеством с единственным элементом); для удобства будем считать, что в этом случае $M[\![P]\!](s)$ – 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 «тупой»: он не понимает «божественного предназначения» [Kpo01] исполняемой программы, но четко знает, в чем состоит то или иное действие, т. е. может действовать по инструкции.

Рассмотрим так называемую *структурную операционную семантику* – подход, предложенный Гордоном Плоткиным[1] [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: состояние меняется на s', а машине «остается» исполнить программу P'.

Чтобы выразить условие останова, вводится *пустая* программа ε:

$P; \varepsilon \equiv \varepsilon; P \equiv P$, для любой программы P.

Запись $\langle P, s\rangle \to \langle \varepsilon, s'\rangle$ означает, что исполнение программы P в состоянии s занимает один шаг, который состоит в изменении состояния на s' (на этом исполнение завершается).

Структурная операционная семантика определяет возможные переходы между конфигурациями с помощью дедуктивной системы специального вида (так называемой *системы переходов*): переход $\langle P, s\rangle \to \langle P', s'\rangle$ может произойти тогда и только тогда, когда он выводим в этой системе.

Система переходов языка while представлена в табл. 2.2 [Apt09]. В описании используются следующие обозначения:

---

[1] Гордон Плоткин (Gordon David Plotkin; род. в 1946 году) – британский ученый-информатик, известный своими работами в области семантики языков программирования; автор таких концепций, как язык программирования вычислимых функций (PCF, Programming Computable Functions) и структурная операционная семантика (SOS, Structural Operational Semantics); лауреат премии Милнера (2012).

- $s(t)$ – значение терма t в состоянии s (значение определяется индуктивно через значения подтермов с применением функций предметной области);
- $s[x := v]$ – состояние, полученное из состояния s путем сопоставления переменной x значения v:

  $s[x := v](x) = v$ и $s[x := v](y) = s(y)$, если $y \neq x$;

- $s \vDash B$ – истинность формулы B в состоянии s.

Таблица 2.2. Система переходов языка программирования while

(1)	$\langle \textbf{skip}, s\rangle \to \langle \varepsilon, s\rangle$	Оператор **skip** исполняется за один шаг и не меняет состояния программы
(2)	$\langle x := t, s\rangle \to \langle \varepsilon, s[x := s(t)]\rangle$	Исполнение оператора присваивания занимает один шаг: вычисляется значение правой части и сопоставляется левой части
(3)	$\dfrac{\langle P_1, s_1\rangle \to \langle P_2, s_2\rangle}{\langle P_1; P, s_1\rangle \to \langle P_2; P, s_2\rangle}$	Если переход допустим, то допустимы и все переходы, получаемые одинаковым «удлинением» пред- и постпрограмм
(4)	$\langle \textbf{if } B \textbf{ then } P_1 \textbf{ else } P_2 \textbf{ end}, s\rangle \to \langle P_1, s\rangle$, 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 состояния

Обратите внимание: в (5) и (7) используется запись $s \vDash \neg B$, а не $s \nvDash B$. Для классической двузначной логики это одно и то же, однако для трехзначной логики это не так: возможен вариант, когда значение истинности условия B не определено (например, вследствие зацикливания); в этом случае перехода нет.

Последовательностью переходов программы P в состоянии s называется конечная или бесконечная последовательность 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 $i \geq 0$ ($i < n - 1$, если последовательность конечна и n – ее длина).

Вычислением программы P в состоянии s называется последовательность переходов, которая не может быть продолжена, т. е. либо бесконечна, либо

заканчивается на конфигурацию вида $\langle \varepsilon, s' \rangle$. В последнем случае говорят, что вычисление *завершается* в состоянии s'. Говорят, что программа P *может зациклиться* в состоянии s, если существует бесконечное вычисление P в s.

Справедливо следующее 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:

$$M[\![P]\!](s) = \left\{ s' \middle| \langle P, s \rangle \xrightarrow{*} \langle \varepsilon, s' \rangle \right\} \cup \begin{cases} \{\omega\}, \text{ если } P \text{ может зациклиться в } s; \\ \emptyset \text{ в противном случае.} \end{cases}$$

Здесь $\xrightarrow{*}$ есть рефлексивно-транзитивное замыкание отношения переходов.

Напомним: *рефлексивно-транзитивным замыканием* бинарного отношения R называется минимальное рефлексивное и транзитивное отношение, содержащее R.

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. $\langle q := 0, r := a;$ **while** $r \geq b$ **do** $q := q + 1; r := r - b$ **end**, $(3, 2, -1, -1)\rangle$,
1. $\langle r := a;$ **while** $r \geq b$ **do** $q := q + 1; r := r - b$ **end**, $(3, 2, 0, -1)\rangle$,
2. $\langle$**while** $r \geq b$ **do** $q := q + 1; r := r - b$ **end**, $(3, 2, 0, 3)\rangle$,
3. $\langle q := q + 1; r := r - b;$ **while** $r \geq b$ **do** $q := q + 1; r := r - b$ **end**, $(3, 2, 0, 3)\rangle$,
4. $\langle r := r - b;$ **while** $r \geq b$ **do** $q := q + 1; r := r - b$ **end**, $(3, 2, 1, 3)\rangle$,
5. $\langle$**while** $r \geq b$ **do** $q := q + 1; r := r - b$ **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. Основополагающий вклад в эту область внес Тони Хоар[1] [Hoa69].

---

[1] Чарльз Энтони Ричард (Тони) Хоар (Charles Antony Richard Hoare; род. в 1934 году) – британский ученый в области информатики и вычислительной техники; автор алгоритма быстрой сортировки (quicksort), модели взаимодействующих последовательных процессов (CSP, Communicating Sequential Processes), аксиоматического подхода к формализации семантики языков программирования (логики Хоара); лауреат премии Тьюринга (1980).

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

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 – *программным контрактом*.

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

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

Заметим: речь идет о *частичной корректности* (методы доказательства завершимости и полной корректности программ рассматриваются в лекции 6).

Пример 2.4

Очевидно, что истинны следующие тройки Хоара:

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

□

Ниже в табл. 2.3 приведена аксиоматическая семантика языка while вместе с пояснениями к ее аксиомам и правилам вывода [Ben12].

В аксиоме оператора присваивания используется обозначение $\varphi[x := t]$ – результат подстановки в формулу φ терма t вместо каждого вхождения переменной x[1].

Пример 2.5

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

$[r := r - b]$ и $[q := q + 1]$.

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

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

Таблица 2.3. Аксиоматическая семантика языка программирования while

(1)	$\{\varphi\}\mathbf{skip}\{\varphi\}$	**Аксиома пустого оператора**: любое 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 стало истинным условие φ, необходимо и достаточно, чтобы перед исполнением оператора было истинно условие, полученное из φ заменой каждого вхождения левой части присваивания на правую часть
(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 φ, 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/или ослаблении постусловия

В результате мы получили ту же самую формулу $a = b \cdot q + r$ и тем самым доказали условие корректности

$$\{a = b \cdot q + r\}\ q := q + 1;\ r := r - b\ \{a = b \cdot q + r\}.$$

Обратите внимание: порядок применения подстановок обратен порядку присваиваний (хотя в данном случае это не важно – операторы независимы):

$$\underbrace{\{\varphi[x := t_2][x := t_1]\}x := t_1}_{\text{шаг 2}};\ \underbrace{\{\varphi[x := t_2]\}x := t_2\{\varphi\}}_{\text{шаг 1}}.$$

□

Остановимся подробнее на правиле вывода (5):

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

Условие φ называется *инвариантом цикла* (истинность этого условия сохраняется при исполнении тела цикла). Для проведения дедуктивной ве-

рификации нужно знать «информативные», как можно более «точные» инварианты, однако их поиск, вообще говоря, не поддается автоматизации. Более подробно это обсуждается в лекциях 3 и 5; пока лишь отметим, что тривиальные условия, *true* и *false*, являются инвариантами любого цикла, но они бесполезны для верификации.

Пример 2.6

Формула $a = b \cdot q + r$ является инвариантом цикла программы *DIV*, что было доказано в предыдущем примере:

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

Здесь в предусловии нет условия цикла, как того требует (5), однако ничто не мешает его добавить (предусловие можно усиливать)[1].

□

Программный контракт является описанием семантики программы, но это описание не обязано быть точным. Правило консеквенции позволяет усиливать предусловие и ослаблять постусловие без ущерба для корректности. Таким образом, для любой программы можно построить семейство контрактов разной степени детализации. В качестве семантики программы разумно рассматривать наиболее «сильный» из них. Эта идея была формализована Эдсгером Дейкстрой[2] на базе понятия *слабейшего предусловия* (*wp*, *weakest precondition*)[3] [Dij76].

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

- *false* – сильнейшая формула (пустое множество состояний);
- *true* – слабейшая формула (множество всех возможных состояний).

Слабейшим предусловием программы P относительно постусловия ψ (обозначение: $wp(P, \psi)$) называется слабейшая формула φ, для которой $\vDash \{\varphi\}P\{\varphi\}$[4].

Заметим, что для заданной программы отображение *wp* ставит в соответствие одному предикату (постусловию) другой предикат (слабейшее предусловие). Такие отображения называют *предикатными преобразователями*.

---

[1] Как правило, инварианты φ, такие что $\vDash \{\varphi\}P\{\varphi\}$, т. е. не учитывающие условие цикла, не являются «информативными».

[2] Эдсгер Вибе Дейкстра (Edsger Wybe Dijkstra; 1930–2002) – голландский ученый-информатик; один из разработчиков концепции структурного программирования, исследователь в области формальной верификации и параллельных вычислений; автор алгоритма поиска кратчайшего пути в графе; лауреат премии Тьюринга (1972).

[3] Другой вариант – *сильнейшее постусловие* (*sp*, *strongest postcondition*).

[4] Строго говоря, это определение *слабейшего свободного предусловия* (*wlp*, *weakest liberal precondition*); определение слабейшего предусловия, данное Дейкстрой, требует *полной* корректности.

Свойства предикатного преобразователя *wp* показаны в табл. 2.4.

Таблица 2.4. Свойства слабейшего предусловия

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

Обратите внимание: в свойстве (2) имеет место импликация, а не эквивалентность – в состояниях, в которых программа *P* зацикливается (если такие есть), истинно и предусловие $wp(P, \psi)$, и предусловие $wp(P, \neg\psi)$.

В табл. 2.5 приведены правила вычисления слабейшего предусловия для языка while [Ben12]. Они повторяют аксиомы и правила вывода аксиоматической семантики, за тем исключением, что здесь (по понятной причине) нет правила консеквенции.

Таблица 2.5. Правила вычисления слабейшего предусловия

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

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 и будем считать, что отображение *M* определяется для пары программа–постусловие:

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

Символ ⊥ («дно») интерпретируется как неопределенность поведения программы вследствие нарушения контракта, ω – как зацикливание (частичная корректность допускает зацикливание).

Пример 2.7

Вычислим слабейшее предусловие программы

$$x := x + y; \quad y := x - y; \quad x := x - y$$

относительно постусловия $(x = a) \wedge (y = b)$.

1. $wp(x := x + y; y := x - y; x := x - y, (x = a) \wedge (y = b)) =$
2. $wp(x := x + y; y := x - y, wp(x := x - y, (x = a) \wedge (y = b))) =$

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

□

Для того чтобы доказать условие корректности $\{\varphi\}P\{\psi\}$, нужно вычислить слабейшее предусловие $wp(P, \psi)$ и проверить общезначимость (тождественную истинность) импликации

$\varphi \rightarrow wp(P, \psi)$.

Задача проверки общезначимости логических формул является вычислительно трудной, порой неразрешимой, хотя для некоторых предметных областей известны эффективные разрешающие процедуры (см. лекцию 8). Главная проблема – вычисление слабейшего предусловия при наличии циклов (рекурсия в правиле 5).

2.2.3. Доказательное программирование

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

$P \equiv Q$; **while** B **do** R **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: $\vDash (\chi \wedge \neg B) \rightarrow \psi$.

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 конструируемого (еще не существующего) цикла – для заданных φ и ψ. Одна из стратегий поиска состоит в обобщении постусловия; например, путем замены некоторых констант на переменные (см. лекцию 7).

Подчеркнем: в методе Дейкстры контракт и инварианты циклов являются неотъемлемой частью программы.

Исследования по семантике программ начались в 1960-е годы и продолжаются по сей день. Подробное описание методов формализации языков программирования, включая не рассмотренную в лекции *денотационную семантику*[1], можно найти в специализированной литературе, например в книге [Gun92].

2.3. Вопросы и упражнения

1. Чем интерпретационный (операционный) подход к определению семантики языков программирования отличается от декларативного (аксиоматического)?
2. Выпишите сигнатуру программы P, реализующей алгоритм Евклида (см. лекцию 1). Расширьте эту сигнатуру функциональным символом НОД$^{(2)}$, определите аксиомы и правила вывода предметной области.
3. Докажите утверждение о единственности вычисления любой while-программы в произвольном начальном состоянии.
4. Постройте вычисление программы целочисленного деления *DIV* для $a = 7$ и $b = 3$ (начальные значения переменных q и r могут быть любыми).

---

[1] 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-х – начале 1970-х годов в трудах Дана Скотта (Dana Scott; род. в 1932 году) и Кристофера Стрейчи (Christopher Strachey; 1916–1975). Денотационная семантика языка программирования while рассматривается, например, в книге [Mit96].

5. Постройте вычисления программ P и Q, реализующих алгоритм Евклида (см. лекцию 1), для $a = 14$ и $b = 21$.
6. Опишите семантику оператора недетерминированного выбора **choice**, используя операционный и аксиоматический подходы.
7. Опишите семантику оператора цикла **repear**–**until**, используя операционный и аксиоматический подходы.
8. 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-й оператор, если истинно условие B). Можно ли это сделать, не выходя за рамки структурной операционной семантики?
9. Докажите условие корректности $\{true\}\ x := a,\ y := b\ \{(x = a) \wedge (y = b)\}$, используя определенные в лекции аксиомы и правила вывода.
10. Предложите инвариант цикла для программы целочисленного деления *DIV*.
11. Предложите инварианты циклов для программ P и Q, реализующих алгоритм Евклида (см. лекцию 1). Совпадают ли эти инварианты?
12. Докажите свойства дистрибутивности (относительно конъюнкции), двойственности и монотонности слабейшего предусловия (см. табл. 2.4).
13. Докажите свойство дистрибутивности слабейшего предусловия относительно дизъюнкции: $wp(P, \psi) \vee wp(P, \chi) \rightarrow wp(P, \psi \vee \chi)$.
14. Какие состояния программы P характеризует предусловие $wp(P, false)$?
15. Вычислите $wp(x := x + a;\ y := y - 1,\ x = (b - y) \cdot a)$.
16. Постройте с помощью метода Дейкстры программу, вычисляющую сумму элементов целочисленного массива заданной длины.
17. Предложите способ вычисления сильнейшего постусловия программы (*sp*, *strongest postcondition*). Указание: приведите программу к SSA-форме[1].
18. Сформулируйте свойства предикатного преобразователя *sp*.
19. Какие состояния программы P характеризует постусловие $sp(P, true)$?
20. Вычислите $sp(x := x + a;\ y := y - 1,\ x = (b - y) \cdot a)$.

[1] 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, $sp(P, \varphi) = (\varphi[оригиналы := версии_н] \wedge ssa(P))[версии_к := оригиналы]$, где $версии_н$ и $версии_к$ – начальные и конечные версии переменных соответственно. В задаче требуется найти способ построения формулы $ssa(P)$.

Лекция 3

Дедуктивная верификация последовательных программ

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

Р. М. Смаллиан[1].
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 инварианты циклов не выводимы из структуры программы. Задание инвариантов дает набросок доказательства корректности, проверку которого можно автоматизировать.

[1] Есть несколько переводов фамилии Smullyan на русский язык: Смаллиан, Смальян, Смольян, Шмульян. Последний вариант – первоначальная фамилия, которую имели родители Рэймонда Меррилла Смаллиана (1919–2017) до эмиграции в США.

3.1. Дедуктивные системы и доказательства

Задача верификации – доказать корректность программы[1], т. е. соответствие программы требованиям. В классическом подходе, рассматриваемом нами, требования задаются с помощью программного контракта (пары условий), а верификация состоит в доказательстве частичной или полной корректности программы относительно этого контракта. Когда говорится о доказательстве, предполагается, что есть *дедуктивная система*, 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») и называется *аксиоматической семантикой*, или *логикой Хоара* (см. лекцию 2).

Итак, мы имеем классическую задачу математической логики: имеются *дедуктивная система* (множество аксиом и правил вывода)[2] 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*. Формула φ, для которой существует доказательство, называется *доказуемой* (обозначение: $\vdash\varphi$) [Ben12]. Иногда для краткости мы будем опускать символ $\vdash$.

Доказательство нагляднее представлять в виде дерева (точнее, ациклического графа): листовые вершины – это используемые аксиомы; нелистовые – результаты применения правил вывода; корень – доказанная теорема (рис. 3.1).

---

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

[2] Более формально понятие дедуктивной системы определяется в лекции 8.

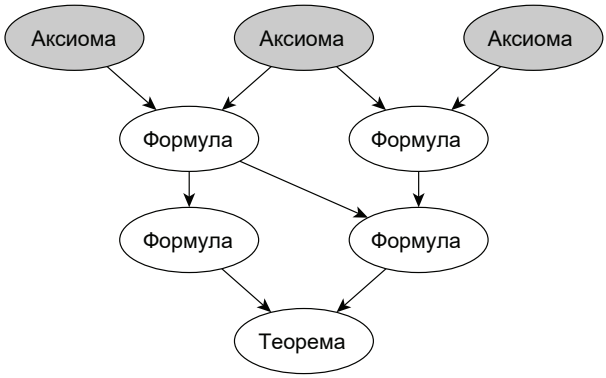


Рис. 3.1 ❖ Дерево логического вывода (доказательство)

3.1.1. Примеры дедуктивных систем

Пожалуй, самой известной дедуктивной системой является геометрия Евклида, описанная в «Началах» в III веке до н. э. [Евк50] и формализованная Давидом Гильбертом[1] [Hil99].

Обратимся к другому примеру – системе Генцена[2] [Ben12]. Напрямую эта система не связана с программированием; цель примера – проиллюстрировать понятие формального логического вывода (построение доказательств без «понимания» предметной области, как это делает компьютер).

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 (высказывание и его отрицание)[3];

---

[1] Давид Гильберт (David Hilbert; 1862–1943) – выдающийся немецкий математик, работавший в самых разных сферах математики; наиболее известен работами в области оснований математики и функционального анализа, а также как автор списка из 23 математических проблем (проблемы Гильберта, 1900); лауреат многих премий, в том числе имени Н. И. Лобачевского (1903).

Говоря о геометрии Евклида, следует отметить и другие формализации (аксиоматики): Тарского (Alfred Tarski; 1901–1983), Вейля (Hermann Weyl; 1885–1955), Биркгофа (George David Birkhoff; 1884–1944).

[2] Герхард Карл Эрих Генцен (Gerhard Karl Erich Gentzen; 1909–1945) – немецкий математик и логик; внес большой вклад в исследование оснований математики и развитие теории доказательств; создатель исчисления секвенций.

[3] Множества литералов называют *клаузами*. Клауза $\{l_1, \ldots, l_n\}$ соответствует дизъюнкции $l_1 \vee \ldots \vee l_n$. Таким образом, аксиомы системы Генцена – это дизъюнкции, включающие противоположные высказывания: $p \vee \neg p \vee \ldots$.

- *α-правила* (см. табл. 3.1):

$$\frac{U \cup \{\alpha_1, \alpha_2\}}{U \cup \{\alpha\}},$$

где U – множество формул, не содержащее α_1 и α_2;

- *β-правила* (см. табл. 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-правила 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, $\wedge$);
4. $\{\neg(p \wedge q), q\}$ α-правило (2, $\wedge$);
5. $\{\neg(p \wedge q), (q \wedge p)\}$ β-правило (3 и 4, $\wedge$);
6. $\{\neg(p \wedge q) \vee (q \wedge p)\}$ α-правило (5, $\vee$).

Обратите внимание на два момента:

1) процедура логического вывода исключительно формальная (проверка доказательства может быть легко выполнена компьютером);
2) подобрать нужные аксиомы и правила вывода не так-то просто (поиск доказательства трудно автоматизировать).

□

3.1.2. Значимость и полнота дедуктивных систем

Разумно использовать такие системы, в которых *доказуемость* утверждений соотносится с их *истинностью*.

- Дедуктивная система называется *значимой* (*sound*), или *семантически непротиворечивой*, если все утверждения, доказуемые в этой системе, являются истинными.

- Дедуктивная система называется *полной* (*complete*), если в ней могут быть доказаны все истинные утверждения (формулируемые на соответствующем языке).

Упомянутые выше дедуктивные системы Евклида (аксиоматика Гильберта) и Генцена являются как значимыми, так и полными [Ben12].

Работать имеет смысл только со значимыми системами: в незначимой системе могут быть выведены ложные утверждения (например, программа, содержащая ошибки, может быть классифицирована как корректная). Полнота дедуктивной системы желательна, но не обязательна: для неполной системы существуют истинные утверждения, которые невозможно в ней вывести (например, могут быть корректные программы, корректность которых невозможно доказать средствами используемой системы). Неполные системы часто встречаются в математике; к ним, например, относится аксиоматика Пеано [Pea89], формализующая арифметику натуральных чисел (см. лекцию 2)[1]. Более подробно тема дедуктивных систем и автоматизированного доказательства теорем раскрывается в лекции 8.

В контексте верификации программ характеристики «значимость» и «полнота» часто используют в ином смысле [Tre08]. Как мы знаем, результатом верификации является *вердикт*: либо положительный («ошибок нет»), либо отрицательный («ошибки есть»)[2].

- Метод верификации называется *значимым*, если из того, что вердикт отрицательный (выдается сообщение об ошибке), следует, что программа ошибочна (невозможны *false positives* – см. лекцию 11).
- Метод верификации называется *полным*, если из того, что вердикт положительный (нет сообщений об ошибках), следует, что программа корректна (невозможны *false negatives*).

3.2. Дедуктивная верификация программ

Под *дедуктивной верификацией* понимается поиск доказательства условия корректности программы в дедуктивной системе соответствующего языка (логике Хоара). Поиск доказательства является творческим процессом, допускающим эффективное[3] решение лишь в частных случаях. Например, для

---

[1] Неполнота аксиоматики Пеано является следствием теоремы Генцена о ее непротиворечивости и теоремы Гёделя (Kurt Gödel, 1906–1978), утверждающей, что дедуктивная система, достаточная для представления арифметики, не может быть одновременно непротиворечивой и полной.

[2] Считаем, что неопределенного вердикта нет.

[3] Мы специально не уточняем, что понимается под *эффективностью*: существование какого-нибудь алгоритма (как в математической логике) или алгоритма из определенного класса.

логики высказываний проверка выполнимости (непротиворечивости) или общезначимости (тождественной истинности) формулы является NP-полной задачей [Pap82]; для теорий первого порядка проверка и вовсе может быть алгоритмически неразрешимой (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? В этом случае, очевидно, доказательства ее корректности не существует (считаем, что дедуктивная система значима). Как отличить эту ситуацию от ситуации, когда доказательство существует, но мы не можем его найти? Общего рецепта нет. Можно порекомендовать декомпозировать доказываемое условие на части и попытаться доказать каждую из них по отдельности. Для тех частей, что не будут доказываться, можно попробовать построить контрпримеры (интерпретации, в которых утверждения ложны).

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). Таким образом,

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

Другими словами, аксиоматическая семантика согласована с операционной.

Проиллюстрируем этот факт на примере присваивания. Исполнение оператора $x := t$ описывается переходом $\langle x := t, s\rangle \to \langle \varepsilon, s[x := s(t)]\rangle$. Для произвольного условия φ имеем:

$$\vdash \{\varphi[x := t]\}\ x := t\{\varphi\}.$$

Зафиксируем произвольное состояние s, такое что $s \vDash \varphi[x := t]$, и покажем, что

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

Это и будет означать, что $\vDash \{\varphi[x := t]\}\ x := t\{\varphi\}$.

Не ограничивая общности рассуждений, будем считать, что условие φ имеет вид $p(x)$, где p – одноместный предикатный символ (обозначим интерпретирующее его отношение символом $[\![p]\!]$). Приведенная ниже цепочка эквивалентностей доказывает требуемое утверждение:

$$s \vDash p(x)[x := t] \;\Leftrightarrow\; s \vDash p(t) \;\Leftrightarrow\; s(t) \in [\![p]\!] \;\Leftrightarrow\; s[x := s(t)] \vDash p(x).$$

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 табл. 3.2.

Таблица 3.2. Доказательство корректности программы целочисленного деления

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

□

Доказательства корректности программ являются громоздкими (пример выше это демонстрирует). Для их декомпозиции на сравнительно простые фрагменты используются так называемые *аннотированные программы* – программы, в которые добавлены *утверждения* [Gri81]. Утверждение – это 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**. Будем считать, что инварианты 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}      // обычное 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 фрагмента относительно ограничивающих его утверждений (в частности, в инвариантности «инвариантов» циклов). Эта работа может быть автоматизирована при условии разрешимости предметной области (см. лекцию 8).

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 этой идее основана работа модулей Jessie и AstraVer 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. $\{(a \geq 0) \wedge (b > 0)\}\ q := 0; r := a\ \{(a = q \cdot b + r) \wedge (0 \leq r \leq a)\}$;
2. $\{(a = q \cdot b + r) \wedge (0 \leq r \leq a)\}\ q := q + 1; r := r - b\ \{(a = q \cdot b + r) \wedge (0 \leq r \leq a)\}$;
3. $((a = q \cdot b + r) \wedge (0 \leq r \leq a) \wedge (r < b)) \to ((a = q \cdot b + r) \wedge (0 \leq r \leq a))$.

□

Вопрос состоит в том, как найти инварианты циклов. Это краеугольный камень всей дедуктивной верификации. Поясним, в чем трудность. Синтез содержательного инварианта цикла $W \equiv$ **while** B **do** P **end** сводится к построению слабейшего предусловия:

$wp(W, \psi)$ – инвариант цикла W.

В самом деле,

$$\varphi \equiv wp(W, \psi) = (\neg B \wedge \psi) \vee \left(B \wedge wp\left(P, \underbrace{wp(W, \psi)}_{\varphi} \right) \right) = (\neg B \wedge \psi) \vee (B \wedge wp(P, \varphi)).$$

Отсюда: $(B \wedge \varphi) = (B \wedge wp(P, \varphi))$; $(B \wedge \varphi) \to wp(P, \varphi)$, а значит, $\{B \wedge \varphi\}P\{\varphi\}$.

Слабейшее предусловие определяется рекурсивно:

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

Таким образом, построение слабейшего предусловия (а вместе с ним и инварианта цикла) приводит, вообще говоря, к бесконечному вычислению. Важно отметить, что итерационный процесс построения слабейшего предусловия является в известном смысле *монотонным*: если отбросить дизъюнктивный член, зависящий от исходной программы, то в конце каждого шага получается более слабая формула по сравнению с той, что была в начале:

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

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

   $\underline{(\neg B \wedge \psi) \vee (B \wedge wp(P, \neg(B \wedge \psi)))} \vee \big(B \wedge wp\big(B \wedge wp(P, wp(W, \psi))\big)\big) =$
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. $(\neg(x > 0) \wedge (x = 0)) \vee ((x > 0) \wedge wp(x := x - 1; W, x = 0)) =$
3. $\underline{(x = 0)} \wedge ((x > 0) \wedge wp(x := x - 1; wp(W, x = 0))) =$
4. $\underline{(x = 0)} \vee ((x > 0) \wedge wp(W, x = 0)[x := x - 1]) =$
5. $\underline{(x = 0)} \vee ((x > 0) \wedge wp(W, x = 1)) =$
6. $\underline{(x = 0)} \vee ((x = 1) \vee (x > 1) \wedge wp(x := x - 1; W, x = 1)) =$
7. $\underline{(x = 0) \vee (x = 1)} \vee ((x > 1) \wedge wp(W, x = 1)[x := x - 1]) =$
8. ...

9. $\underline{(x = 0) \vee (x = 1) \vee (x = 2) \vee \ldots} =$ бесконечное число шагов
10. $x \in \mathbb{N}_0$ обобщение

□

Напоследок рассмотрим стратегии верификации 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 с циклами.

3.2.2. Верификация линейных программ

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

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

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

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

   $$wp(P, \psi) \equiv \psi[x_n := t_n] \ldots [x_1 := t_1];$$

2) доказать, что исходное предусловие φ не слабее вычисленного:

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

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

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), для каждого пути построить условие прохождения по нему и доказать соответствующее условие корректности. Таким образом, задача сводится к предыдущей. Однако проще поступить так: вычислить слабейшее предусловие для программы в целом и работать с ним (рассматривать отдельные пути или их подмножества имеет смысл, если не доказывается «большое» условие, построенное для всей программы).

Пример 3.7

Обратимся к более крупному фрагменту программы сортировки массива:

```
{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:

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

2) докажем условие корректности, соответствующее пути:

$\{true \wedge (a > b)\}$
$x := a; y := b; t := x; x := y; y := t$
$\{(x \le y) \wedge ((x = a) \wedge (y = b)) \vee ((x = b) \wedge (y = a))\}$:

a) вычислим слабейшее предусловие:

$$wp(x := a; y := b; t := x; x := y; y := t, (x \le y) \wedge ((x = a) \wedge (y = b)) \vee ((x = b) \wedge (y = a))) =$$
$$(x \le y) \wedge (((x = a) \wedge (y = b)) \vee ((x = b) \wedge (y = a)))[y := t][x := y][t := x][y := b][x := a] =$$
$$(b \le a) \wedge (((b = a) \wedge (a = b)) \vee ((b = b) \wedge (a = a))) =$$
$$(b \le a);$$

b) докажем утверждение $(a > b) \to (b \le a)$.

□

3.2.4. Верификация циклических программ

Пусть для простоты верифицируемая программа содержит один оператор цикла:

$P \equiv Q$; **if** 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 $(\chi \wedge \neg B) \to \psi$.

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 [Мсс62].

Сигнатура теории содержит два функциональных символа:

- $read: A \times I \to E$ (соответствует операции [ ] в правой части присваивания: $x := a[i]$);
- $write: A \times I \times E \to A$ (соответствует операции [ ] в левой части присваивания: $a[i] := x$).

Здесь I, E и A – это типы индекса, элемента и массива соответственно. Семантика задается следующими аксиомами[1]:

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

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 $(s = sum(x, n)) \wedge (n \leq N)$:

1) проверим, что это действительно инвариант:

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

   a) $wp(s := s + x[n];\ n := n + 1, (s = sum(x, n)) \wedge (n \leq N)) =$
   $((s = sum(x, n)) \wedge (n \leq N))[n := n + 1][s := s + x[n]] =$
   $(s + x[n] = sum(x, n + 1)) \wedge (0 \leq n + 1 \leq N) =$
   $(s = sum(x, n)) \wedge (n + 1 \leq N);$

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

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

   a) $wp(s := 0, n := 0, (s = sum(x, n)) \wedge (n \leq N)) =$
   $((s = sum(x, n)) \wedge (n \leq N))[n := 0][s := 0] =$
   $(0 = sum(x, 0)) \wedge (0 \leq N)) =$
   $N \geq 0;$

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

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

□

[1] Предполагается, что все используемые переменные связаны кванторами всеобщности.

Более подробно дедуктивная верификация программ с циклами рассматривается в лекциях 5 и 6.

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 (анализа программ). Приведите примеры значимых и незначимых, полных и неполных методов.
6. Оцените возможность автоматизации дедуктивной верификации программ. Какие проблемы возникают на пути к автоматизации?
7. Рассмотрите возможность дедуктивной «фальсификации» программ, т. е. 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. Предложите инварианты циклов для двух реализаций алгоритма Евклида из лекции 1 (программ P и Q). Докажите их частичную корректность. Подсказка: введите функциональный символ НОД$^{(2)}$ вместе с необходимыми аксиомами.

10. Напишите пред- и постусловие для программ сортировки числовых массивов. Докажите частичную корректность программы, реализующей «метод пузырька».
11. 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}
```

12. 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 $[0, 2^{32} - 1]$).

13. 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 *maxPrimerFactor*(*a*) – максимальный простой делитель целого числа $a > 1$.

Лекция 4

Инструменты дедуктивной верификации программ

Час работы научит большему, чем день объяснений.

Ж.-Ж. Руссо.
Эмиль, или О воспитании

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), а также модулей (plugins) Jessie и AstraVer, автоматизирующих дедуктивную верификацию C-программ на соответствие ACSL-спецификациям. Изложение базируется на примерах (подробное описание языка ACSL и основанных на нем инструментов можно найти в 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] вместе с модулями (plugins) дедуктивной верификации Jessie [Jess] и AstraVer[1] [AV].

Язык ACSL предназначен для формальной спецификации программ, написанных на языке C [Ker88]. Язык позволяет в виде так называемых *аннотаций* определять пред- и постусловия функций, утверждения, инварианты циклов и другие элементы спецификаций.

ACSL-аннотации бывают двух видов: *внешние* (*global annotations*), описывающие функциональность программы или библиотеки на уровне программного интерфейса, и *внутренние* (*statement annotations*), привязанные к конкретной реализации.

К внешним аннотациям относятся:

- *контракты функций* – пред- и постусловия функций;
- *глобальные инварианты* – ограничения целостности данных;
- *инварианты типов данных* – ограничения на типы данных (так называемые *ограничения подтипов*);
- *вспомогательные определения* – спецификационные переменные, предметные области (функциональные и предикатные символы вместе с явными или аксиоматическими описаниями) и леммы.

К внутренним аннотациям относятся:

- *утверждения* – условия, истинность которых должна обеспечиваться в определенных точках программы;
- *аннотации циклов* – инварианты и другие свойства циклов;
- *контракты операторов* – пред- и постусловия операторов или блоков операторов программы;
- *вспомогательный код* (*ghost code*) – действия, модифицирующие спецификационные переменные.

Сразу оговоримся: мы не преследуем цель охватить все средства ACSL (для этого есть документация); наша задача – познакомить с основными возможностями языка, достаточными для верификации несложных, но вполне реалистичных функций.

4.1.1. Базовые сведения о языке

ACSL-спецификации пишутся непосредственно в исходном коде программы в виде комментариев, имеющих вид `/*@ … */` или `//@ …` (первым символом комментария-аннотации обязательно должен быть `@`: при наличии пробела между `/*` или `//` и `@` текст комментария игнорируется, как это происходит с обычным комментарием).

Лексика языка в основном заимствована из C. Отличия состоят в следующем:

- идентификаторы можно начинать с обратного слеша (\);

[1] AstraVer – существенно переработанный в ИСП РАН вариант модуля Jessie.

- в язык добавлены:
  - логические связки импликации и эквивалентности (`==>` и `<==>`);
  - кванторы существования и всеобщности (`\exists` и `\forall`);
  - встроенные функции (`\min`, `\max`, `\abs` и т. п.).

Операции языка ACSL резюмированы в табл. 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
||	|	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 формулы), id – идентификатор		
Операции модификации структур и массивов		
{s **for** $.field = e$}		модификация структуры
{a **for** [i]= e}		модификация массива
Здесь s – структура, a – массив, $field$ – имя поля структуры, i – индекс элемента массива, e – выражение		

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

запись $e_1 \underset{<_1}{\geq} e_2 \underset{<_2}{\geq} e_3 \underset{<_3}{\geq} \ldots \underset{<_{n-1}}{\geq} e_n$,

где $\underset{<_i}{\geq}$ – символы операций сравнения с одинаковой «ориентацией» (либо они все из множества {==, <=, <}, либо из множества {==, >=, >}), эквивалентна записи

$$\left(e_1 \underset{<_1}{\geq} e_2\right) \&\& \left(e_2 \underset{<_2}{\geq} e_3\right) \&\& \ldots \&\& \left(e_{n-1} \underset{<_{n-1}}{\geq} e_n\right).$$

Приоритеты операций ACSL согласованы с приоритетами, принятыми в C. Побитовые операции имеют больший приоритет по сравнению с логическими. Приоритет кванторов существования и всеобщности, как и конструкций связывания переменных и именования выражений, ниже приоритета условного оператора (*B* ? e_1 : e_2), при этом `\forall`, `\exists` и `\let` имеют одинаковый приоритет, больший приоритета *id*: *E*.

4.1.2. Типы данных

Система типов данных ACSL включает как встроенные типы C, называемые *машинными*, так и *математические типы*: **boolean**, **integer** и **real**. Считается, что машинные типы являются подтипами математических: типы **char**, **short**, **int**, **long** (знаковые и беззнаковые варианты) – подтипы **integer**, а типы **float** и **double** – подтипы **real**[1]. Допускается преобразование значений математических типов в значения машинных типов. Значения типа **integer** неявно приводятся к значениям типа **boolean**, однако для преобразования из **boolean** в **integer** нужно использовать оператор приведения типов. Побитовые операции определены не только для машинных целочисленных типов, но и для типа **integer** (значения последнего можно рассматривать как двоичные последовательности бесконечной длины в дополнительном коде).

Из базовых типов данных в ACSL можно строить составные типы: *структуры* и *массивы*. Значения составных типов можно проверять на равенство/неравенство, использовать в качестве параметров функций и т. д. При декларации массива указывается его длина; узнать длину массива можно с помощью встроенной функции `\length`. В остальном все похоже на язык C.

Пример 4.1

Рассмотрим примеры определения составных типов данных [Bau10].

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

Обратите внимание: внутри аннотаций можно использовать комментарии; они начинаются с двойного слеша (//).

□

4.1.3. Вспомогательные определения

Часто в спецификациях программ используются вспомогательные определения (переменные, предикаты и функции), а также леммы, упрощающие

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

жизнь системам автоматизированного доказательства теорем. В этом разделе мы на нескольких примерах рассмотрим такие средства.

Для выражения некоторых свойств (например, инвариантов циклов) бывает удобным ввести специальные переменные и обновлять их значения в определенных точках программы.

Пример 4.2

В этом примере объявляется спецификационная переменная `count` типа **`integer`**.

```
/*@ // объявление спецификационной переменной
  @ logic integer count = 0;
  @*/
```

Для изменения значения этой переменной (в том или ином месте программы) нужно использовать *ghost-код*, например:

```
//@ ghost count++;
```

□

В спецификациях могут определяться (а потом использоваться) предикаты и функции. Эти определения могут быть как *явными*, в виде формул, так и *аксиоматическими*.

Пример 4.3

Ниже определяется предикат `IsMultiple`, проверяющий, что целое число `x` является делителем целого числа `a`.

```
/*@ predicate IsMultiple(integer a, integer x) =
  @   a % x == 0;
  @*/
```

□

Пример 4.4

В этом примере аксиоматически (с помощью трех аксиом) задается функция `gcd`, вычисляющая НОД двух натуральных чисел.

```
/*@ axiomatic GCD {
  @   logic integer gcd(integer a, integer b);
  @   axiom gcd_equal:
  @     \forall integer x;
  @       (x > 0) ==> (gcd(x, x) == x);
  @   axiom gcd_comm:
  @     \forall integer x, y;
  @       (x > 0 && y > 0) ==> (gcd(x, y) == gcd(y, x));
  @   axiom gcd_add:
  @     \forall integer x, y;
```

```
@       (x > 0 && y > 0) ==> (gcd(x + y, y) == gcd(x, y));
@ }
@*/
```

Первая аксиома гласит, что НОД двух одинаковых чисел совпадает с ними: НОД(x, x) = x. Вторая аксиома – это аксиома коммутативности: НОД(x, y) = НОД(y, x). Наконец, третья аксиома утверждает, что если из большего (первого) числа вычесть меньшее (второе), то НОД не изменится: ($x > y$) → НОД(x, y) = НОД($x - y$, y).

□

Мы неоднократно говорили, что доказательство теорем сложно автоматизировать (см. лекцию 8). Чтобы помочь системам доказательства теорем (подсказать, в каких направлениях стоит двигаться при построении логического вывода, а каких стоит избегать), применяются леммы.

Пример 4.5

Определяемая ниже лемма `NotMultiple` задает достаточные условия неделимости. В ней (помимо известного нам `IsMultiple`) используется предикат `IsPrime`, проверяющий натуральное число на простоту.

```
/*@ lemma NotMultiple:
  @   \forall integer a, x;
  @     (a > 0 && x > 0 && IsMultiple(a, x) && IsPrime(x)) ==>
  @     (\forall integer y; (0 < y < x) ==> !IsMultiple(a + y, x));
  @*/
```

□

4.1.4. 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 клаузах **`ensures`** можно использовать следующие конструкции:

- `\old(`*e*`)` – значение выражения *e* до вызова специфицируемой функции (при записи без `\old` значение выражения вычисляется на состоянии после вызова);
- `\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

указаны в клаузе **assigns**. Это можно сделать как с помощью L-выражений языка C, так и используя специальные конструкции ACSL, например

`x[0..n-1]` или `*(x+(0..n-1))`.

Обе записи обозначают область, образованную *n* элементами массива *x*. Если функция не модифицирует память, в **assigns** следует указать `\nothing`. Если блоков несколько, они перечисляются через запятую или записываются в отдельных клаузах.

В языке есть встроенный предикат `\valid`, проверяющий доступность (выделенность) блока памяти по заданному адресу или диапазону адресов, например

`\valid(x)` или `\valid(x+(0..n-1))`.

Обратите внимание: предикат `\valid`, в отличие от **assigns**, принимает адрес.

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 4.6

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 ячейку памяти, в которую записывается остаток от деления; возвращаемым значением 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; // общее условие на q и *r
  @ 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 описывает определенный вариант поведения функции: обработку некорректных

входных данных, непредвиденную ошибку, нормальное исполнение и т. п. В случае если допускаются разные варианты поведения, имеет смысл представить спецификацию функции в следующей форме (для наглядности здесь приведены только 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 активации ветви
  @   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);
  @*/
```

Если в описании ветви отсутствует клауза **assumes**, это равносильно наличию **assumes** \true.

Пример 4.7

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, что упорядоченность массива требуется не в предусловии, а в условии ветви `failure`: если функция возвратила допустимый индекс, гарантируется, что по этому индексу находится искомый элемент (не важно, отсортирован массив или нет); если же массив не отсортирован, функция не обязана осуществлять поиск (допустима ситуация, когда искомый элемент есть, но функция возвращает `-1`).

□

ACSL допускает *неполные спецификации* (в которых дизъюнкция условий всех ветвей не покрывает область определения функции), а также *спецификации с перекрывающимися ветвями* (в которых условия разных ветвей могут быть одновременно истинными). В примере выше спецификация полна, но имеет перекрывающиеся ветви: условие ветви `success` (`\true`, поскольку **`assumes`** отсутствует), очевидно, полностью покрывает область определения функции и совместимо с условием ветви `failure`.

Для контроля за полнотой и разделимостью ветвей функциональности предназначены клаузы **`complete behaviors`** и **`disjoint behaviors`** соответственно.

4.1.5. Утверждения и аннотации циклов

Утверждения (*assertions*) задают ограничения на состояние, которые должны быть истинными в определенных точках программы. В ACSL утверждения начинаются с 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`**, часто дополняемая клаузой **`loop assigns`**:

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

Как и утверждения, инварианты можно определять для избранных ветвей:

```
/*@ for b_1, ..., b_n: loop invariant φ; */
```

Отметим также, что для одного цикла можно написать несколько клауз **loop invariant**: инвариант образуется путем конъюнкции всех указанных условий.

Язык программирования C сложнее языка while, используемого нами для разъяснения основ дедуктивной верификации: в частности, в нем есть несколько операторов циклов (**while**, **do-while**, **for**) и допустимы условия с побочными эффектами. С учетом этого клаузы **loop invariant** φ и **loop assigns** *L* интерпретируются так:

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.

Приведем примеры функций на языке C и их аннотаций на ACSL.

Пример 4.8

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 4.9

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;
}
```

□

Для доказательства завершимости и полной корректности программ используются клаузы **loop variant**, задающие *оценочные функции* циклов, т. е. функции, значения которых монотонно изменяются на каждой итерации и при этом остаются ограниченными некоторой величиной (см. лекцию 6).

4.2. Обзор платформы Frama-C

Для автоматизации проверки C-программ на соответствие ACSL-спецификациям можно использовать платформу статического анализа Frama-C [FraC] вместе с модулем (plugin) Jessie [Jess], [Mar18] или AstraVer [AV]. И тот, и другой модули отвечают за дедуктивную верификацию.

4.2.1. Платформа Frama-C

Платформа Frama-C предназначена для статического анализа программ на языке C. Под *статическим анализом* понимается вычисление характеристик программ (в том числе соответствие функциональным требованиям) без исполнения на компьютере, т. е. путем вывода из исходного или бинарного кода [Nie99]. Платформа разработана на языке OCaml [Min13] и распростра-

няется с открытым исходным кодом. К созданию Frama-C причастны две французские организации: CEA-LIST (Laboratoire d'Intégration de Systèmes et des Technologies) и INRIA (Institut National de Recherche en Informatique et en Automatique).

Платформа состоит из ядра и подключаемых модулей-анализаторов. Ядро управляет анализом программ, сохраняет информацию, полученную в ходе анализа, и предоставляет модулям доступ к этой информации. Есть несколько предопределенных анализаторов, но, что важно, платформу можно расширять новыми модулями, при этом их можно строить на базе имеющихся (взаимодействие между модулями осуществляется через известные интерфейсы ядра). Основными анализаторами Frama-C являются:

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

Мы будем использовать модуль Jessie или его усовершенствованный вариант AstraVer.

4.2.2. Модули Jessie/AstraVer и платформа Why3

Модули Jessie и AstraVer автоматизируют верификацию C-программ, снабженных ACSL-аннотациями. Оба являются препроцессорами (front-ends) 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: модуль Jessie реализован с помощью Why версии 2, но подготавливает данные для Why версии 3 (таким образом, для установки Jessie требуются обе версии Why [Krak]); в AstraVer эта проблема устранена (используется только Why3).

Задачами связки Jessie/AstraVer – 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).

Jessie и AstraVer решают первую задачу; за оставшиеся две отвечает Why3.

Платформа Why3 базируется на языке WhyML, позиционирующемся и как язык программирования, и как язык промежуточного представления кода. В платформе поддерживаются разные типы данных и предметные области (целочисленная и вещественная арифметика, теории массивов, очередей, отображений и др.). Для проверки условий верификации используются внешние инструменты доказательства теорем (provers): *автоматические* (Alt-Ergo, CVC4, Yices, Z3) и *интерактивные* (Coq, PVS).

Запуск Frama-C с модулем Jessie или AstraVer осуществляется следующими командами:

```
frama-c -jessie имя-файла
frama-c -av имя-файла
```

При запуске 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.

Чтобы система Why3 «подцепила» имеющиеся пруверы, нужно выполнить команду

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

Более подробную информацию об инструментах Frama-C, Jessie, AstraVer и Why3 можно найти по указанным ранее ссылкам. Рекомендуем также ознакомиться с работой «*ACSL by Example*» [Bur16], рассматривающей практические аспекты спецификации и верификации C-функций и содержащей большое число примеров.

4.3. Вопросы и упражнения

1. Укажите основные конструкции языка ACSL. С помощью каких средств определяются программные контракты? Как записываются инварианты циклов?
2. Что такое ветвь функциональности? Дайте определения полного множества ветвей и разделимого множества ветвей.
3. Определите аксиоматически предикаты *IsMultiple*(*a*, *x*) и *IsPrime*(*x*).
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) алгоритм Евклида;
   b) нахождение суммы элементов числового массива;
   c) сортировка числового массива «методом пузырька».

   Для указанных функций определите на языке ACSL инварианты циклов.
6. Докажите с использованием Frama-C и Jessie (или AstraVer) частичную корректность следующих функций:
   a) целочисленное деление (функция *idiv*);
   b) дихотомический поиск в упорядоченном массиве (функция *bsearch*).
7. Проверифицируйте с помощью Frama-C и Jessie (или AstraVer) ваши реализации следующих функций:

a) нахождение суммы элементов числового массива;
b) сортировка числового массива «методом пузырька».

8. Специфицируйте на языке ACSL и проверифицируйте с помощью Frama-C и Jessie (или AstraVer) следующие функции:
   a) нахождение максимального делителя натурального числа, отличного от него самого:

```
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 меньшей 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

Силлогизмы состоят из предложений, предложения – из слов, а слова суть знаки понятий. Поэтому если сами понятия, составляя основу всего, спутаны и необдуманно отвлечены от вещей, то нет ничего прочного в том, что построено на них. Поэтому единственная надежда – в истинной индукции.

Ф. Бэкон.
Новый Органон

Рассматривается один из первых подходов к формальной верификации программ – метод Флойда. Подход состоит из двух частей: первая (метод индуктивных утверждений) предназначена для доказательства частичной корректности, т. е. корректности без учета завершимости; вторая (метод фундированных множеств) используется для доказательства завершимости и полной корректности (этой части посвящена лекция 6). Метод Флойда имеет дело с блок-схемами и позволяет анализировать программы с общим видом потока управления, в том числе не отвечающие принципам структурного программирования[1]. Подход состоит в рассечении циклов, сопоставлении точкам сечения утверждений, построении условий верификации, связывающих утверждения соседних точек (индуктивных переходов), и их доказательстве.

[1] Метод был разработан в конце 1960-х [Flo67], до того, как была предложена методология структурного программирования [Dah72].

5.1. Синтаксис блок-схем

Метод, рассматриваемый в этой лекции, был предложен Робертом Флойдом[1] более полувека назад [Flo67]. Несмотря на почтенный возраст, он до сих пор используется в инструментах дедуктивной верификации, например в модулях Jessie и AstraVer платформы Frama-C (см. лекцию 4). Метод Флойда, логика Хоара, слабейшее предусловие Дейкстры – это разные грани одной парадигмы разработки и верификации программ, которую иногда называют методом Флойда–Хоара–Дейкстры (порядок имен может отличаться).

Метод Флойда оперирует с блок-схемами. Как и при определении языка while, будем считать, что задана сигнатура Σ (множество функциональных и предикатных символов вместе с их арностью), а также множество переменных V.

Для удобства разобьем множество V на три подмножества:

- V_I – *входные* переменные – содержат исходные данные и не меняются во время исполнения программы (блок-схемы);
- V_T – *внутренние* переменные – хранят промежуточные результаты вычислений;
- V_O – *выходные* переменные – инициализируются полученным решением в конце работы программы.

Оператор $(x_1, \ldots, x_n) := (t_1, \ldots, t_n)$, или в «векторной» форме $\vec{x} := \vec{t}$, где $x_1, \ldots, x_n$ – разные переменные, а $t_1, \ldots, t_n$ – термы, интерпретируется следующим образом:

- операционная семантика (см. лекцию 2):

  $$\langle \vec{x} := \vec{t}, s\rangle \to \langle \varepsilon, s[\vec{x} := s(\vec{t})]\rangle;$$

- аксиоматическая семантика (там же):

  $$\{\varphi[\vec{x} := \vec{t}]\}\vec{x} := \vec{t}\{\varphi\}.$$

$s[\vec{x} := s(\vec{t})] \equiv s[x_1 := s(t_1), \ldots, x_n := s(t_n)]$ – состояние, полученное из состояния s сопоставлением переменным $x_1, \ldots, x_n$ значений $s(t_1), \ldots, s(t_n)$ соответственно (т. е. значений термов $t_1, \ldots, t_n$ в состоянии s) без изменения означивания других переменных.

$\varphi[\vec{x} := \vec{t}] \equiv \varphi[x_1 := t_1, \ldots, x_n := t_n]$ – результат подстановки в формулу φ термов $t_1, \ldots, t_n$ вместо всех вхождений переменных $x_1, \ldots, x_n$ соответственно.

Проще говоря, сначала вычисляются значения всех термов, а потом осуществляются присваивания. Можно считать, что значения термов вычисля-

---

[1] Роберт Флойд (Robert W Floyd; 1936–2001) – американский ученый-информатик; известен как автор метода верификации программ и как соавтор алгоритма поиска кратчайшего пути в ориентированном графе (алгоритм Флойда–Уоршелла); лауреат премии Тьюринга (1978); обладатель медали «Пионер компьютерной техники» (1991).

ются параллельно и независимо друг от друга, а затем параллельно присваиваются переменным.

Пусть ξ – некоторое выражение (левая часть присваивания, правая часть присваивания или условие). Чтобы подчеркнуть, что все переменные выражения ξ принадлежат множеству U, будем писать $\xi(U)$.

При построении блок-схем используются операторы следующих типов:

1) *начальный оператор* $\boxed{\textbf{start}: \vec{x}(V_T) := \vec{t}(V_I)}$;
2) *оператор присваивания* $\boxed{\textbf{assign}: \vec{x}(V_T) := \vec{t}(V_I \cup V_T)}$;
3) *условный оператор* $\boxed{\textbf{test}: B(V_I \cup V_T)}$;
4) *оператор соединения* $\boxed{\textbf{join}}$;
5) *завершающий оператор* $\boxed{\textbf{halt}: \vec{x}(V_O) := \vec{t}(V_I \cup V_T)}$.

Обозначим множество всех возможных операторов 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 (пометка ϵ называется *пустой*);
- $L: N \to Op$ – функция разметки узлов операторами.

Для удобства мы иногда будем называть узлы блок-схемы операторами. Например, фраза «в блок-схеме $\langle N, E, L\rangle$ присутствует ровно один начальный оператор» означает, что существует единственный узел $n \in N$, такой что оператор $L(n)$ имеет тип $\boxed{\textbf{start}}$.

Присваивания (включая начальный и конечный операторы) изображаются прямоугольниками, условия – ромбами, соединения – кружками (см. рис. 5.1).

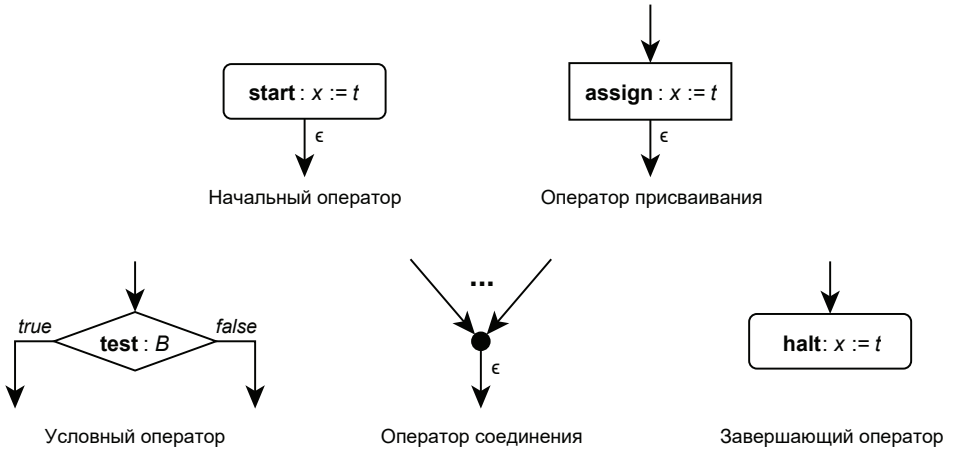


Рис. 5.1 ❖ Графическое представление операторов блок-схем

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

1) в блок-схеме присутствует ровно один начальный оператор и по крайней мере один завершающий;
2) каждый оператор находится на пути от начального оператора к некоторому завершающему;
3) для каждого узла число исходящих дуг и их пометки соответствуют типу оператора:
   - из начального оператора, оператора присваивания и оператора соединения исходит одна дуга, при этом она имеет пометку ϵ;
   - из условного оператора исходят две дуги: одна с пометкой *true*, другая с пометкой *false*;
   - из завершающего оператора не исходит ни одной дуги;
4) для каждого узла число входящих дуг соответствует типу оператора:
   - в начальный оператор не входит ни одной дуги;
   - в оператор присваивания, условный оператор и завершающий оператор входит ровно одна дуга;
   - в оператор соединения входит не менее одной дуги.

Далее мы будем рассматривать только корректно определенные блок-схемы.

Заметим, что для каждого узла n и пометки l в блок-схеме существует не более одной дуги вида (n, l, n'). Если такая дуга существует (оператор $L(n)$ не является завершающим, а пометка l является допустимой), говорят, что узел n' является *последователем* узла n по пометке l. Обозначение: $succ(n, l)$ или $succ(n)$, если $l = \epsilon$.

При изображении блок-схем будем опускать пустые пометки ϵ и ключевые слова, задающие типы операторов: **start**, **assign** и т. д. (тип однозначно определяется по контексту).

Пример 5.1

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

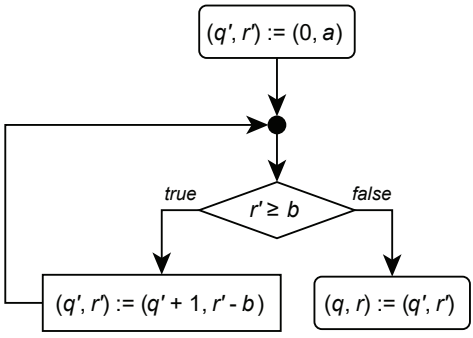


Рис. 5.2 ❖ Блок-схема целочисленного деления

□

5.2. Семантика блок-схем

Как и прежде, *состоянием* программы (блок-схемы) называется отображение, которое каждой переменной ставит в соответствие ее значение. 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 псевдоузел n_ω, соответствующий завершению работы) и состояние. Определим отношение переходов на множестве конфигураций, как показано в табл. 5.1.

Таблица 5.1. Отношение переходов на множестве конфигураций блок-схемы

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

Для блок-схем, как и для while-программ, можно определить понятие вычисления. Приводимые ниже определения практически дословно повторяют свои аналоги из лекции 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 $i \geq 0$ ($i < n - 1$, если последовательность конечна и n – ее длина).

Вычислением блок-схемы P в состоянии s называется последовательность переходов, которая не может быть продолжена.

Говорят, что вычисление *завершается* в состоянии s', если оно конечно и его последняя конфигурация имеет вид $\langle n_\omega, s'\rangle$ (обозначение: $\langle n_0, s\rangle \xrightarrow{*} \langle n_\omega, s'\rangle$); блок-схема P *может зациклиться* в состоянии s, если существует бесконечное вычисление P в s.

Семантика блок-схем определяется следующим образом:

$$M[\![P]\!](s) = \left\{ s' \in S \mid (n_0, s) \xrightarrow{*} (n_\omega, s') \right\} \cup \begin{cases} \{\omega\}, \text{ если } P \text{ может зациклиться в } s \\ \varnothing \text{ в противном случае} \end{cases}.$$

Для каждой блок-схемы P и состояния s множество $M[\![P]\!](s)$ состоит из одного элемента (блок-схемы детерминированы), 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.

Определение остается тем же: блок-схема P называется *частично корректной* относительно предусловия φ и постусловия ψ (обозначение: $\models \{\varphi\}P\{\psi\}$), если для любого состояния s, такого что $s \models \varphi$ и $M[\![P]\!](s) \neq \omega$, имеет место $M[\![P]\!](s) \models \psi$.

Пусть P – произвольная корректно определенная 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 на **halt** – это может быть произвольная последовательность смежных дуг (возможно, с повторами).

Построим формулу φ_π и подстановку θ_π:

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

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

5.3.1. Метод обратных подстановок

Договоримся, что последний оператор $L(n_k)$ не является условным оператором (неправильно отделять условие от дуги с пометкой).

Для каждого $0 \le m \le k + 1$ определим $\varphi_\pi^{(m)}$ и $\theta_\pi^{(m)}$ – соответственно формулу и подстановку для пути $N_\pi^{(m)} = \{n_i\}_{i=m}^{k}$ – концовки (суффикса) исходного пути.

Заметим: $\varphi_\pi \equiv \varphi_\pi^{(0)}$ 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 табл. 5.2.

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

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

Здесь $\theta' = \theta \cdot [\vec{x} := \vec{t}]$ есть *суперпозиция* подстановок θ и $[\vec{x} := \vec{t}]$. С практической точки зрения, θ' – это подстановка, эффект от применения которой эквивалентен последовательному применению сначала θ, потом $[\vec{x} := \vec{t}]$. Более строго:

$$[\vec{y} := \vec{r}] \cdot [\vec{x} := \vec{t}] = [\vec{y} := \vec{r}[\vec{x} := \vec{t}], \vec{x}_н := \vec{t}_н],$$

где $\vec{x}_н$ – кортеж переменных, входящих в $\vec{x}$, но не входящих в $\vec{y}$, а $\vec{t}_н$ – кортеж соответствующих термов из подстановки $[\vec{x} := \vec{t}]$.

Обратите внимание: путь просматривается от конца к началу, поэтому подстановки применяются в порядке, обратном порядку исполнения операторов (этим обусловливается название метода). Идея здесь та же, что при построении слабейшего предусловия (см. лекцию 2). Действительно, $\varphi_\pi = wp(op(\pi), true)$ – слабейшее условие, гарантирующее проход по пути π (здесь $op(\pi)$ – последовательность операторов, расположенных вдоль пути π).

5.3.2. Точки сечения, индуктивные утверждения и условия верификации

Чтобы было удобнее описывать метод Флойда, совершим следующие «преобразования» блок-схемы:

- расширим множество узлов двумя псевдоузлами n_α и n_ω;
- добавим псевдодугу $(n_\alpha, \epsilon, n_s)$, где n_s – начальный узел:
  - будем называть ее *начальной*;
- для каждого завершающего узла n_h добавим псевдодугу $(n_h, \epsilon, n_\omega)$:
  - такие псевдодуги будем называть *завершающими*.

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

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

Элементы множества C называются *точками сечения* (*cut points*). Выделяют *начальную*, *завершающие* и *внутренние* точки сечения. Пути между точками сечения[1], не содержащие других точек сечения, называются *базовыми путями*. Базовые пути, соединяющие внутренние точки сечения, называются *внутренними*.

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

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

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

Это условие выражает простую мысль: если в точке i истинно утверждение χ_i и исполнение пойдет по пути π (что обеспечивается условием φ_π), то в точке j (когда мы туда попадем) будет истинно утверждение χ_j (для этого сейчас должно быть истинно $\chi_j \theta_\pi$).

Почему утверждения, сопоставленные точкам сечения, называются *индуктивными*? Название происходит от метода *математической индукции*, согласно которому для доказательства некоторого утверждения о натуральных числах в общем виде, например $\forall n \in \mathbb{N}_0(\varphi(n))$, достаточно

- проверить истинность *базиса индукции*: $\varphi(0)$;
- доказать *индуктивный переход*: $\forall n \in \mathbb{N}_0(\varphi(n) \to \varphi(n+1))$.

В методе Флойда доказываемое утверждение имеет вид «каким бы ни было вычисление, если начальное состояние удовлетворяет предусловию и истинны условия верификации, то всегда при прохождении вычисления через точку сечения будет истинно соответствующее индуктивное утверждение (в частности, при прохождении вычисления через завершающую точку сечения будет истинно постусловие)».

Индукция осуществляется по числу вхождений точек сечения в префикс вычисления. Базис индукции – истинность предусловия при прохождении через начальную точку сечения. Индуктивный переход (доказательство индуктивного утверждения для очередной точки сечения по индуктивному утверждению для предыдущей) обеспечивается условиями верификации для базовых путей.

---

[1] Когда мы говорим, что путь начинается (оканчивается) в точке сечения (n, l, n'), мы имеем в виду, что он начинается (оканчивается) с узла n' (в узле n).

Приведенные рассуждения не вполне формальны. Читателю предлагается самостоятельно доказать корректность метода Флойда.

5.3.3. Общая схема метода индуктивных утверждений

Подведем итог. Пусть даны блок-схема P, предусловие φ и постусловие ψ. Метод индуктивных утверждений Флойда состоит в следующем:

1) выбираются точки сечения;
2) для каждой внутренней точки сечения формулируется индуктивное утверждение;
3) строятся условия верификации для всех базовых путей;
4) условия верификации доказываются.

Если удается доказать все построенные условия верификации, то блок-схема частично корректна относительно заданной спецификации: $\vDash \{\varphi\}P\{\psi\}$.

Все шаги метода Флойда, за исключением шага 2 и, строго говоря, шага 4, могут быть выполнены автоматически. Выбор подходящих индуктивных утверждений требует *понимания* программы и с трудом поддается автоматизации (см. лекцию 7). Шаг 4 может быть реализован с помощью SMT-решателей (см. лекцию 8), однако в общем случае требует человеческого вмешательства.

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 псевдодуги (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;
2) BB – путь между точками B и B;
3) BC – путь между точками B и C.

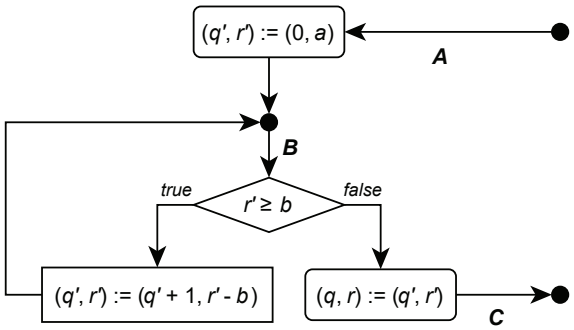


Рис. 5.3 ❖ Точка сечения блок-схемы целочисленного деления

Рассмотрим путь AB:

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

Имеем:

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

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

Таким образом, $\vDash (\varphi \wedge \varphi_{AB}) \to \chi\theta_{AB}$.

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

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

Имеем:

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

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

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

$\vDash \varphi \to (b \ge 0)$, следовательно, $\vDash ((b \le r' \le a) \wedge \varphi) \to ((b \le r' \le (a + b)) \wedge \varphi)$.

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

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

Имеем

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

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

Истинность импликации $(\chi \wedge \varphi_{BC}) \rightarrow \psi\theta_{BC}$ очевидна.

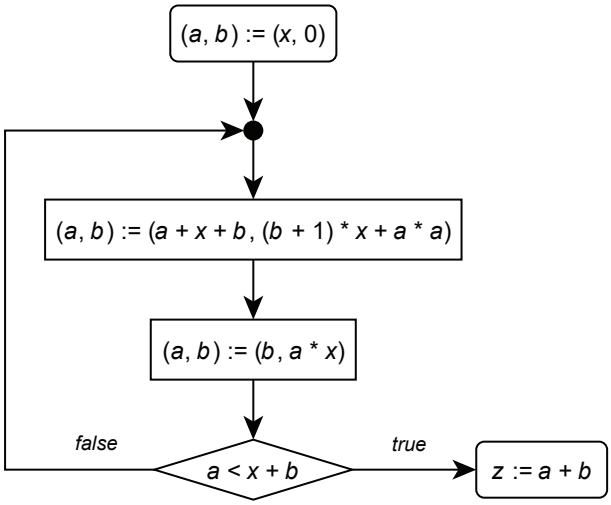
Таким образом, приведенная блок-схема частично корректна относительно φ и ψ.

□

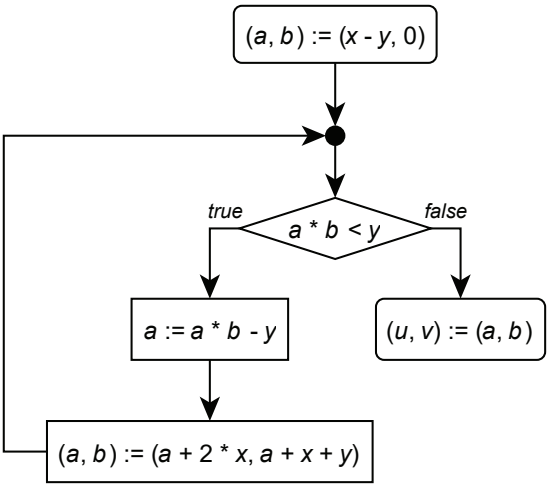
5.4. Вопросы и упражнения

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?
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 возведение 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.

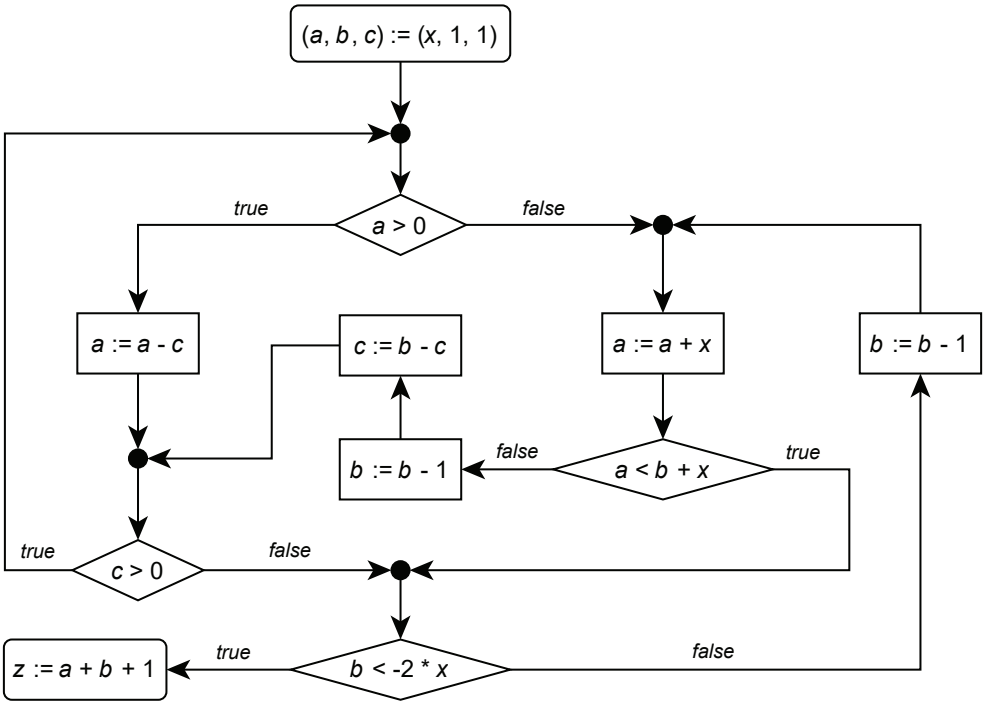
7. Докажите частичную корректность приведенной ниже блок-схемы относительно предусловия $\varphi \equiv (x \geq 0)$ и постусловия $\psi \equiv (z \geq 0)$ [Буз14].



8. Докажите частичную корректность приведенной ниже блок-схемы относительно предусловия $\varphi \equiv (x > y)$ и постусловия $\psi \equiv ((u - x) = (v - y))$ [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, гарантирующее проход по этому пути.
11. Докажите значимость метода индуктивных утверждений Флойда: если для программы и ее спецификации метод выдает положительный вердикт, то данная программа является частично корректной относительно данной спецификации.

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 подхода заключается в следующем: каждой точке сечения сопоставляется оценочная функция; значение функции уменьшается при каждом прохождении через эту точку (при условии что истинно индуктивное утверждение), но в то же время не может уменьшаться бесконечно долго в силу характера изменений и ограничений на область значений. Если удается найти такие функции и доказать условия верификации, программа завершается и, более того, является полностью корректной.

6.1. Завершимость программ и полная корректность

Одной из составляющих понятия корректности вычислительной программы является *завершимость*, т. е. гарантированный останов через конечное число шагов при запуске в любом допустимом начальном состоянии. Основной причиной, по которой программа может не завершиться, является *зацикливание* – вхождение вычислений в *бесконечный цикл* (условие выхода из кото-

рого никогда не становится истинным). Анализ завершимости программы представляет весьма сложную задачу: достаточно вспомнить, что проблема останова алгоритмически неразрешима [Tur36].

Сразу отметим, что существует широкий класс программ, для которых завершение работы является редким событием, сигнализирующим, 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, телекоммуникационные сети, системы управления. Методам формальной спецификации и верификации реагирующих систем посвящена вторая часть книги, начинающаяся с лекции 9.

Вспомним определение: программа (блок-схема) P называется *полностью корректной* относительно предусловия φ и постусловия ψ (обозначение: $\vDash \langle\varphi\rangle P\langle\psi\rangle$), 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 $\{\varphi\}P\{\psi\}$ и $\langle\varphi\rangle P\langle true\rangle$, то $\langle\varphi\rangle P\langle\psi\rangle$.

Иными словами, для доказательства полной корректности программы достаточно доказать ее частичную корректность и завершимость (см. табл. 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 следует ее полная корректность

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

6.2. Метод фундированных множеств

Для изложения первого метода нам потребуются вспомогательные понятия.

(*Нестрогим*) *частичным порядком* на множестве M называется бинарное отношение $\preccurlyeq$, 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 $a, b, c \in M$ 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.

Бинарное отношение $\prec$ на множестве M называется *строгим частичным порядком*, если оно удовлетворяет следующим свойствам:

- *антирефлексивность* (*иррефлексивность*): для всех $a \in M$ справедливо $a \not\prec a$[1];
- *асимметричность*: для всех $a, b \in M$ из того, что $a \prec b$, следует, что $b \not\prec a$;
- *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.

В этом разделе нам будет удобнее работать со строгими порядками.

Частично упорядоченное множество $\langle M, \prec \rangle$ называется *фундированным*, если в нем отсутствуют бесконечно убывающие последовательности элементов: $a_0 \succ a_1 \succ \cdots$[2] (*условие обрыва убывающих цепей*).

Другими словами, для любого непустого подмножества $W \subseteq M$ частично упорядоченное множество $\langle W, \prec_W \rangle$, где $\prec_W \equiv \prec \cup (W \times W)$, имеет *минимальный элемент* [Ерш87]:

> элемент $a \in W$, для которого не существует элемента $b \in W$,
> такого что $b \prec a$ (либо $a \prec b$, либо $a = b$, либо a и b несравнимы)[3].

Минимальный элемент не следует путать с *наименьшим*. Последний определяется так:

> элемент $c \in W$, такой что для всех $b \in W$ имеет место $c = b$ или $c \prec b$.

Пример 6.1

Рассмотрим примеры фундированных множеств.

- $\langle \mathbb{N}_0, < \rangle$ – расширенное множество натуральных чисел $\mathbb{N}_0 = \{0, 1, \ldots\}$ с отношением «строго меньше»;
- $\langle \mathbb{Z}, \prec \rangle$ – множество целых чисел $\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$ со следующим отношением: $a \prec b$ тогда и только тогда, когда $a \neq b$ и $b \vdots a$ (b делится на a без остатка);
- $\langle \Sigma^*, \subset \rangle$ – множество конечных слов в алфавите Σ со следующим отношением: $a \subset b$ тогда и только тогда, когда слово a является строгим подсловом слова b.

□

---

[1] Запись $a \not\prec b$ означает то же, что и $\neg(a \prec b)$.

[2] Запись $a \succ b$ эквивалентна $b \prec a$.

[3] Строго говоря, эти определения не эквивалентны: эквивалентность имеет место, если принять *аксиому выбора*.

6.2.1. Оценочные функции и условия завершимости

Пусть мы верифицируем некоторую блок-схему и выбрали точки сечения (см. лекцию 5). Для каждой точки сечения i сформулируем индуктивное утверждение χ_i, а для каждого базового пути π (пусть он начинается в точке сечения i и оканчивается в точке сечения j) запишем условие верификации:

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

Выберем фундированное множество $\langle M, \prec \rangle$ и для каждой точки сечения i зададим в виде терма u_i *оценочную функцию*, которая сопоставляет каждому состоянию, удовлетворяющему условию χ_i, элемент множества M.

Для каждого внутреннего базового пути π (считаем, что он соединяет точки сечения i и j) построим *условие завершимости* (*TC*, *Termination Condition*):

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

Идея проста: если в точке i истинно утверждение χ_i и вычисление пойдет по пути π, то в точке j (когда мы в ней окажемся) значение оценочной функции u_j будет «меньше» значения оценочной функции u_i в точке i (в текущем 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 своем полном варианте метод индуктивных утверждений и метод фундированных множеств, состоит из следующих шагов:

1) выбираются точки сечения;
2) для каждой внутренней точки сечения задаются два выражения: индуктивное утверждение и оценочная функция;
3) строятся условия верификации для всех базовых путей и условия завершимости для всех внутренних базовых путей;
4) условия верификации и условия завершимости доказываются.

Если удается доказать все построенные условия верификации и условия 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): ей сопоставлено индуктивное утверждение

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

Задачи, которые нужно решить:

1) предложить оценочную функцию для точки B;
2) написать 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 $u \equiv (a - q')$. Очевидно, что она определена во всех состояниях, удовлетворяющих χ. Докажем, что множество значений u вложено в $\mathbb{N}_0$:

- $(a = q' \cdot b + r') \wedge (0 \leq r' \leq a) \wedge (a \geq 0) \wedge (b > 0) \equiv$
- $\left(q' = \dfrac{a - r'}{b}\right) \wedge (0 \leq r' \leq a) \wedge (a \geq 0) \wedge (b > 0) \rightarrow$
- $\left(q' \leq \dfrac{a}{b}\right) \wedge (a \geq 0) \wedge (b > 0) \rightarrow$
- $(q' \leq a) \equiv$
- $(a - q') \geq 0 \equiv$
- $u \in \mathbb{N}_0$.

Доказательства условий верификации для усиленных вариантов предусловия φ и индуктивного утверждения χ практически не отличаются от приведенных в лекции 5.

Докажем условие завершимости

$$(\chi \wedge \varphi_{BB}) \rightarrow (u\theta_{BB} < u),$$

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

- $(\chi \wedge \varphi_{BB}) \rightarrow ((a - q')[q' := q' + 1, r' := r' - b] < (a - q')) \equiv$
- $(\chi \wedge \varphi_{BB}) \rightarrow (((a - (q' + 1)) < (a - q')) \equiv$
- $(\chi \wedge \varphi_{BB}) \rightarrow (0 < 1) \equiv$
- $(\chi \wedge \varphi_{BB}) \rightarrow true \equiv$
- $true$.

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

□

6.3. Метод счетчиков

Еще один способ доказательства завершимости и 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 инкрементируется при исполнении тела цикла. В инвариант цикла встраивается условие, ограничивающее значение счетчика сверху. Важно, чтобы верхняя граница (так называемая *ограничивающая функция*) зависела только от входных переменных. Для доказательства полной корректности исходной программы достаточно доказать частичную корректность «инструментированной» программы со счетчиками.

Пример 6.3

В программу целочисленного деления 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, включающий ограничение на значение q. Рассмотрим следующего претендента:

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

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

$$(a = q \cdot b + r) \wedge (0 \le r \le a) \wedge (a \ge 0) \wedge (b > 0),$$

также является инвариантом. В предыдущем примере мы показали, что $q \le a$ является следствием $(a = q \cdot b + r) \wedge (0 \le r \le a) \wedge (a \ge 0) \wedge (b > 0)$. Таким образом, χ – инвариант, откуда (с учетом того, что χ содержит ограничение на значение 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 $a \ge 0$. Для этого введем вспомогательную переменную-счетчик i (см. табл. 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. Несложная модификация, показанная в табл. 6.3 (строка 5), делает ее пригодной для доказательства полной корректности [Apt09].

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

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

Изменяется только правило вывода для оператора цикла: появляется условие «строгого уменьшения» значения оценочной функции при исполнении тела цикла, а также условие принадлежности значения оценочной функции фундированному множеству при истинности индуктивного утверждения.

Пример 6.5

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

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

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

Вычислим слабейшее предусловие (см. лекцию 3):

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

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

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

Следовательно, программа завершается.

□

6.5. Аннотирование программ для доказательства завершимости

В лекции 3 нами было введено понятие *аннотированной программы*. Так называется программа, в которую добавлены специальные утверждения (*аннотации*), описывающие свойства ее фрагментов. До сих пор мы имели дело со следующими типами аннотаций:

- пред- и постусловия;
- ограничения на состояние программы в определенных точках;
- инварианты циклов.

Для доказательства завершимости и полной корректности программ применяются аннотации, позволяющие задавать оценочные функции:

{**bd**: *u*[, ⟨*M*, ≺⟩]}.

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

Пример 6.6

Ниже приведена программа целочисленного деления, аннотированная инвариантом и оценочной функцией.

```
⟨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 преобразуйте эту программу к блок-схеме и запишите условия верификации и условия завершимости.

□

Для записи аннотаций типа **bd** на языке ACSL (см. лекцию 4) используется клауза **loop variant** [Bau10]:

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

Здесь u – оценочная функция, а R – необязательный предикат, задающий фундированное множество (если параметр не указан, используется $\langle \mathbb{N}_0, < \rangle$).

Пример 6.7

Приведенная ниже ACSL-спецификация аналогична аннотациям 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 задать специфический частичный порядок, можно использовать конструкцию **predicate** (см. лекцию 4):

```
//@ 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;
  @*/
```

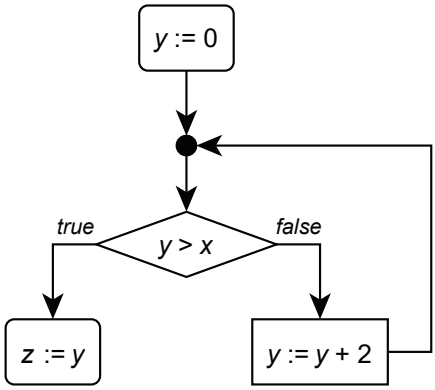
□

6.6. Вопросы и упражнения

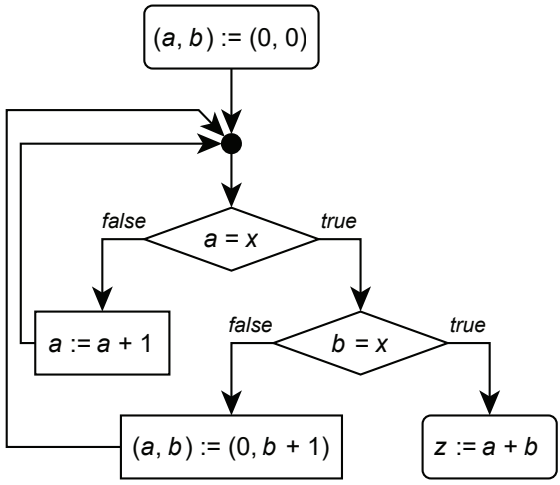
1. Каким свойствам удовлетворяет отношение строгого частичного порядка?
2. Дайте определение фундированного множества. Приведите примеры.
3. Являются ли фундированными следующие частично упорядоченные множества:
   а) $\langle \mathbb{Q}_+, < \rangle$, где $\mathbb{Q}_+$ – множество положительных рациональных 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) $\langle \Sigma^*, \prec \rangle$, где Σ – конечный алфавит, а $\prec$ – отношение лексикографического порядка?

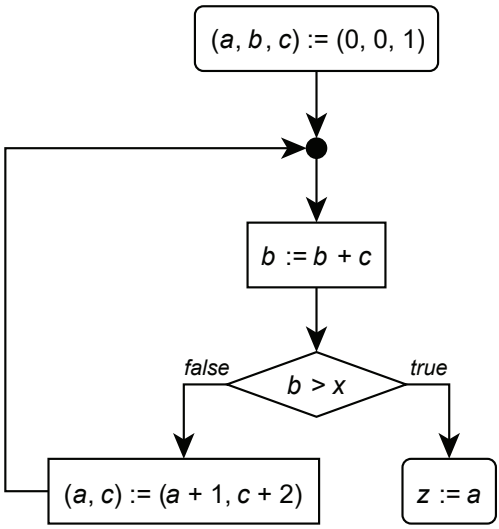
4. Верно ли, что метод фундированных множеств является более общим по сравнению с методом счетчиков, т. е. всегда, когда удается построить ограничивающую функцию для счетчика цикла, для этого цикла можно построить и оценочную функцию?
5. Докажите завершимость приведенной ниже блок-схемы для любых значений 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. Аннотируйте на языке ACSL функцию дихотомического поиска в упорядоченном числовом массиве (см. лекцию 4). Докажите полную корректность этой функции.
10. Докажите корректность (значимость) метода Флойда: если для блок-схемы и ее спецификации метод выдает положительный вердикт, то эта блок-схема является полностью корректной относительно данной спецификации.

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 может быть сведена к построению инвариантов программ, расширенных счетчиками. Описываются два класса методов: эвристические методы и методы, основанные на абстрактной интерпретации, т. е. символьном 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. Представим себе, что наша цель – разработать инструмент, который берет на вход программу (код на языке программирования, например C) и формальную спецификацию (программный контракт, например

в форме ACSL-аннотаций), анализирует их (по возможности автоматически) и выдает вердикт:

- *положительный*, если программа корректна (соответствует спецификациям);
- *отрицательный*, если доказать корректность не удалось[1].

С учетом изложенного ранее такой инструмент мог бы работать следующим образом:

1) анализ исходного кода (лексический, синтаксический, семантический) и конструирование *графа потока управления* программы;
2) построение множества *точек сечения* графа потока управления (множества дуг, обладающего тем свойством, что в каждом цикле графа имеется по крайней мере одна дуга из этого множества); перечисление *базовых путей* (путей, соединяющих точки сечения и не содержащих точки сечения внутри себя) (см. лекцию 5);
3) конструирование *формул* и *подстановок* для базовых путей (см. лекцию 5):
   a) формула пути определяет слабейшее условие, гарантирующее проход по этому пути;
   b) подстановка задает преобразование состояния программы, совершаемое при исполнении операторов, расположенных вдоль пути;
4) синтез *индуктивных утверждений* и *оценочных функций* для внутренних точек сечения (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 (или опровержение) построенных условий верификации и завершимости (см. лекцию 8); выдача *вердикта*.

Возможность автоматизации шага 1 не вызывает сомнений – теория трансляции и компиляции программ хорошо исследована [Сер12]. Шаг 2 в простейшем случае сводится к обходу графа потока управления с запоминанием его вершин: если очередная дуга ведет в уже пройденную вершину, она добавляется в множество внутренних точек сечения (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.

---

[1] Такая постановка задачи, очевидно, допускает тривиальное решение, когда инструмент всегда возвращает отрицательный вердикт. Требуя, чтобы для любой корректной программы инструмент возвращал положительный вердикт, а для любой некорректной – отрицательный, мы уйдем в дебри алгоритмически неразрешимых задач. Важно найти баланс.

Известно, что в общем случае шаг 4 не поддается автоматизации[1]. Тем не менее для ряда предметных областей существуют подходы, позволяющие строить индуктивные утверждения (их рассмотрению посвящена оставшаяся часть лекции). Мы не будем отдельно рассматривать синтез оценочных функций – эта задача может быть сведена к построению инвариантов циклов в программах, расширенных счетчиками [Неп88].

Напомним: *инвариантом цикла* **while** B **do** P **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:

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

На основе какой информации можно построить инвариант? Учтите, что для инварианта χ должна быть истинна импликация $(\chi \wedge \neg B) \rightarrow \psi$. Налицо четыре возможности (и 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* (в основном используют φ и ψ) [Gri81];
- методы на основе *абстрактной интерпретации* (используют B и P) [Cou78].

7.2. Эвристические методы синтеза инвариантов циклов

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 отправную точку – область определения. В соответствии с этим основное внимание мы уделим постусловию.

7.2.1. Ослабление постусловия

Итак, нам известно постусловие ψ:

while B **do** P **end** $\{\psi\}$

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

В качестве кандидата на роль инварианта рассмотрим следующую формулу [Неп88]:

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

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 (предполагаем, что программа корректна). Однако пользы от такого инварианта мало – он слишком слаб (в логическом смысле).

Другим кандидатом на роль инварианта является само постусловие [Gri81]:

$$\chi_s \equiv \psi.$$

Вообще говоря, χ_s не инвариант: ψ истинно после выхода из цикла, но для остальных итераций это не так (иначе непонятно, зачем совершать дополнительные итерации, если цель вычислений уже достигнута – постусловие истинно). Дэвид Грис[1] использует метафору воздушного шарика [Gri81]. Постусловие – это воздушный шарик, из которого выпущен воздух, а инвариант – надутый шарик[2]; 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».

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 сильно помогает в синтезе инварианта – «отрезок» $[\chi_w, \chi_s]$ содержит бесконечное множество формул (если, конечно, B не *false*).

Будем строить инвариант путем ослабления постусловия. Для этого имеются следующие эвристики (для усиления условий можно использовать схожие соображения) [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 общего вида*): название этой эвристики говорит само за себя.

---

[1] Дэвид Грис (David Gries; род. в 1939 году) – американский ученый и педагог в области информатики; автор книг «Конструирование компиляторов для цифровых вычислительных машин» (1971), «Наука программирования» (1981) и др.; лауреат Образовательной премии Американской федерации обработки информации (AFIPS, American Federation of Information Processing Societies) (1986).

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

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
end
{a = q·b + r ∧ 0 ≤ r < b}
```

В качестве аппроксимации инварианта рассмотрим постусловие

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

Очевидно, это не инвариант. Применяя эвристику 1, устраним конъюнктивный член $(0 \le r < b)$. Получим $(a = q \cdot b + r)$; это, как легко убедиться, инвариант, но слишком слабый. Вернемся к постусловию и применим к нему эвристику 2: расширим область значений переменной r с полуинтервала $[0, b)$ до полупрямой $[0, \infty)$. Получим адекватный инвариант $(a = q \cdot b + r) \wedge (r \ge 0)$.

□

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)}
```

Начнем строить инвариант цикла с постусловия, т. е. $s = sum(x, N)$ (определение функции *sum* дано в лекции 3). Замена входной переменной N на внутреннюю переменную n сразу дает решение $s = sum(x, n)$.

□

7.2.2. Комбинирование пред- и постусловий

В некоторых случаях при построении инварианта цикла нужно учитывать как постусловие, так и предусловие [Gri81]. Основная идея состоит в построе-

нии формулы, которая «объединяла» бы в себе оба условия. Большой науки здесь нет – ограничимся примером.

Пример 7.3

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

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

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

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

являющееся инвариантом. Условие это, хотя и позволяет доказать частичную корректность программы, не является *полным*: в нем нет ограничений на элементы $x[n], \ldots, x[N-1]$. Усилив его ограничением, взятым из предусловия, получим

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

С одной стороны, это кажется лишним (для верификации); с другой – дает лучшее понимание того, что происходит в цикле, что важно, например, при синтезе программы по спецификациям [Gri81].

□

7.3. Методы синтеза инвариантов на основе абстрактной интерпретации

Если эвристические методы синтеза инвариантов циклов базировались на *спецификации*, то в методах следующего типа используется *реализация*. Методы, о которых идет речь, основаны на *абстрактной интерпретации*.

Ключевые концепции этого подхода были сформулированы Патриком Кюзо[1] [Cou77].

Под *абстрактной интерпретацией* понимается *символьное* исполнение программы на *упрощенной модели* (*abstract domain*). Абстрактная интерпретация осуществляется пошагово: результатом каждого шага является не конкретное состояние, а символьное представление множества возможных состояний. Важно отметить: такое представление является *приближенным* (что отличает абстрактную интерпретацию от традиционного символьного исполнения), но обязательно *аппроксимацией сверху* (*over-approximation*) – представление включает все возможные состояния и, может быть, некоторые другие.

Упрощенные модели, т. е. способы приближенного представления множеств состояний, могут быть разными и определяются предметной областью. Ниже мы коснемся двух моделей: интервальной арифметики и теории выпуклых многогранников. Важная для верификации разновидность абстрактной интерпретации – предикатная абстракция – рассматривается в лекции 11.

7.3.1. Интервальная арифметика

Интервальная арифметика [Шок81] – известная упрощенная модель, используемая для абстрактной интерпретации вычислительных программ. Множество возможных значений числовой переменной аппроксимируется интервалом[2], а множество возможных состояний, так называемое *абстрактное состояние*, – отображением переменных в интервалы.

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.

---

[1] Патрик Кюзо (Patrick Cousot; род. в 1948 году) – французский ученый в области информатики; совместно со своей женой Радхией Кюзо (Radhia Cousot; 1947–2014) заложил основы абстрактной интерпретации; обладатель серебряной медали Французского национального центра научных исследований (CNRS, Centre National de la Recherche Scientifique) (1999).

[2] Под *интервалом* здесь понимается отрезок, неограниченный полуотрезок или вся числовая прямая. В англоязычной литературе *интервалом* называют произвольный (не обязательно открытый) промежуток.

Пример 7.4

Рассмотрим простую программу и проинтерпретируем ее, используя целочисленную интервальную арифметику. Программа и шаги ее интерпретации (вместе с необходимыми комментариями) показаны в табл. 7.1.

Таблица 7.1. Абстрактная интерпретация в целочисленной интервальной арифметике

Программа	Аппроксимация множества возможных состояний
{0 ≤ x ≤ 2 ∧ 0 ≤ y ≤ 3}	$\{x \mapsto [0, 2], y \mapsto [0, 3]\}$ **Начальное абстрактное состояние**
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**: $x \leq \max(y) = 3 \Rightarrow x \in [0, 2]$
// ветвь else	(уточнения нет)
	$y \geq \min(x) = 0 \Rightarrow y \in [0, 3]$
	(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}	$\{x \mapsto [-2, 4], y \mapsto [0, 4]\}$ **Конечное абстрактное состояние**

Заметим: 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.

□

7.3.2. Общие сведения об абстрактной интерпретации

В лекции 2 мы говорили, что для языка программирования задана *формальная семантика*, если определена функция M, ставящая в соответствие каждой программе P на этом языке 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 возможных начальных состояний во множество возможных конечных состояний:

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

Здесь S – множество состояний программы P.

Обратите внимание: пара $\langle 2^S, \subseteq \rangle$ является частично упорядоченным множеством (см. лекцию 6), а функция $M[\![P]\!]$ *монотонна*, т. е.

$M[\![P]\!](X) \subseteq M[\![P]\!](Y)$ для всех $X \subseteq Y$.

Для заданной упрощенной модели AD и заданной программы P функция переходов «трансформируется» в *обобщенную функцию переходов*[1]:

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

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

Для краткости будем опускать индекс AD у обобщенных функций переходов.

Пусть $\langle M, \preccurlyeq \rangle$ – частично упорядоченное множество. Дадим несколько вспомогательных определений [Bir67].

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 (обозначение: $\sup W$) – это такая верхняя грань a, что $a \preccurlyeq a'$ для любой другой верхней грани a'[2].

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 $\sup W$ – наименьший элемент M (предполагаем, что $M \neq \emptyset$), т. е. полнота предполагает наличие наименьшего элемента.

Имеет место следующий факт (*теорема Тарского*[3] *о неподвижной точке*) [Bir67]: если $\langle M, \preccurlyeq \rangle$ – полное частично упорядоченное множество и $f: M \to M$ – монотонная функция, то f имеет *неподвижную точку*, т. е. существует элемент $a \in M$, такой что $f(a) = a$[4].

Заметим, что $\langle 2^S, \subseteq \rangle$ – полное частично упорядоченное множество. В самом деле, для любого $W \subseteq 2^S$ (множества подмножеств S)

$\sup W = \bigcup_{X \in W} X$ (полагается: $\sup \emptyset = \emptyset$).

---

[1] В общем случае речь идет о преобразованиях одного абстрактного состояния (элемента некоторого частично упорядоченного множества) в другое (элемент того же самого множества). Вид абстрактных состояний может быть самым разным – важно, чтобы преобразование было монотонным.

[2] Из антисимметричности частичного порядка (см. лекцию 6) следует, что точная верхняя грань если существует, то единственна.

[3] Альфред Тарский (Alfred Tarski; 1901–1983) – выдающийся польско-американский логик; основатель формальной теории истинности; автор ряда результатов о разрешимости и неразрешимости формальных теорий в логике первого порядка; член-корреспондент Британской академии (1966).

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

Таким образом, из теоремы Тарского следует, что (обобщенная) функция переходов $M[\![P]\!]$ имеет неподвижную точку. Это же справедливо в отношении *функции достижимости*

$$M^*[\![P]\!](X) \equiv M[\![P]\!](X) \cup X,$$

представляющей как множество начальных состояний, так и множество состояний, достижимых из начальных при исполнении программы P. Многократное применение функции достижимости $M^*[\![P]\!]$,

$$M^*[\![P]\!](M^*[\![P]\!](\ldots(X))),$$

дает множество всех состояний (начальных, промежуточных, конечных), возможных при последовательном исполнении P соответствующее число раз.

Как вся эта математика связана с поиском инварианта цикла? Вернемся к нашей задаче: требуется найти инвариант цикла **while** B **do** P **end**. Пусть зафиксирована некоторая упрощенная модель и $M[\![P]\!]$ – обобщенная функция перехода P (тела цикла). В качестве кандидата на роль инварианта возьмем *символьное представление неподвижной точки* функции достижимости $M^*[\![P]\!]$. Заметим, что X – неподвижная точка $M^*[\![P]\!]$ тогда и только тогда, когда $M[\![P]\!](X) \subseteq X$, т. е. свойство принадлежности состояния множеству X *инвариантно* относительно преобразования $M[\![P]\!]$.

Если χ – символьное представление неподвижной точки $X \subseteq S$, то

$$\{\chi\}P\{\chi\} \quad (\text{в частности, } \{\chi \wedge B\}P\{\chi\}) \quad \text{и} \quad \{\chi\}\ \textbf{while}\ B\ \textbf{do}\ P\ \textbf{end}\ \{\chi \wedge \neg B\}\,.$$

Подходящая неподвижная точка может быть выражена как предел последовательности[1]

$$X^{(i+1)} = M^*[\![P]\!](X^{(i)}),$$

где $X^{(0)}$ – это аппроксимация сверху множества состояний, в которых истинно предусловие цикла. Чтобы итерационный процесс *стабилизировался* за конечное число шагов, применяют так называемые *операторы расширения* (*widening operator*).

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 $(X \cup Y) \subseteq (X \nabla Y)$;
- для любой последовательности $\{Y^{(i)}\}_{i=0}^{\infty}$ и любого множества $X^{(0)}$ последовательность $X^{(i+1)} = X^{(i)} \nabla Y^{(i)}$ когда-нибудь стабилизируется [Mon09].

Расширение можно представить себе так: множества состояний X и Y описываются системами ограничений; система ограничений $X \nabla Y$ получается из

[1] Строго говоря, для этого нужно потребовать *непрерывность* функции $M^*[\![P]\!]$: для всех последовательностей $X_0 \subseteq X_1 \subseteq \cdots$ должно быть выполнено равенство $M^*[\![P]\!](\bigcup_{i=0}^{\infty} X_i) = \bigcup_{i=0}^{\infty} M^*[\![P]\!](X_i)$.

системы ограничений X путем выкидывания ограничений, противоречащих ограничениям Y.

Инвариант цикла строится с помощью итерационного процесса

$$X^{(i+1)} = X^{(i)} \nabla M[\![P]\!](X^{(i)}),$$

критерием завершения которого является условие $M[\![P]\!](X^{(i)}) \subseteq X^{(i)}$. Оператор расширения начинает применяться только начиная с некоторого шага, когда накоплено достаточно информации о структуре множества достижимых состояний (см. следующий пример).

7.3.3. Теория выпуклых многогранников

Разъясним основные концепции абстрактной интерпретации на примере *теории выпуклых многогранников* (*convex polyhedra*) [Cou78]. Пусть программа работает с числовыми переменными и совершает над ними линейные преобразования. В этом случае множество возможных состояний программы удобно аппроксимировать выпуклым многогранником (системой линейных неравенств). Операторы программы интерпретируются как трансформации многогранника. Так, присваивание $x := x + c$, где c – некоторая константа, приводит к сдвигу многогранника вдоль «оси» x на величину c. Для присваивания общего вида $x := t$ есть две возможности:

1) в терм t (линейное выражение) входит переменная x: старое значение x выражается через новое и подставляется в систему ограничений, описывающую многогранник; например, для $x := x + c$ подстановка имеет вид $x := x - c$ (слева – старое значение, справа – новое);
2) в противном случае (старое значение нельзя выразить через новое): переменная x исключается, например с помощью метода Фурье–Моцкина (Fourier–Motzkin elimination) [Yua06];
   а) если при этом t – линейное выражение, в систему ограничений добавляется равенство $x = t$.

Каждый раз, когда соединяются разные пути исполнения программы (оператор **join**), вычисляется *выпуклая оболочка* совокупности многогранников, построенных для разных ветвей программы. Таким образом, на выходе получается многогранник, охватывающий все многогранники, полученные на входе:

$CP = Conv(\bigcup_{j=1}^{n} CP_j)$, где $Conv(X)$ – выпуклая оболочка X, n – число путей.

Циклы обрабатываются особым образом. Занумеруем дуги, ведущие в начало цикла, от 1 до n так, чтобы номер n имела дуга обратной связи (переход на очередную итерацию). Как и прежде, вычисляется выпуклая оболочка $CP^{(i)} = Conv\left(\left(\bigcup_{j=1}^{n-1} CP_j\right) \cup CP_n^{(i)}\right)$; в качестве результата $(i + 1)$-й итерации берется многогранник

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

где ∇ – оператор расширения ($X \nabla Y$– это многогранник, описываемый той же системой неравенств, что и X, за вычетом неравенств, которым не удовлетворяет Y). Предполагается, что $CP_n^{(0)} = Conv(\bigcup_{j=1}^{n-1} CP_j)$.

Пример 7.5

Найдем инвариант приведенной ниже циклической программы [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 интерпретациями, выпишем явно множества возможных состояний программы при исполнении не более чем n итераций цикла. Ниже приведены такие множества для $n \le 3$ (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)}.

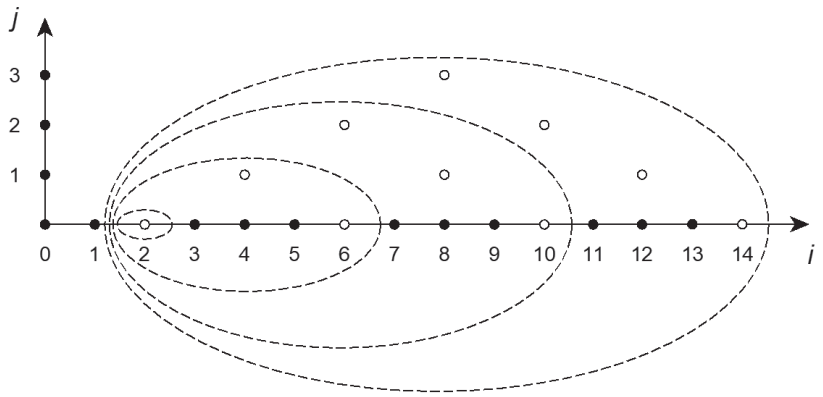
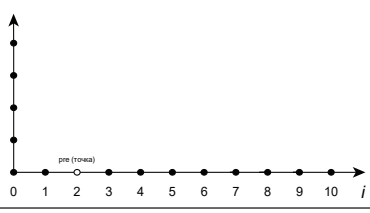
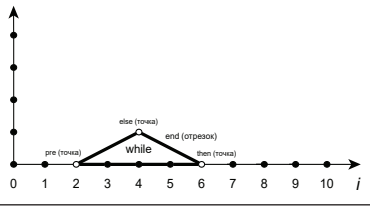
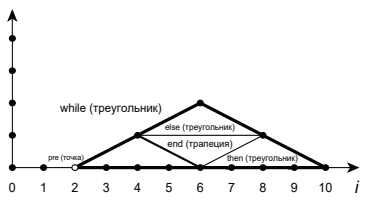
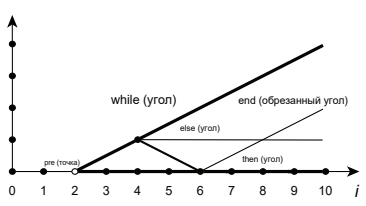


Рис. 7.1 ❖ Множества возможных состояний при исполнении не более чем n итераций ($n \in \{0, \ldots, 3\}$)

Рассмотрим теперь абстрактную интерпретацию этой программы в теории выпуклых многогранников (см. табл. 7.2). Введем следующие обозначения:

- $\varphi^{(i)}_{pre}$ – многогранник (система неравенств), соответствующий предусловию цикла: $\varphi_{pre} = \{i = 2, j = 0\}$ – многогранник, состоящий из одной точки (2, 0);
- $\varphi^{(i)}_{begin}$ – многогранник перед исполнением i-й итерации цикла: $\varphi^{(1)}_{begin} = \varphi_{pre} = \{i = 2, j = 0\}$; $\varphi^{(i+1)}_{begin} = \varphi^{(i)}_{begin} \nabla \varphi^{(i)}_{while}$ ($\varphi^{(i)}_{while}$ описан ниже);
- $\varphi^{(i)}_{then}$ – многогранник после исполнения ветви **then** условного оператора (i-я итерация цикла);
- $\varphi^{(i)}_{else}$ – многогранник после исполнения ветви **else** условного оператора (i-я итерация цикла);
- $\varphi^{(i)}_{end}$ – многогранник, соответствующий соединению $\varphi^{(i)}_{then}$ и $\varphi^{(i)}_{else}$;
- $\varphi^{(i)}_{while}$ – многогранник, соответствующий соединению φ_{pre} и $\varphi^{(i)}_{end}$.

Таблица 7.2. Абстрактная интерпретация в теории выпуклых многогранников

i	**Абстрактное состояние**	**Иллюстрация**
0	$\varphi_{pre} = \{i = 2, j = 0\}$	
1	$\varphi^{(1)}_{begin} = \varphi_{pre} = \{i = 2, j = 0\}^*$ $\varphi^{(1)}_{then} = \{i = 6, j = 0\}$ $\varphi^{(1)}_{else} = \{i = 4, j = 1\}$ $\varphi^{(1)}_{end} = Conv(\varphi^{(1)}_{then}, \varphi^{(1)}_{else}) = \{i + 2j = 6, 4 \le i \le 6\}$ $\varphi^{(1)}_{while} = Conv(\varphi_{pre}, \varphi^{(1)}_{end}) = \{2j + 2 \le i, i + 2j \le 6, 0 \le j\}$ * Нет многогранника для дуги обратной связи.	
2	$\varphi^{(2)}_{begin} = \varphi^{(1)}_{while} = \{2j + 2 \le i, i + 2j \le 6, 0 \le j\}^*$ $\varphi^{(2)}_{then} = \{2j + 6 \le i, i + 2j \le 10, 0 \le j\}$ $\varphi^{(2)}_{else} = \{2j + 2 \le i, i + 2j \le 10, 1 \le j\}$ $\varphi^{(2)}_{end} = Conv(\varphi^{(2)}_{then}, \varphi^{(2)}_{else}) = \{2j + 2 \le i, 6 \le i + 2j \le 10, 0 \le j\}$ $\varphi^{(2)}_{while} = Conv(\varphi_{pre}, \varphi^{(2)}_{end}) = \{2j + 2 \le i, i + 2j \le 6, 0 \le j\}$ * 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.	

Итак, для данной программы с помощью абстрактной интерпретации в теории выпуклых многогранников извлекается инвариант $(2 \cdot j + 2 \le i) \wedge (0 \le j)$.
□

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 программе из примера 7.5 имеют вид линейных ограничений. Каким образом их можно задействовать для абстрактной интерпретации в теории выпуклых многогранников?
9. Определите оператор расширения для интервальной арифметики.
10. Осуществите абстрактную интерпретацию в интервальной арифметике следующей программы [Cou77]:

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

11. Осуществите абстрактную интерпретацию в теории выпуклых многогранников программы из примера 7.4.
12. Используя абстрактную интерпретацию, найдите инвариант цикла программы из примера 7.5 для предусловия $(i \geq 0) \wedge (j \geq 0) \wedge (i + j \leq 2)$.

Лекция 8

Автоматическое доказательство теорем, SAT- и SMT-решатели

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 первого порядка. Вводятся понятия разрешающей процедуры и разрешимой теории, приводятся примеры разрешимых и неразрешимых теорий. Большое внимание уделяется базовым принципам работы SAT- и SMT-решателей – средств, составляющих ядро инструментов формальной верификации программ: 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 составляется множество формул (условий верификации и завершимости); их общезначимость (тождественная истинность) свидетельствует о корректности программы (см. лекции 5 и 6).

Таким образом, одной из важнейших задач дедуктивной верификации является проверка (доказательство или опровержение) полученных условий. Это может быть выполнено «на бумаге» или с помощью специальных программных средств [Har09]:

- *систем доказательства теорем* (*provers*);
- *систем проверки выполнимости*, так называемых *решателей* (*solvers*)[1];
- *систем помощи в доказательстве* (*proof assistants*);
- *систем проверки доказательств* (*proof checkers*).

Далее мы будем апеллировать к такому понятию, как (*формальная*) *теория*. *Теорией* называется четверка $T = \langle \Sigma, \Phi, A, R \rangle$ [Men64], где

- Σ – конечный или счетный *алфавит*;
- $\Phi \subseteq \Sigma^*$ – (счетное) множество *формул*;
- $A \subseteq \Phi$ – не более чем счетное множество *аксиом*;
- $R = \{R_i \subseteq (\Phi \times \ldots \times \Phi) \times \Phi\}_{i=1}^{n}$ – конечное множество *правил вывода*.

Обычно множество формул задается индуктивно (например, с помощью формальной грамматики) и существует алгоритм проверки, является ли данная формула аксиомой (обратите внимание: множество аксиом может быть бесконечным).

У термина «теория» имеется множество синонимов: «аксиоматика», «исчисление», «дедуктивная система» и др.

Мы уже говорили (см. лекцию 3), что *теоремами* теории называются такие формулы, которые могут быть *доказаны*, т. е. выведены из аксиом при помощи правил вывода (обозначение: $\vdash_T \varphi$, где T – рассматриваемая теория, или $\vdash \varphi$, если из контекста ясно, о какой теории идет речь).

Применение правил вывода было бы не более чем лингвистической игрой, если бы формулам теории не приписывался смысл – суждения о некоторой «реальности», допускающие проверку на истинность или ложность. Далее мы ограничимся *значимыми* теориями, т. е. теориями, в которых из доказуемости формул вытекает их истинность. Чтобы убедиться, что формула такой теории общезначима, достаточно построить ее вывод, однако автоматизировать этот процесс не всегда возможно.

Как правило, системы доказательства теорем не работают напрямую с аксиоматикой, а используют специализированные процедуры.

Разрешающей процедурой (*decision procedure*) теории T называется алгоритм DP_T, который для любой формулы φ этой теории ($\varphi \in \Phi_T$) завершается с выдачей вердикта (*verdict*):

- *true*, если φ – теорема (общезначимая формула);

---

[1] Между системами первого и второго типов четкой границы нет: термин «прувер» используют в отношении средств проверки общезначимости формул, термин «солвер» – в отношении средств проверки выполнимости (при выполнимости формулы солвер, как правило, возвращает ее модель – интерпретацию, в которой формула истинна).

- *false* в противном случае:

$$\langle \varphi \in \Phi_T \rangle DP_T \langle (verdict = true) \leftrightarrow (\vdash_T \varphi) \rangle.$$

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

Частичной разрешающей процедурой теории T будем называть алгоритм DP_T^{ω}, который для любой $\varphi \in \Phi_T$ в случае завершения работы определяет, является φ теоремой или нет:

$$\{\varphi \in \Phi_T\} DP_T^{\omega} \{(verdict = true) \leftrightarrow (\vdash_T \varphi)\}.$$

Теория называется *полуразрешимой*, если для нее существует частичная разрешающая процедура DP_T^s следующего вида:

$$\langle \varphi \in \Phi_T \wedge (\vdash_T \varphi) \rangle DP_T^s \langle verdict = true \rangle,$$

$$\{\varphi \in \Phi_T \wedge (\nvdash_T \varphi)\} DP_T^s \{verdict = false\}.$$

Проще говоря, для теорем DP_T^s завершается с положительным вердиктом; для других формул DP_T^s может не завершиться, но если завершается, то с отрицательным вердиктом.

Известно, что логика первого порядка (в целом) не является разрешимой, но является полуразрешимой. Частичной разрешающей процедурой для нее является, например, метод резолюций Робинсона[1] [Rob65]. К этому методу мы еще вернемся.

8.1.1. Примеры разрешимых и неразрешимых теорий

Важным направлением исследований в области автоматического доказательства теорем является поиск разрешимых теорий (прежде всего специализаций логики первого порядка) и создание для них эффективных разре-

---

[1] Джон Алан Робинсон (John Alan Robinson; 1930–2016) – британский философ и логик, автор метода резолюций (1965); внес основополагающий вклад в становление логического программирования; лауреат премии Эрбрана (1996).

К методу резолюций причастен также французский логик Жак Эрбран (Jacques Herbrand; 1908–1931) [чьим именем названа премия, лауреатом которой является Робинсон]: теорема Эрбрана (1930) – это фундамент, на котором построен метод Робинсона.

Кстати, фамилию Робинсон имеют несколько известных математиков-логиков: Джон Алан Робинсон (упомянутый выше), 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 Гильберта) и Джордж Робинсон (автор некоторых техник оптимизации в процедурах автоматического доказательства теорем).

шающих процедур. Для более глубокого изучения вопросов разрешимости и неразрешимости разных теорий можно порекомендовать работы [Har09] и [Kro08], мы же ограничимся примерами (см. табл. 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 год
Евклидова геометрия	Альфред Тарский, 1949 год
Гиперболическая геометрия	Вольфрам Швабхаузер (Wolfram Schwabhäuser), 1959 год
Теория первого порядка с равенством	Леопольд Лёвенгейм (Leopold Löwenheim), 1915 год
Теория первого порядка с равенством и одним унарным функциональным символом	Анджей Эренфойхт (Andrzej Ehrenfeucht), 1959 год
Арифметика Пресбургера (теория первого порядка, описывающая натуральные числа, с равенством, сложением, но без умножения)	Мойжеш Пресбургер (Mojżesz Presburger), 1929 год
Теория равенства неинтерпретируемых функциональных символов (бескванторная теория первого порядка с равенством и произвольным числом функциональных символов)	Вильгельм Фридрих Аккерман (Wilhelm Friedrich Ackermann), 1954 год
Бескванторная теория массивов (теория первого порядка с равенством, операцией получения элемента по индексу и операцией модификации элемента)	Джон Маккарти (John McCarthy), 1962 год

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 год
Арифметика натуральных чисел	Джон Баркли Россер (John Barkley Rosser), 1936 год
Арифметика целых чисел	Альфред Тарский, Анджей Мостовский (Andrzej Mostowski), 1949 год
Арифметика рациональных чисел	Джулия Холл Робинсон (Julia Hall Robinson), 1949 год

Таблица 8.2 *(окончание)*

Неразрешимая теория	**Кем и когда доказана неразрешимость**
Теории групп и колец	Альфред Тарский, 1949 год
Теория полей	Джулия Холл Робинсон, 1949 год
Арифметика Робинсона (фрагмент арифметики Пеано без схемы индукции)	Рафаэль Митчел Робинсон (Raphael Mitchel Robinson), 1950 год
Общая теория массивов (теория первого порядка с равенством, операцией получения элемента по индексу и операцией модификации элемента)	Джон Маккарти, 1962 год

8.1.2. Задача выполнимости, SAT- и SMT-решатели

Вспомним базовые определения математической логики.

Формула называется

- *выполнимой* (*SAT*, *satisfiable*), если существует *интерпретация*, в которой она истинна (каждая такая интерпретация называется *моделью*);
- *общезначимой*, если она истинна в любой интерпретации;
- *невыполнимой* (*UNSAT*, *unsatisfiable*), если она ложна во всех интерпретациях (не имеет моделей);
- *необщезначимой*, если существует интерпретация, в которой она ложна.

Общезначимость и выполнимость являются двойственными понятиями:

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

Задача проверки выполнимости формул логики высказываний называется *SAT* (*ВЫП*). Так как число интерпретаций любой формулы логики высказываний конечно, эта задача разрешима, однако может потребовать большого объема вычислений: известно, что для конъюнктивных нормальных форм (КНФ) задача SAT является NP-полной [Pap82]. Проверка выполнимости формул теорий первого порядка называется *SMT* (*Satisfiability Modulo Theories – выполнимость с учетом теорий*). Как отмечалось выше, для некоторых теорий эта задача разрешима, но для многих таковой не является.

Инструменты, проверяющие выполнимость формул логики высказываний и теорий первого порядка, называются *SAT-* и *SMT-решателями* соответственно.

Ниже мы рассмотрим основные 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 КНФ, или, что то же самое, в *клаузальной форме* [Ben12].

Напомним основные понятия:

- *литерал* (*буква*) – это элементарное высказывание (*атом*) или его отрицание;
- *клауза* (*дизъюнкт*) – множество *литералов*;
- *формула* (*клаузальная форма*) – множество клауз.

Клауза соответствует дизъюнкции литералов, а формула – конъюнкции дизъюнктов. Пустая клауза обозначается □, пустая формула – ∅.

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

Пример 8.1

Рассмотрим примеры клаузальных форм:

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

□

Почему именно КНФ? Во-первых, большинство формул, возникающих на практике, имеют вид конъюнкций условий: это касается как спецификаций требований («память выделена, размер массива неотрицателен, возвращаемый индекс принадлежит заданному диапазону»), так и внутренних формул, конструируемых в процессе верификации. Конечно, сами условия, соединяемые конъюнкцией, не обязаны иметь вид «плоских» дизъюнкций, однако приведение их к такой унифицированной форме упрощает реализацию решателей (без потери универсальности)[1]. Это вторая причина.

8.2.1. Кодировка Цейтина

Итак, для передачи формулы в SAT-решатель ее нужно преобразовать к КНФ. Только не стоит это делать «в лоб» (оцените размер результата при при-

[1] Более тонкий вопрос связан с производительностью решателей. С одной стороны, нормализация формул «стирает» полезную информацию, которую можно было бы задействовать при проверке выполнимости. С другой стороны, это открывает дорогу для многих оптимизаций, в том числе для аппаратного ускорения.

ведении ДНФ к эквивалентной КНФ). Следует принять в расчет, что преобразование не обязано быть эквивалентным: результатом может быть любая *равносильная* (*equisatisfiable*) КНФ, т. е. КНФ, выполнимая тогда и только тогда, когда выполнима исходная формула.

За счет введения дополнительных переменных можно построить равносильное представление, размер которого линейно зависит от размера формулы, подаваемой на вход. Идея такого построения принадлежит Г. С. Цейтину[1] [Цей68]; соответствующую процедуру сейчас называют *кодировкой* (*преобразованием*) *Цейтина* (*Tseitin encoding*).

Суть в следующем. Пусть φ – исходная формула, а $\psi \circ \chi$ – ее подформула (символ $\circ$ обозначает некоторую связку, для определенности двуместную). Если подформула $\psi \circ \chi$ «не вписывается» в структуру КНФ, она заменяется на *новое* элементарное высказывание $p_{\psi\circ\chi}$, при этом в конструируемую формулу добавляется конъюнктивный член $p_{\psi\circ\chi} \leftrightarrow (p_\psi \circ p_\chi)$, представленный в виде КНФ. Здесь p_γ – это либо γ, если γ – литерал, либо элементарное высказывание, сопоставленное γ.

Кодировку Цейтина можно изобразить в виде следующей рекурсивной процедуры:

$encode(\varphi) \equiv p_\varphi \wedge link(\varphi, p_\varphi)$;

$link(\psi \circ \chi, p_{\psi\circ\chi}) \equiv cnf(p_{\psi\circ\chi} \leftrightarrow (p_\psi \circ p_\chi)) \wedge link(\psi, p_\psi) \wedge link(\chi, p_\chi)$;

$link(\neg\psi, p_{\neg\psi}) \equiv link(\psi, \bar{p}_{\neg\psi})$;

$link(\gamma, p_\gamma) \equiv true$, если γ – литерал (в этом случае $p_\gamma \equiv \gamma$).

В табл. 8.3 приведены КНФ-представления эквивалентностей вида $f \leftrightarrow (p \circ q)$.

Таблица 8.3. Представление некоторых логических операций в виде КНФ

Операция	КНФ	Клаузальная форма
$f \leftrightarrow (p \wedge q)$	$(\bar{p} \vee \bar{q} \vee f) \wedge (p \vee \bar{f}) \wedge (q \vee \bar{f})$	$\{\bar{p}\bar{q}f, p\bar{f}, q\bar{f}\}$
$f \leftrightarrow (p \vee q)$	$(\bar{p} \vee \bar{q} \vee \bar{f}) \wedge (\bar{p} \vee f) \wedge (\bar{q} \vee f)$	$\{pq\bar{f}, \bar{p}f, \bar{q}f\}$
$f \leftrightarrow (p \to q)$	$(\bar{p} \vee q \vee \bar{f}) \wedge (p \vee f) \wedge (\bar{q} \vee f)$	$\{\bar{p}q\bar{f}, pf, \bar{q}f\}$
$f \leftrightarrow (p \oplus q)$	$(\bar{p} \vee \bar{q} \vee \bar{f}) \wedge (\bar{p} \vee q \vee f) \wedge (p \vee \bar{q} \vee f) \wedge (p \vee q \vee \bar{f})$	$\{\bar{p}\bar{q}\bar{f}, \bar{p}qf, p\bar{q}f, pq\bar{f}\}$

Пример 8.2

Построим кодировку Цейтина для формулы $(p \oplus q) \to \bar{r}$.

[1] Григорий Самуилович Цейтин (1936–2022) – советский/российский и американский ученый в области информатики и математики; наиболее известен разработкой способа преобразования формул логики высказываний к равносильным КНФ соизмеримого размера (1968); выдающийся член ACM (2006).

Сопоставив этой формуле новую переменную f в подформуле $(p \oplus q) - h$, получим:

$$f \wedge cnf(f \leftrightarrow (h \to \bar{r})) \wedge cnf(h \leftrightarrow (p \oplus q)) \equiv$$

$$f \wedge (\bar{h} \vee \bar{r} \vee \bar{f}) \wedge (h \vee f) \wedge (r \vee f) \wedge (\bar{p} \vee \bar{q} \vee \bar{h}) \wedge (\bar{p} \vee q \vee h) \wedge (p \vee \bar{q} \vee h) \wedge (p \vee q \vee \bar{h}) \equiv$$

$$\{f, \bar{h}\bar{r}\bar{f}, hf, rf, \bar{p}\bar{q}\bar{h}, \bar{p}qh, p\bar{q}h, pq\bar{h}\}.$$

□

8.2.2. Метод резолюций для логики высказываний

Для опровержения клаузальных форм (под *опровержением* здесь понимается доказательство невыполнимости) применяется процедура, называемая *резолюцией* [Ben12]. Процедура состоит в последовательном применении *правила резолюции*, сохраняющем свойство выполнимости, т. е. преобразующем формулу F к равносильной формуле F' (обозначение: $F' \approx F$). Если на некотором шаге выводится клауза □, то исходная формула невыполнима (соответственно, ее отрицание общезначимо).

Дадим необходимые определения.

Если l – литерал, то литерал $l^C = \begin{cases} \bar{p}, & \text{если } l = p \\ p, & \text{если } l = \bar{p} \end{cases}$ называется *контрарным* к l, а пара литералов l и l^C – *контрарной парой*.

Пусть клауза C_1 содержит литерал l, а клауза C_2 – контрарный литерал l^C. Тогда говорят, что клаузы C_1 и C_2 *сталкиваются* (*clash*) по контрарной паре l и l^C, при этом клауза

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

называется *резольвентой* C_1 и C_2.

Обратите внимание: $Res(C_1, C_2)$ является логическим следствием $C_1 \cup C_2$.

Правило резолюции состоит в добавлении в клаузальную форму резольвенты каких-нибудь двух сталкивающихся клауз (при наличии). Если клаузы сталкиваются не по одной, а по нескольким контрарным парам, правило можно не применять – получится тавтология:

$$Res(C_1 \cup \{l_1, l_2\}, C_2 \cup \{l_1^C, l_2^C\}) = C_1 \cup C_2 \cup \{l_2, l_2^C\} = true.$$

Отметим, что классическое правило вывода *modus ponens* $\left(\frac{p, p \to q}{q}\right)$ является частным случаем правила резолюции:

$$Res(p, p \to q) = Res(p, \bar{p}q) = q.$$

Пример 8.3

Клаузы $pq\bar{r}$ и $qr\bar{s}$ сталкиваются по контрарной паре $\bar{r}$ и r. Соответствующая резольвента определяется равенством $Res(pq\bar{r}, qr\bar{s}) = pq \cup q\bar{s} = pq\bar{s}$.

□

Процедура резолюции описывается следующим псевдокодом:

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

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

return true
```

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

1) $C_n = C$;
2) для всех $i \in \{1, \ldots, n\}$:
   a) либо $C_i \in F$;
   b) либо C_i – резольвента некоторых предшествующих клауз.

Резолютивный вывод □ из F является *опровержением* F.

Процедура резолюции является значимой и полной процедурой опровержения клаузальных форм; кроме того, она всегда завершается [Ben12].

Пример 8.4

Докажем невыполнимость формулы $\{p, \bar{p}q, \bar{r}, \bar{p}\bar{q}r\}$, используя метод резолюций [Ben12]. Вот один из возможных резолютивных выводов:

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

□

8.2.3. Алгоритм DPLL проверки выполнимости

Пожалуй, самым известным алгоритмом проверки выполнимости КНФ (простой перебор не в счет) является DPLL [Dav62]. Метод назван в честь Дэвиса[1], Патнэма[2], Логемана[3] и Лавленда[4]. Иногда этот алгоритм называют DLL: дело в том, что Патнэма не было в числе авторов статьи [Dav62], однако он (совместно с Дэвисом) разработал предшествующий вариант метода – алгоритм DP [Dav60].

Оба подхода основаны на простых правилах вывода, важнейшее из которых – *распространение единицы* (*unit propagation*, или *BCP*, *Boolean Constraint Propagation*) – является частным случаем правила резолюции[5].

Правило 0: удаление тавтологий (вспомогательное)
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 (основное)
Клауза, состоящая из одного 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 (дополнительное к 1)
Чистым литералом (*pure literal*) 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 $F' \approx F$.

---

[1] Мартин Дэвид Дэвис (Martin David Davis; 1928–2023) – американский математик; вместе с Хилари Патнэмом внес вклад в решение десятой проблемы Гильберта – доказательство алгоритмической неразрешимости диофантовых уравнений (завершил доказательство советский/российский математик Юрий Владимирович Матиясевич); лауреат премии Эрбрана (2005).

[2] Хилари Уайтхолл Патнэм (Hilary Whitehall Putnam; 1926–2016) – влиятельный американский философ; известен своими работами в области методологии науки, философии языка и сознания, логики; автор (совместно с Солом Крипке) каузальной теории референции; лауреат премии Рольфа Шока (2011).

[3] Джорж Уолл Логеман (George Wahl Logemann; 1938–2012) – американский информатик, соавтор алгоритма DPLL.

[4] Дональд Лавленд (Donald W. Loveland; род. в 1934 году) – американский ученый, специализирующийся в области автоматизированного доказательства теорем; лауреат премии Эрбрана (2001).

[5] Следует заметить, что на практических задачах исполнение BCP «съедает» 50–90 % машинного времени [Mah04], поэтому структуры данных, используемые для представления КНФ, специально оптимизируют под это правило.

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, ложную в I') then
    return false
  end;

  // решение
  if F' = ∅ (все клаузы F' истинны 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: $F'[p := x]$), 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)[1].

В чем отличие алгоритма DPLL от предшествующего ему алгоритма DP? В алгоритме Дэвиса–Патнема вместо поиска модели (перебора интерпретаций с возвратами) используется основанная на резолюции процедура исключения переменной. Правила 1 и 2 остаются, но когда они не применимы, делается следующее:

- выбирается элементарное высказывание p;
- строятся все возможные резольвенты клауз, сталкивающихся по p и $\neg p$;
- построенные резольвенты добавляются в множество клауз;
- из множества удаляются клаузы, содержащие p или $\neg p$.

На практике поиск модели, т. е. алгоритм DPLL, оказывается эффективнее.

Очевидно, что на время работы алгоритма DPLL влияет как выбор высказывания, по которому осуществляется ветвление, так и очередность сопоставления ему значений истинности. За это отвечают эвристики. Для оптимизации также используют *нехронологические возвраты* (*NCB*, *Non-Chronological Backtracking*), основанные на *анализе конфликтов* (*CDCL*, *Conflict-Driven Clause Learning*) [Mar96]:

- если на некоторой частичной интерпретации формула оказывается ложной, анализируется причина конфликта;
- возврат делается не на уровень выше, а на уровень, где было первое причастное к конфликту присваивание;
- выводится новая клауза – отрицание «конфликтующих» присваиваний (например, для $p := true$ и $q := false$ выводится $\bar{p}q$, т. е. $p = false$ или $q = true$);
- выведенная клауза добавляется в формулу (это своего рода защита от бесперспективных присваиваний).

Такой подход используется во многих SAT-решателях, например в MiniSat [MinS] и zChaff [Chaff]. Желающим узнать, как устроены современные

[1] Удаление здесь следует воспринимать абстрактно: в конкретной реализации за этой операцией может стоять установка или сброс флага, изменение счетчика и т. п.

реализации DPLL, рекомендуем диссертацию Линьтао Чжана[1] [Zha03], где подробно описываются применяемые эвристики и оптимизации.

Есть и другие подходы к задаче SAT: метод на основе *двоичных решающих диаграмм* (*BDD*, *Binary Decision Diagrams*) (см. лекцию 14) и метод Стольмарка[2] [Har09].

8.3. Теории первого порядка и SMT-решатели

Метод резолюций может использоваться и для логики первого порядка. Ранее мы упоминали, что общий вариант метода был предложен Робинсоном [Rob65].

Известно, что любую замкнутую, т. е. не содержащую свободных переменных, формулу логики первого порядка можно свести к равносильной (не обязательно эквивалентной) клаузальной форме[3]. Это делается в несколько этапов [Ben12]:

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 *скулемовской стандартной форме* (*ССФ*) – элиминация кванторов существования с помощью *скулемовских функций*[4];
4) запись матрицы ССФ (представленной в КНФ) в виде множества клауз.

---

[1] Линьтао Чжан (Lintao Zhang) – известный специалист в области проверки выполнимости и формальной верификации компьютерных систем; сотрудник Microsoft Research.

[2] Гуннар Стольмарк (Gunnar Stålmarck) – шведский информатик; автор оригинального (и запатентованного) метода проверки выполнимости формул логики высказываний (1989).

[3] Если формула не является замкнутой, ее выполнимость не изменится при «навешивании» на свободные переменные кванторов существования. Таким образом, можно считать, что формула, имеющаяся в нашем распоряжении, замкнута.

[4] Название связано с именем норвежского математика Туральфа Альберта Скулема (Thoralf Albert Skolem; 1887–1963), предложившего следующий подход к элиминации кванторов существования: при элиминации $\exists y$ переменная y заменяется на терм $f(x_1, \ldots, x_n)$, где f – новый функциональный символ (так называемая *скулемовская функция*), а $x_1, \ldots, x_n$ – переменные кванторов всеобщности, синтаксически предшествующие $\exists y$.

8.3.1. Метод резолюций для логики первого порядка

Элементарной формулой, или *атомом*, в логике первого порядка называется формула вида $P(t_1, \ldots, t_n)$, где P – предикатный символ арности n, а $t_1, \ldots, t_n$ – термы. Определения литерала, клаузы (дизъюнкта) и формулы (клаузальной формы) сохраняются (с точностью до понятия атома).

Общая схема метода резолюций остается прежней, но поиск контрарных литералов становится сложнее и требует привлечения механизма *унификации*.

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

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

Унификатор θ называется *наиболее общим унификатором* (*НОУ*), если любой другой унификатор η (для того же множества формул) может быть представлен в виде суперпозиции θ и некоторой подстановки μ:

$$\eta = \theta\mu.$$

Пусть C_1 и C_2 – клаузы, не имеющие общих переменных[1], а $L_1 = \{l_{1,i}\}_{i=1}^{n}$ и $L_2^C = \{l_{2,i}^C\}_{i=1}^{m}$ – их непустые подмножества, такие что объединение L_1 и $L_2^C = \{l_{2,i}^C\}_{i=1}^{m}$ может быть унифицировано. Пусть θ – НОУ $L_1 \cup L_2^C$. 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 первого порядка, но может не завершиться для выполнимых формул.

Известны оптимизации и частные случаи метода резолюций, с которыми можно познакомиться, например, в книге [Cha73]. Альтернативой является «обратный метод» Маслова[2] [Mac64], предложенный примерно в то же время, что и метод резолюций. Обзор этого метода есть в дополнении к русскому изданию [Cha73].

---

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

[2] Сергей Юрьевич Маслов (1939–1982) – советский математик; специалист по дедуктивным системам и автоматизации доказательства теорем; автор обратного метода установления выводимости (1964).

Для построения НОУ можно использовать алгоритм Мартелли–Монтанари[1] [Mar82]. Первоначальный алгоритм унификации, применяемый Робинсоном, менее эффективен.

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:

$P(t_1, \ldots, t_n) = P(t'_1, \ldots, t'_n)$ тогда и только тогда, когда $t_i = t'_i$ для всех $i \in \{1, \ldots, n\}$.

На вход алгоритму подается система уравнений, которая преобразуется по следующим правилам (пока хотя бы одно из них применимо):

1) изменить уравнение вида $t = x$, где t – терм, не являющийся переменной, а x – переменная, на $x = t$;
2) удалить уравнение вида $x = x$, где x – переменная;
3) для уравнения вида $t_1 = t_2$, где t_1 и t_2 – термы, не являющиеся переменными:
   a) если внешние функциональные символы t_1 и t_2 различаются:
      i) *система не имеет решения*;
   b) иначе, т. е. если уравнение имеет вид $f(t_1, \ldots, t_k) = f(t'_1, \ldots, t'_k)$:
      i) заменить его на систему уравнений $\{t_i = t'_i\}_{i=1}^{k}$;
4) для уравнения вида $x = t$, где x – переменная, имеющая другие вхождения, а t – терм, отличный от x:
   a) если x входит в t:
      i) *система не имеет решения*;
   b) иначе:
      i) заменить все другие вхождения x на t.

Результатом работы алгоритма (при наличии решения) является система в разрешенном виде: в левых частях равенств стоят переменные, не имеющие вхождений в правые части. Эта система и задает НОУ.

Метод резолюций, будучи общим подходом к проверке выполнимости формул, не учитывает семантику функциональных и предикатных символов (описывающую их теорию). Конечно, если формула невыполнима вообще, она невыполнима ни в какой теории, однако это большая редкость, когда решение принимается исключительно из логических соображений. Если теорию можно выразить в виде конечного числа формул, то, представив их в клаузальной форме и используя на общих основаниях в резолюции, мы учтем теорию, но, как правило, это не самый эффективный путь.

Говоря о разрешающих процедурах теорий, отметим, что единого подхода к их построению нет: для каждой теории ищется своя процедура, как можно более эффективная.

---

[1] Уго Монтанари (Ugo Montanari; род. в 1943 году) – итальянский ученый, известный своими работами в области оснований программирования; вместе с Альберто Мартелли (Alberto Martelli) является автором алгоритма унификации формул; член Европейской академии (Academia Europaea).

8.3.2. Теория равенства

В этом разделе мы познакомимся с *теорией равенства* (полное название: *бескванторная теория равенства неинтерпретируемых функциональных символов*) [Nel80]. Сигнатура теории включает произвольное число функциональных символов (в том числе констант) и двуместный предикатный символ равенства (=). Аксиомы отражают рефлексивность, симметричность и транзитивность равенства, а также свойство *подстановочности*:

$$\left(\bigwedge_{i=1}^{k}(t_i = t_i')\right) \to f(t_1, \ldots, t_k) = f(t_1', \ldots, t_k').$$

Разрешимость теории доказал Аккерман[1], а эффективную разрешающую процедуру предложили Нельсон[2] и Оппен[3] [Nel80]. Процедура основана на построении *конгруэнтного замыкания* отношения на графе; в настоящее время известны алгоритмы решения этой задачи с вычислительной сложностью $O(m \log m)$, где m – число дуг графа [Nei07].

Ориентированным графом с помеченными вершинами и упорядоченными дугами называется пятерка $G = \langle V, \Sigma, L, \deg^+, [\cdot] \rangle$, в которой:

- V – множество вершин;
- Σ – алфавит пометок;
- $L: V \to \Sigma$ – функция разметки вершин;
- $\deg^+: V \to \mathbb{N}_0$ – полустепени исхода вершин:
  - $\deg^+(v)$ – число дуг, исходящих из вершины v;
- $[\,]: V \times \mathbb{N} \to V$ – занумерованное множество дуг:
  - $v[i]$ – конец i-й дуги, исходящей из вершины v, где $1 \le i \le \deg^+(v)$[4].

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

1) $L(v) = L(u)$;
2) $\deg^+(v) = \deg^+(u)$;

---

[1] Вильгельм Фридрих Аккерман (Wilhelm Friedrich Ackermann; 1896–1962) – немецкий математик, логик и педагог, ученик Давида Гильберта; в теории алгоритмов его именем названа функция $A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1,1) & m > 0 \text{ и } n = 0. \\ A\big(m-1, A(m,n-1)\big) & m > 0 \text{ и } n > 0 \end{cases}$

[2] Грегори Нельсон (Charles Gregory Nelson; 1953–2015) – американский ученый-информатик; автор (совместно с Дереком Оппеном) разрешающей процедуры для теории равенства (1976), быстрого алгоритма построения конгруэнтного замыкания (1977), метода комбинирования разрешающих процедур бескванторных теорий (1978); лауреат премии Эрбрана (2013).

[3] Дерек Оппен (Derek Charles Oppen; 1948–2017) – исследователь Стэнфордского университета (1975–1980), вице-президент компании Nortel (1996–2002).

[4] Концевые вершины дуг с разными номерами могут совпадать.

3) для всех $1 \le i \le \deg^+(v)$:
   a) либо $v[i] = u[i]$;
   b) либо $(v[i], u[i]) \in R$.

Обратите внимание: каждая вершина конгруэнтна сама себе (по любому отношению).

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: *отношение эквивалентности* – это бинарное отношение, удовлетворяющее свойствам рефлексивности, симметричности и транзитивности.

Пусть проверяется выполнимость конъюнкции

$$(t_1 = t'_1) \wedge \ldots \wedge (t_p = t'_p) \wedge (s_1 \neq s'_1) \wedge \ldots \wedge (s_q \neq s'_q),$$

где t_i, t'_i, s_j и s'_j – некоторые термы.

Разрешающая процедура состоит из следующих шагов:

1) построить граф $G = \langle V, \Sigma, L, \deg^+, [\cdot] \rangle$ над множеством термов:
   a) V – термы, входящие в конъюнкцию, и их подтермы;
   b) Σ – функциональные символы и переменные;
   c) пометить вершины (для всех $v \in V$):
      i) если v – переменная или константа:
         1) положить $L(v) = v$;
      ii) если $v = f(v_1, \ldots, v_k)$, где $v_1, \ldots, v_k$ – некоторые термы:
         1) положить $L(v) = f$;
   d) построить дуги (для всех $v \in V$):
      i) если v – переменная или константа:
         1) положить $\deg^+(v) = 0$;
      ii) если $v = f(v_1, \ldots, v_k)$:
         1) положить $\deg^+(v) = k$;
         2) положить $v[i] = v_i$ для всех $i \in \{1, \ldots, k\}$;
4) построить конгруэнтное замыкание R^* отношения $R = \{(t_i, t'_i) \mid i \in \{1, \ldots, p\}\}$:
   a) положить $R^* = R$;
   b) пока возможно:
      i) добавить в R^* пару вершин $(v, u) \notin R^*$, такую что истинно хотя бы одно из условий[1]:
         1) v и u конгруэнтны по отношению R^*;
         2) $(u, v) \in R^*$;
         3) $(v, w) \in R^*$ и $(w, u) \in R^*$ для некоторой вершины w;

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

5) если $(s_i, s'_i) \notin R^*$ для всех $i \in \{1, \ldots, q\}$:
   а) *конъюнкция выполнима*;
6) в противном случае:
   а) *конъюнкция невыполнима*.

Пример 8.5

Проиллюстрируем работу этой разрешающей процедуры на простом примере:

$F \equiv (f(g(x)) \neq f(a)) \wedge (g(x) = a)$.

По термам формулы F строится граф G (см. рис. 8.1).

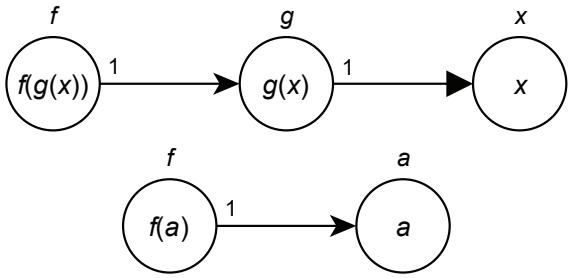


Рис. 8.1 ❖ Граф G, соответствующий термам формулы F

Для отношения $R = \{(g(x), a)\}$ (единственное равенство в F есть $g(x) = a$) на множестве вершин графа G строится конгруэнтное замыкание R^*:

- исходное отношение:
  - $(g(x), a)$;
- конгруэнтность (включая рефлексивность):
  - (a, a), (x, x), $(f(a), f(a))$, $(g(x), g(x))$, $(f(g(x)), f(g(x)))$;
  - $(f(g(x)), f(a))$;
- симметричность:
  - $(a, g(x))$, $(f(a), f(g(x)))$;
- транзитивность не задействуется.

Единственным неравенством в F является литерал $f(g(x)) \neq f(a)$. Проверяется вхождение пары $(f(g(x)), f(a))$ в отношение R^*. Поскольку вхождение имеет место, формула невыполнима.

□

8.3.3. Комбинирование теорий

Условия верификации программ часто выражаются в терминах нескольких теорий (целочисленная арифметика, операции над массивами, функцио-

нальные уравнения). Чтобы оперировать с такими условиями, нужен метод, позволяющий по разрешающим процедурам отдельных теорий получить разрешающую процедуру для их совокупности.

Первым (и до сих пор используемым) подходом к комбинированию разрешающих процедур в SMT-решателях является метод Нельсона–Оппена [Nel79] (это те же исследователи, что предложили разрешающую процедуру для теории равенства). Метод применим для *бесквантворных теорий*, не имеющих общих функциональных и предикатных символов, за исключением символа *равенства*, обязательного для всех комбинируемых теорий.

Идея подхода состоит в следующем. По исходной формуле (конъюнкции литералов) строится множество формул (также конъюнкций), 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: выводимость равенства может быть установлена разрешающей процедурой соответствующей теории.

Метод Нельсона–Оппена, как он описан выше, применим (является значимым и полным) только для *выпуклых теорий*, т. е. теорий, удовлетворяющих следующему свойству: из формулы теории (конъюнкции литералов) выводится непустая дизъюнкция равенств переменных тогда и только тогда, когда из нее выводится хотя бы одно их этих равенств в отдельности [Nel79].

Примеры выпуклых и невыпуклых теорий приведены в табл. 8.3.

Таблица 8.3. Примеры выпуклых и невыпуклых теорий

Выпуклая теория	**Невыпуклая теория**
Линейная арифметика (сложение, умножение на константу, константы 0 и 1, сравнения = и ≤) над $\mathbb{R}$	Линейная арифметика над $\mathbb{Z}$
Теория равенства	Теория массивов

Почему целочисленная линейная арифметика не выпукла? Например, из формулы

$$(x = 1) \wedge (y = 2) \wedge (1 \le z) \wedge (z \le 2)$$

выводится дизъюнкция равенств $(x = z) \vee (y = z)$, однако ни одно из них не выводится отдельно от другого [Nel79]. Для вещественных чисел теория выпукла, но не потому, что в аналогичной ситуации в ней выводятся отдельные равенства, а потому, что не выводится дизъюнкция.

Для невыпуклых теорий процедура усложняется. Когда перестают выводиться равенства, предпринимается попытка вывести дизъюнкцию равенств. Если это невозможно, исходная формула выполнима. Если же выводится дизъюнкция

$(x_1 = y_1) \vee \ldots \vee (x_k = y_k)$,

процедура рекурсивно применяется к k подзадачам (*case split*):

$F \wedge (x_1 = y_1), \ldots, F \wedge (x_k = y_k)$, где F – текущая система формул.

Исходная формула выполнима тогда и только тогда, когда выполнима хотя бы одна из них.

Пример 8.6

Поясним работу метода Нельсона–Оппена на примере формулы, включающей арифметические соотношения (линейная арифметика) и функциональное неравенство (теория равенства) [Неп88]:

$F \equiv (x \le y) \wedge (y + z \le x) \wedge (z \ge 0) \wedge (f(f(x) - f(y)) \ne f(z))$.

Вначале по формуле F строятся две формулы: F_1 и F_2 ($F \approx F_1 \wedge F_2$):

$F_1 \equiv (x \le y) \wedge (y + z \le x) \wedge (z \ge 0) \wedge (w_1 = w_2 - w_3)$;

$F_2 \equiv (w_2 = f(x)) \wedge (w_3 = f(y)) \wedge (f(w_1) \ne f(z))$.

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:

$F_2' \equiv (w_2 = f(x)) \wedge (w_3 = f(y)) \wedge (f(w_1) \ne f(z)) \wedge (x = y)$.

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 $w_1 = z$ (обе переменные равны 0), которое добавляется в F_2':

$F_2'' \equiv (w_2 = f(x)) \wedge (w_3 = f(y)) \wedge \underline{(f(w_1) \ne f(z))} \wedge (x = y) \wedge \underline{(w_1 = z)}$.

Формула F_2'' невыполнима (нарушение аксиомы подстановочности равенства), 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. Используя кодировку Цейтина, постройте равносильную КНФ для следующих формул:
   a) $(x \wedge y) \rightarrow z$;
   b) $(x \wedge \neg y) \vee (\neg x \oplus z)$;
   c) $maj(x, y, z)$, где maj – функция голосования.
6. Представьте в виде формулы логики высказываний вариант задачи «Голубятня» (Pigeon hole) для $n = 2$ (требуется разместить $n + 1$ голубей по n клеткам так, чтобы в каждой клетке было не больше одного голубя). Используя кодировку Цейтина, преобразуйте формулу к равносильной КНФ. Проверьте выполнимость формулы с помощью алгоритма DPLL.
7. Приведите к ПНФ следующие формулы (A и B – бескванторные 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) $\exists x\forall y\, A(x, y) \rightarrow \exists x\forall y\, B(x, y)$.
8. 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) $\forall x\exists y\forall v\exists z\, R(x, y, z, v)$.
9. Постройте НОУ (если возможно) для следующих пар атомов [Ben12]:
   a) $P(a, x, f(g(y)))$ $\quad$ $P(y, f(z), f(z))$;
   b) $P(x, g(f(a)), f(x))$ $\quad$ $P(f(a), y, y)$;
   c) $P(x, g(f(a)), f(x))$ $\quad$ $P(f(y), z, y)$;
   d) $P(a, x, f(g(y)))$ $\quad$ $P(z, h(z, u), f(u))$.
10. Используя разрешающую процедуру для теории равенства, докажите выполнимость $(x \neq y) \wedge (f(x) = f(y))$ и невыполнимость $(x = y) \wedge (f(x) \neq f(y))$.
11. Используя разрешающую процедуру для теории равенства, проверьте выполнимость формулы $f(f(f(\mathrm{a}))) = a \wedge f\big(f(f(f(f(a))))\big) = a \wedge f(a) \neq a$ [Неп88].

12. Какие теории называются выпуклыми? Приведите примеры выпуклых и невыпуклых теорий. Докажите, что теория массивов (см. лекцию 3) не является выпуклой.
13. Напишите программу (на удобном вам языке программирования), реализующую процедуру резолюции для логики высказываний.
14. Напишите программу, реализующую алгоритм DPLL проверки выполнимости КНФ.
15. Напишите программу, реализующую алгоритм унификации Мартелли–Монтанари.
16. Напишите программу, реализующую процедуру резолюции для логики первого порядка.
17. Напишите программу, реализующую разрешающую процедуру для теории равенства.

Лекция 9

Параллельные программы и логика LTL

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 (на каждом шаге исполняется оператор одного из процессов, выбранного недетерминированным образом). Особое внимание уделяется так называемым реагирующим системам, которые работают в «бесконечном цикле» и отвечают 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 параллельных программ используется алгоритм Петерсона взаимного исключения двух процессов.

9.1. Параллельные программы

До настоящего момента мы рассматривали *последовательные* программы. Реальные системы часто состоят из нескольких *параллельно* исполняемых

процессов: это и программы обработки больших массивов данных, и клиент-серверные приложения, и разного рода системы управления. Если, не ограничиваясь ПО, обратиться к аппаратуре (микропроцессорам, видеокартам, маршрутизаторам), станет очевидно, что параллелизм – не исключение, а правило. Заметим, что для проектирования микросхем используют специальные языки, так называемые HDL (Hardware Description Languages), например Verilog и VHDL [Har12], так что модели аппаратуры – те же программы, просто особого вида.

Разработка параллельной программы – непростая задача, требующая аккуратной *синхронизации* процессов. Ошибки, связанные с синхронизацией, обнаружить крайне сложно: они проявляются лишь при определенном порядке исполнения операторов, а число возможных порядков огромно[1] (как правило, программист не знает, когда планировщик приостановит один процесс и возобновит другой). В лекции 1 мы упоминали случай с аппаратом лучевой терапии «Therac-25», когда ошибка синхронизации стала причиной переоблучения и смерти нескольких человек. 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 (далее просто *программой*) называется конечное множество последовательных программ над общим множеством переменных. Отдельная программа этого множества называется *процессом*. Переменные, используемые в рамках одного процесса, называются *локальными*; переменные, задействованные в нескольких процессах, – *разделяемыми*, или *глобальными*.

Для программы $\{P_1, \ldots, P_n\}$ часто используют запись $P_1 \parallel \cdots \parallel P_n$.

Будем считать, что процессы программы занумерованы: запись P_i обозначает процесс с *номером* (*идентификатором*) $i \in \{1, \ldots, n\}$. Также будем считать, что каждый оператор *op* снабжен меткой $\lambda \in \Lambda$, уникальной в рамках одного процесса. Метка записывается перед оператором и отделяется от него двоеточием:

λ: *op*.

Пусть V – множество переменных, а D – их общий домен (множество значений). *Конфигурацией* программы $P_1 \parallel \cdots \parallel P_n$ называется пара $\langle l, s \rangle$, где

[1] Явление экспоненциального роста числа состояний параллельной программы (или числа возможных вычислений) в зависимости от числа процессов известно как *комбинаторный взрыв* (*state explosion*).

- l: $\{1, \ldots, n\} \to \Lambda$ – *состояние управления*[1];
- s: $V \to D$ – *состояние данных*.

Если l – состояние управления, то $l_i \equiv l(i)$ – метка текущего (еще не исполненного) оператора процесса P_i (*точка исполнения*).

Запись $P_i@\lambda_i$, или просто $@\lambda_i$ (если из контекста ясно, о каком процессе идет речь), обозначает тот факт, что исполнение процесса P_i находится в точке λ_i.

9.1.2. Асинхронный параллелизм: семантика чередований

Семантика параллельной программы может быть определена с использованием парадигмы *чередования* (*interleaving*), известной также как *асинхронный параллелизм*. Пусть процессы описаны на языке программирования с формализованной операционной семантикой, например на языке while (см. лекцию 2). Тогда отношение переходов программы $P \equiv P_1 \,||\, \cdots \,||\, P_n$ определяется следующим семейством правил вывода:

$$\frac{\lambda_i \neq \lambda_\omega, \langle \lambda_i, s\rangle \xrightarrow[i]{} \langle \lambda_i', s'\rangle}{\langle l, s\rangle \to \langle l[i := \lambda_i'], s'\rangle}, \quad i \in \{1, \ldots, n\}.$$

Здесь λ_i – точка исполнения процесса P_i, λ_ω – псевдометка, обозначающая завершение процесса, а $\xrightarrow[i]{}$ и $\to$ суть отношения переходов процесса P_i и программы P в целом.

Следует подчеркнуть, что поведение параллельной программы *недетерминировано*: оно зависит от порядка исполнения процессов. Стратегию выбора процесса, так же как и сущность, реализующую эту стратегию, будем называть *планировщиком*. Реальные планировщики в операционных системах могут быть вполне себе детерминированными, однако пользовательские приложения должны быть рассчитаны на работу в разном окружении: корректность должна обеспечиваться на всех возможных вычислениях (см. лекцию 2).

Используя концепцию планировщика, исполнение параллельной программы $P_1 \,||\, \cdots \,||\, P_n$ можно записать следующим образом (λ_i^0 – начальная метка процесса P_i, s_0 – начальное состояние данных):

```
⟨l, s⟩ := ⟨{1 ↦ λ₁⁰, …, n ↦ λₙ⁰}, s₀⟩;    // инициализация
Active := {1, …, n};                     // множество активных процессов
while Active ≠ ∅ do                      // пока есть активные процессы:
  i := schedule(Active);                 // - выбирается один из них
  ⟨l, s⟩ := execute(i, ⟨l, s⟩);          // - исполняется текущий оператор
  Active := {i | l_i ≠ λ_ω}              // - множество обновляется
end
```

[1] Здесь вместо меток операторов можно использовать программы, подлежащие исполнению, как это принято в структурной операционной семантике (см. лекцию 2).

Пример 9.1

Рассмотрим программу $P \equiv P_1 \parallel P_2$, в которой

$$P_1 \equiv \lambda_1 : x := y \quad \text{и} \quad P_2 \equiv \lambda_2 : y := x,$$

с условием корректности

$$\langle (x = a) \wedge (y = b) \rangle P \langle (x = b) \wedge (y = a) \rangle,$$

где a и b – некоторые константы.

Проще говоря, от программы P требуется поменять местами значения переменных x и y.

Начальным состоянием данных является означивание $\{x \mapsto a, y \mapsto b\}$, или кратко – (a, b). В соответствии с определенной выше семантикой чередования отношение переходов программы P состоит из четырех переходов:

- $\langle (\lambda_1, \lambda_2), (a, b) \rangle \to \langle (\lambda_\omega, \lambda_2), (b, b) \rangle$;
- $\langle (\lambda_\omega, \lambda_2), (b, b) \rangle \to \langle (\lambda_\omega, \lambda_\omega), (b, b) \rangle$;
- $\langle (\lambda_1, \lambda_2), (a, b) \rangle \to \langle (\lambda_1, \lambda_\omega), (a, a) \rangle$;
- $\langle (\lambda_1, \lambda_\omega), (a, a) \rangle \to \langle (\lambda_\omega, \lambda_\omega), (a, a) \rangle$.

Таким образом, в состоянии (a, b) есть два возможных вычисления программы P: результатом одного из них является состояние (b, b); другого – (a, a). Ни то, ни другое не удовлетворяет постусловию (если, конечно, $a \neq b$)[1].

□

«Ошибка» в примере может быть объяснена несоответствием замысла программиста и используемой *модели вычислений*. Существуют модели, в которых приведенная программа корректна. Пусть, например, исполнение устроено так:

- на каждом шаге определяется множество активных процессов;
- операторы активных процессов исполняются согласованно, используя двухфазную дисциплину:
  1) считываются значения переменных и производятся вычисления;
  2) результаты вычислений записываются в переменные[2].

Такая модель вычислений, известная как *синхронный параллелизм*, широко используется в цифровой схемотехнике [Har12]. Работа системы управляется *тактовым сигналом*, фронты которого (скачки с низкого уровня на высокий или, наоборот, с высокого на низкий) разбивают непрерывное время на дискретные единицы – *такты*. За время такта по входным сигналам

---

[1] Для того чтобы программа была некорректной относительно предусловия φ и постусловия ψ, достаточно существования хотя бы одного вычисления $\pi = \{s_i\}_{i=0}^{n}$, такого что $s_0 \vDash \varphi$ и $s_n \nvDash \psi$.

[2] Параллельные присваивания в одну переменную либо запрещаются, либо осуществляются с применением специальных механизмов разрешения конфликтов (арбитров).

и битам состояния (значениям *триггеров*) вычисляются выходные сигналы и биты следующего состояния (записываемые в триггеры по ближайшему фронту).

В известном смысле синхронность исполнения процессов упрощает верификацию, избавляя от необходимости рассматривать разные порядки исполнения (с синхронным параллелизмом мы еще столкнемся в лекциях 14 и 15, посвященных символьной проверке моделей).

9.1.3. Реагирующие системы и справедливость планировщика

Далее мы будем рассматривать *асинхронные* параллельные программы, работающие к тому же в «бесконечном цикле». Речь идет о так называемых *реагирующих системах* (*reactive systems*), целью которых является поддержание взаимодействия с окружением. Реагирующие системы идентифицируют входные *события* и отвечают на них, исполняя те или иные *действия*. Это обширный класс программ, включающий операционные системы, драйверы устройств, телекоммуникационные среды и т. п.

На данном этапе под *событием* будем понимать условие на значения глобальных переменных (выполнение [срабатывание] этого условия в процессе вычисления), а под *действием* – часть программы, запускаемую, когда соответствующее условие становится истинным (исполнение этой части программы).

Пример 9.2

Рассмотрим *синхронный канал* передачи сообщений, также известный как *рандеву*. Канал передает сообщение от процесса-источника (*src*) к процессу-приемнику (*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 получению сообщения;
- *dstMsg* – сообщение, принимаемое приемником.

Реализация канала приведена ниже (процессы, взаимодействующие через канал, нам не интересны и поэтому не показаны).

```
while true do                    // бесконечный цикл
  if srcRdy = 1 ∧ dstRdy = 1     // проверка готовности сторон
  then                           // если обе стороны готовы:
    dstMsg := srcMsg;            // - передача сообщения
    dstRdy := 0;                 // - уведомление сторон
```

```
    srcRdy := 0                  //   о завершении передачи
  else                           // если хотя бы одна сторона не готова:
    skip                         // - канал бездействует
  end
end
```

В примере можно выделить событие готовности к передаче сообщения

$(srcRdy = 1) \wedge (dstRdy = 1)$

и действие по передаче сообщения (и сбросу флагов готовности)

$dstMsg := srcMsg; \quad dstRdy := 0; \quad srcRdy := 0.$

□

При анализе свойств реагирующих систем часто предполагают, что планировщик является *справедливым* (*fair*): каждый процесс периодически (время от времени) выбирается для исполнения; другими словами, невозможна ситуация, когда какой-нибудь процесс ожидает своей очереди бесконечно долго. К этой теме мы вернемся в конце лекции.

9.2. Спецификация реагирующих систем

Поведение реагирующих систем описывается в терминах *событий* (*стимулов* и *реакций*), распределенных во *времени*. Для спецификации требований к реагирующим системам часто используются так называемые *темпоральные*, *или временны́е*, *логики* – формальные языки, позволяющие задавать взаимосвязи событий во времени: причинно-следственные связи, ограничения на относительный порядок, величины задержек между событиями (в логиках реального времени) и т. п.

Пример 9.3

Приведем примеры *темпоральных свойств*:

- система работает *до тех пор*, *пока* не будет остановлена;
- если запрос принят, он *когда-нибудь* будет обработан;
- при наличии двух или более заявок заявка с бóльшим приоритетом будет обслужена *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

Особое распространение в информатике получили две темпоральные логики:

- *логика линейного времени* (*LTL*, *Linear-time Temporal Logic*) [Pnu77];
- *логика ветвящегося времени* (*CTL*, *Computation Tree Logic*) [Cla86].

В нашей книге мы будем использовать первую из них – LTL. Эта логика была предложена Амиром Пнуэли[1] и является первой темпоральной логикой, предназначенной для спецификации и верификации программ.

В LTL к синтаксису классической логики высказываний добавлены два так называемых *темпоральных оператора*:

- унарный оператор **X** (от англ. *neXt time* – в следующий момент времени)[2];
- бинарный оператор **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):

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi\ \mathbf{U}\ \varphi,$$

где p – это элементарное высказывание из множества *AP*.

Для удобства в формулах LTL можно использовать производные логические связки (например, $\wedge$ и $\rightarrow$), логические константы (*true* и *false*), а также производные темпоральные операторы, включая

- унарный оператор **F** (от англ. *in the Future* – когда-нибудь в будущем);
- унарный оператор **G** (от англ. *Globally* – глобально, всегда).

На содержательном уровне темпоральные операторы интерпретируются так:

- $\mathbf{X}\varphi$ – формула φ станет истинной 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 *в будущем*;
- $\mathbf{G}\varphi \equiv \neg\mathbf{F}\neg\varphi$ – формула $\neg\varphi$ ложна *сейчас* и *никогда* не станет истинной *в будущем* (*всегда*, начиная с настоящего момента времени, истинна формула φ).

---

[1] Амир Пнуэли (Amir Pnueli; 1941–2009) – израильский и американский ученый-информатик, известный своими работами по темпоральной логике и ее применению к спецификации и верификации свойств программ; лауреат премии Тьюринга (1996).

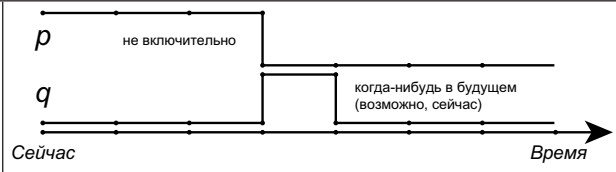
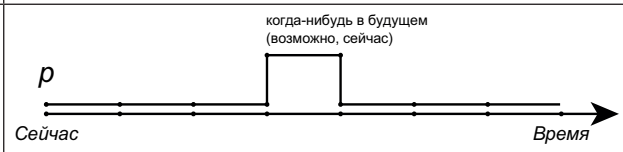
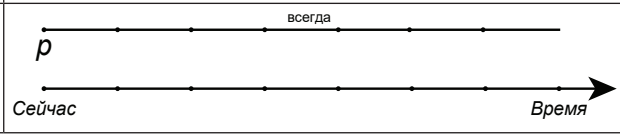
[2] Обычно рассматриваются темпоральные логики будущего времени, хотя имеются разновидности и для прошедшего времени.

Обратите внимание: для истинности формулы **F**φ достаточно истинности φ в настоящий момент времени. Чтобы исключить из рассмотрения настоящее время, нужно «сдвинуть» формулу вдоль временно́й оси оператором **X**:

XFφ.

Семантика темпоральных операторов LTL иллюстрируется в табл. 9.1 с помощью временны́х диаграмм.

Таблица 9.1. Временны́е диаграммы темпоральных операторов

Формула LTL	**Временна́я диаграмма**
X*p* – в следующий момент времени	
p **U** *q* – до тех пор, пока не	
F*p* – когда-нибудь в будущем	
G*p* – всегда	

Если нужно выразить свойство «формула φ истинна, пока ложна формула ψ», не требуя при этом, чтобы формула ψ стала когда-нибудь истинной, это можно сделать с помощью оператора **W**, называемого *слабым* **U** (от англ. *Weak until*):

φ **W** $\psi \equiv (\varphi$ **U** $\psi) \vee$ **G**φ.

Другим полезным оператором является **R** (от англ. *Release* – освобождение, разблокировка) – оператор, двойственный к **U**:

φ **R** $\psi \equiv \neg(\neg\varphi$ **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 (после

этого ограничение на истинность ψ снимается); если такой момент никогда не наступит, формула ψ остается истинной навсегда (говорят: *φ освобождает ψ*).

Пример 9.4

Выразим в логике LTL следующее свойство: «память, занимаемая объектом, не будет освобождена сборщиком мусора, пока счетчик ссылок не станет равным нулю»:

(счетчик ссылок = 0) **R** ¬(память освобождается).

Только после того, как счетчик ссылок станет равным нулю, память *может* быть освобождена (что не означает, что она будет освобождена сразу после этого или будет освобождена вообще).

□

Рассмотрим две полезные формулы LTL:

- **GF**φ – формула φ будет истинной бесконечное число раз (один из вариантов свойства *живости*);
- **FG**φ – когда-нибудь формула φ станет истинной и останется такой навсегда (свойство *стабилизации*).

Пример 9.5

Увлекательным занятием является формализация высказываний, выраженных на естественном языке, средствами той или иной логики, например LTL. В [Кар10] приведено много занятных примеров.

- «Мы будем бороться, пока не победим»:
  - (мы боремся) **U** (мы победили): есть уверенность в будущей победе, но для этого нужно побороться;
  - (мы боремся) **W** (мы победили): победа не гарантирована, но бороться все равно нужно[1].
- «Сегодня он играет джаз, а завтра Родину продаст» (из песни Владлена Бахнова):
  - (он играет джаз) → **X**(он продает Родину): если он *сегодня* (прямо сейчас) играет джаз, то *завтра* (считаем, что «период дискретизации» составляет один день) он продаст Родину;
  - **G**{(он играет джаз) → **X**(он продает Родину)}: если *когда-нибудь* он начнет играть джаз, то *на следующий день* после этого он продаст Родину;
  - **G**{(он играет джаз) → **XF**(он продает Родину)}: если *когда-нибудь* он начнет играть джаз, то *после этого* (рано или поздно) он продаст Родину[2].

□

[1] Как справедливо заметила одна студентка, обе формализации допускают «излишнюю агрессию» – борьбу после победы.

[2] Представленные формулы не исключают возможности многократной «продажи Родины».

9.2.2. Структуры Крипке и семантика LTL

Семантика темпоральных логик может быть определена с помощью такого формализма, как *структура Крипке*[1] [Kri63]. В общих словах это система возможных миров (состояний) и переходов между ними. Каждый мир статичен и интерпретируется традиционным образом, например заданием отображения, сопоставляющего каждому элементарному высказыванию значение истинности: *true* или *false*.

Дадим определение. Пусть задано множество элементарных высказываний *AP*. *Структурой Крипке* над множеством *AP* называется четверка $\langle S, S_0, R, L\rangle$, где

- S – (конечное) множество состояний;
- $S_0 \subseteq S$ – (непустое) 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.

Итак, состояния структуры Крипке – это мгновенные «снимки» некоторого модельного мира. «Тик часов» происходит при переходе между состояниями. Естественным ограничением, налагаемым на отношение переходов, является возможность бесконечного *прогресса времени*:

для каждого $s \in S$ должно существовать $s' \in S$, такое что $(s, s') \in R$.

Другими словами, в структуре Крипке не должно быть тупиковых состояний (в любом состоянии должна быть исходящая дуга, хотя бы петля). Отношение переходов, удовлетворяющее этому ограничению, называется *полным*, или *тотальным*.

В дальнейшем будем рассматривать только структуры Крипке с полными отношениями 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 $t \in \mathbb{N}_0$ – некоторый момент времени. 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 (структуры и формулы должны быть над общим множеством элементарных высказываний). Истинность формулы φ на вычислении $\pi = \{s_i\}_{i=0}^{\infty}$ (обозначение: $\pi \vDash \varphi$) определяется индуктивно по структуре формулы, как показано в табл. 9.2.

---

[1] Сол Аарон Крипке (Saul Aaron Kripke; 1940–2022) – американский философ; наиболее известен своим подходом к интерпретации модальных логик через систему возможных миров (семантика Крипке); лауреат премии Рольфа Шока (2001).

Таблица 9.2. Семантика логики LTL

(1)	$\pi \vDash p$, где $p \in AP$	$s_0 \vDash p$, т. е. $p \in L(s_0)$
(2)	$\pi \vDash \neg\varphi$	$\pi \nvDash \varphi$
(3)	$\pi \vDash \varphi \vee \psi$	$\pi \vDash \varphi$ или $\pi \vDash \psi$
(4)	$\pi \vDash \mathbf{X}\varphi$	$\pi^{(1)} \vDash \varphi$
(5)	$\pi \vDash \varphi\ \mathbf{U}\ \psi$	существует такое $T \geq 0$, что $\pi^{(T)} \vDash \psi$ и для всех $0 \leq t \leq T$ имеет место $\pi^{(t)} \vDash \varphi$

Как следствие, семантика операторов **F** и **G** определяется в соответствии с табл. 9.3.

***Таблица 9.3. Семантика темпоральных операторов* F *и* G**

(6)	$\pi \vDash \mathbf{F}\varphi$	существует такое $T \geq 0$, что $\pi^{(T)} \vDash \varphi$
(7)	$\pi \vDash \mathbf{G}\varphi$	для любого $t \geq 0$ выполняется $\pi^{(t)} \vDash \varphi$

Заметим: истинность формул LTL определяется не столько вычислениями, сколько индуцированными ими траекториями (важны не сами состояния, а их пометки). Приведенное определение практически без изменений переписывается в терминах траекторий.

9.2.3. Аксиоматическое определение LTL

Логику LTL можно определить более абстрактно – аксиоматически. Дедуктивная система, задающая LTL (один из возможных вариантов), показана в табл. 9.4 [Dam].

Таблица 9.4. Аксиоматическое определение логики LTL

(0)	φ	**Аксиома**: общезначимая формула логики высказываний с подстановкой вместо элементарных высказываний любых формул LTL
(1)	$\mathbf{G}\{\varphi \to \psi\} \to (\mathbf{G}\varphi \to \mathbf{G}\psi)$	**Аксиома**: дистрибутивность оператора **G** относительно импликации
(2)	$\neg\mathbf{X}\varphi \leftrightarrow \mathbf{X}\neg\varphi$	**Аксиома**: линейность оператора **X**
(3)	$\mathbf{X}\{\varphi \to \psi\} \to (\mathbf{X}\varphi \to \mathbf{X}\psi)$	**Аксиома**: дистрибутивность оператора **X** относительно импликации
(4)	$\mathbf{G}\{\varphi \to \mathbf{X}\varphi\} \to (\varphi \to \mathbf{G}\varphi)$	**Аксиома**: индукция по времени
(5)	$(\varphi\ \mathbf{U}\ \psi) \leftrightarrow (\psi \vee (\varphi \wedge \mathbf{X}\{\varphi\ \mathbf{U}\ \psi\}))$	**Аксиома**: развертка оператора **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**: *modus ponens*

Таблица 9.4 *(окончание)*

(8)	$\dfrac{\varphi}{\mathbf{G}\varphi}$	**Правило вывода**: глобализация[1]
(9)	$\dfrac{\varphi}{\mathbf{F}\varphi}$	**Правило вывода**: осуществимость[2]

Известно, что приведенная аксиоматика является значимой и полной (см. лекцию 3) [Dam]. Нам она интересна как источник тождеств и правил преобразования формул.

Обратите внимание: темпоральные операторы **F** и **G** участвуют здесь на общих основаниях, а не как частные случаи **U**; эквивалентность $\mathbf{F}\varphi \leftrightarrow (\mathit{true}\ \mathbf{U}\ \varphi)$ является в этой дедуктивной системе теоремой (попробуйте построить доказательство).

9.3. Взаимное исключение процессов

Параллельные программы, как и последовательные, можно верифицировать дедуктивно. Мы не будем касаться методов верификации параллельных программ на соответствие программным контрактам; желающим самостоятельно изучить этот вопрос рекомендуем монографии [Неп88] и [Apt09]. Предмет нашего интереса – реагирующие системы, специфицированные с помощью LTL.

9.3.1. Алгоритм Петерсона

Для примера обратимся к алгоритму Петерсона[3] взаимного исключения процессов [Pet81]. Пусть программа P состоит из двух однотипных процессов P_0 и P_1, работающих в бесконечном цикле. Периодически каждый из них входит

---

[1] Правило глобализации выражает следующую мысль: если формула истинна *вообще* (безотносительно времени), то она истинна *всегда*. В определенном смысле оно соответствует правилу *универсального обобщения* $\dfrac{\varphi}{\forall t\varphi}$, где t – переменная (интерпретируемая здесь как время).

[2] Правило осуществимости выражает такую мысль: если формула истинна *вообще*, то она станет истинной *когда-нибудь*.

[3] Гари Петерсон (Gary L. Peterson) – американский информатик, известный в основном как автор рассматриваемого здесь алгоритма взаимного исключения двух процессов (1981).

в *критическую секцию* (критическая секция процесса P_i помечена меткой CRS_i – см. код ниже). Вход процесса в критическую секцию осуществляется в два этапа:

- сначала процесс сообщает о своем намерении (метка SET_i);
- затем он ожидает возможности войти (метка TST_i).

Выход из критической секции производится путем сброса флага, установленного при входе (метка RST_i).

```
while true do
  NCS_i: // некритическая секция
  SET_i: flag_i := 1; turn := i; // запрос
  TST_i: while (flag_1-i = 1) ∧ (turn = i) do skip end; // ожидание
  CRS_i: // критическая секция
  RST_i: flag_i := 0 // выход
end
```

В программе три переменные:

- $flag_0$ и $flag_1$ – флаги запросов на вход в критическую секцию от процессов P_0 и P_1 соответственно;
- *turn* – номер процесса, желающего войти в критическую секцию (одних только флагов недостаточно).

Начальное состояние задается равенствами $turn = 0$ и $flag_i = 0$, где $i \in \{0, 1\}$.

Алгоритм Петерсона корректен относительно следующей LTL-спецификации[1] [Ben12]:

- $\mathbf{G}\{\neg(@CRS_0 \wedge @CRS_1)\}$ – свойство *безопасности* (*safety*)[2]: два процесса не могут одновременно пребывать в критической секции;
- $\mathbf{G}\{@SET_i \rightarrow \mathbf{F}@CRS_i\}$, где $i \in \{0, 1\}$, – свойства *живости* (*liveness*)[3]: если процесс запросил вход в критическую секцию, то рано или поздно его запрос будет удовлетворен.

Свойство живости (в рассматриваемом примере, и не только) может быть доказано только в предположении *справедливости* (*fairness*) планировщика:

$$fairness \rightarrow liveness.$$

Под *корректностью* модели (алгоритма) относительно формулы LTL понимается истинность формулы на всех возможных вычислениях модели.

---

[1] Читатели, знакомые с таким понятием, как *консистентность памяти* (*memory consistency*), могут справедливо заметить, что алгоритм Петерсона корректен лишь для «сильных» моделей консистентности. Семантика чередований, определенная выше, предполагает *последовательную консистентность* (*sequential consistency*) – этого вполне достаточно.

[2] В общем случае под *свойством безопасности* понимается свойство, для опровержения которого достаточно предъявить контрпример конечной длины.

[3] В отличие от свойств безопасности, *свойства живости* могут быть опровергнуты только контрпримерами бесконечной длины.

9.3.2. Справедливость планировщика while-программ

Для программ на языке while справедливость можно формализовать следующим образом: для каждого оператора строится формула LTL, как показано в табл. 9.6 (считаем, что в конце кода каждого процесса есть метка λ_ω, означающая его завершение); конъюнкция всех этих формул и есть условие справедливости.

Таблица 9.6. Формулы LTL, выражающие справедливость планировщика

(1)	λ: **skip**; λ': P	$\mathbf{G}\{@\lambda \to \mathbf{F}@\lambda'\}$
(2)	λ: $x := t$; λ': P	см. (1)
(3)	λ: **if** B **then** λ_1: P_1 **else** λ_2: P_2 **end**	$\mathbf{G}\{@\lambda \to \mathbf{F}\{@\lambda_1 \vee @\lambda_2\}\}$
(4)	λ: **while** B **do** λ_1: P_1 **end**; λ_2: P_2	см. (3)

Вопрос: как дополнить формулу $\mathbf{G}\{@\lambda \to \mathbf{F}\{@\lambda_1 \vee @\lambda_2\}\}$ из строки (3), чтобы уточнить, когда переход осуществляется по метке λ_1, а когда – по λ_2?

$\mathbf{G}\{(@\lambda \wedge B) \to \mathbf{F}@\lambda_1\}$ и $\mathbf{G}\{(@\lambda \wedge \neg B) \to \mathbf{F}@\lambda_2\}$?

Предположим, что процесс находится в точке λ и истинно условие B. Можно ожидать, что исполнение пойдет по ветви **then**. Пусть, однако, перед тем как процесс проверил условие, планировщик выбрал другой процесс, который изменил состояние так, что условие стало ложным. Когда наступит черед первого процесса, его исполнение пойдет по ветви **else**, а не **then**, как ожидалось.

Можно предложить такой вариант:

$\mathbf{G}\{@\lambda \to @\lambda\ \mathbf{U}\ ((B \wedge @\lambda_1) \vee (\neg B \wedge @\lambda_2))\}$.

Располагая формализованной операционной семантикой языка while и условиями справедливости планировщика (подобными тем, что приведены в табл. 9.6), можно в дедуктивной манере доказать корректность алгоритма Петерсона относительно приведенной выше спецификации [Ben12]. Мы не будем это делать, поскольку есть более простой и элегантный способ – *проверка модели* (*model checking*)[1], которому посвящены следующие лекции.

9.4. Вопросы и упражнения

1. Дайте определение параллельной программы. В чем, на ваш взгляд, состоит основная трудность верификации параллельных программ?

[1] Появление метода model checking связывают с именами Эдмунда Мельсона Кларка (Edmund Melson Clarke, Jr.; 1945–2020), Эрнеста Аллена Эмерсона (Ernest Allen Emerson II; род. в 1954 году) и Иосифа Сифакиса (Ιωσήφ Σηφάκης; род. в 1946 году). За разработку метода ученые удостоились премии Тьюринга (2007).

2. В чем разница между синхронным и асинхронным параллелизмом?
3. Расширим язык while оператором **parallel** $P_1 \parallel P_2$ **end**, суть которого состоит в порождении двух параллельных процессов (*fork*) и ожидании их завершения (*join*). Определите операционную семантику этого языка.
4. Дополнительно к предыдущему заданию расширим язык оператором **atomic** P **end** (где P – программа, не содержащая операторов **parallel**), ограничивающим чередования процессов: если процесс находится во внутренней точке **atomic**-блока, он не может быть прерван другим процессом. Определите операционную семантику этого языка.
5. В асинхронных параллельных программах обычно не используется темпоральный оператор **X**. Чем это объясняется?
6. 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) $\mathbf{FG}q$?
7. Формализуйте в логике LTL следующие высказывания:
   a) «событие x никогда не произойдет»;
   b) «событие x произойдет ровно один раз»;
   c) «событие x произойдет не менее двух раз»;
   d) «событие x произойдет не более двух раз»;
   e) «событие x произойдет ровно два раза».
8. Представьте в виде формулы LTL следующие высказывания:
   a) «Канал может терять сообщение только конечное число раз»;
   b) «Между каждой парой событий a происходит событие b»;
   c) «В ответ на стимул x всегда возникает реакция y, причем между x и y
      i) нет событий z»;
      ii) есть хотя бы одно событие z»;
      iii) есть ровно одно событие z».
9. Формализуйте высказывание «сегодня он играет джаз, а завтра Родину продаст» с помощью оператора **R**.
10. Приведите формулу $\neg(\varphi\ \mathbf{U}\ \psi)$ к *отрицательной нормальной форме*, т. е. к форме, в которой отрицания стоят непосредственно перед элементарными высказываниями [Кар10].
11. Определите семантику темпоральных операторов **W** и **R** на вычислении структуры Крипке. Нарисуйте соответствующие временны́е диаграммы.
12. Докажите следующие эквивалентности:
    a) $\varphi\ \mathbf{U}\ \psi \equiv \mathbf{F}\psi \wedge (\varphi\ \mathbf{W}\ \psi)$;
    b) $\varphi\ \mathbf{R}\ \psi \equiv \psi\ \mathbf{W}\ (\varphi \wedge \psi)$;
    c) $\varphi\ \mathbf{W}\ \psi \equiv \psi\ \mathbf{R}\ (\varphi \vee \psi)$.

13. 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) $\varphi\ \mathbf{U}\ \psi \equiv (\varphi \vee \psi)\ \mathbf{U}\ \psi$.
14. Обоснуйте на содержательном уровне значимость следующих правил вывода:
    a) $\dfrac{\varphi \to \psi}{\mathbf{G}\varphi \to \mathbf{G}\psi}$;
    b) $\dfrac{\varphi \to \psi}{\mathbf{X}\varphi \to \mathbf{X}\psi}$;
    c) $\dfrac{\varphi \to \mathbf{X}\varphi}{\varphi \to \mathbf{G}\varphi}$.
15. Докажите формально следующие утверждения:
    a) $\mathbf{XG}\{\varphi \to \psi\} \to (\mathbf{XC}\varphi \to \mathbf{XG}\psi)$;
    b) $\varphi \to (\varphi\ \mathbf{U}\ \varphi)$;
    c) $\mathbf{FG}\varphi \to \mathbf{GF}\varphi$.

Лекция 10

Инструмент проверки моделей Spin

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 их соответствие LTL-спецификациям, т. е. проверять, является ли заданная модель (программа на языке Promela) «моделью» (в логическом понимании) заданной формулы LTL (истинна ли формула на всех вычислениях модели). Описанные средства могут быть использованы на практике при проектировании протоколов, 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]. Вместе с инструментом Spin (Simple Promela Interpreter) он был

разработан Джерардом Хольцманом[1] в Исследовательском центре компьютерных наук (Computer Sciences Research Center) при лабораториях Bell. Язык не предназначен для реализации программных или аппаратных систем: это язык *моделирования*, причем упор в нем сделан на средствах координации процессов, а не на вычислениях.

Promela-модели состоят из *процессов*, работающих параллельно и взаимодействующих через общие *переменные* или посредством передачи *сообщений* через *каналы*. Процессы могут порождаться как *статически*, в момент создания модели, так и *динамически*, при исполнении других процессов. Метод *проверки моделей* (*model checking*) предполагает, что общее число созданных процессов конечно (подробнее об этом мы поговорим в следующих лекциях).

10.1.1. Типы данных, переменные и выражения

Базовые типы данных языка Promela включают *целочисленные типы* (**int**, **unsigned** и т. п.) и *специализированные типы* (**pid**, **chan** и **mtype**). К составным типам относятся *массивы* и *структуры*. Информация о целочисленных типах представлена в табл. 10.1.

Таблица 10.1. Целочисленные типы данных языка Promela

Тип данных	Множество значений	Битовая длина
bit	{0, 1}	1
bool	{*false*, *true*}	1
byte	[0, 255]	8
short	$[-2^{15}, 2^{15} - 1]$	16
int	$[-2^{31}, 2^{31} - 1]$	32 (как минимум)
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:

- **pid** – идентификатор процесса (множество значений[2]: [0, 255]);
- **chan** – идентификатор канала ([0, 255], 0 – неинициализированный канал);
- **mtype** – идентификатор типа сообщения ([1, 255]).

Декларация переменных в Promela осуществляется, как показано ниже:

$Tx_1 [= t_1], \ldots, x_n [= t_n];$

---

[1] Джерард Хольцман (Gerard J. Holzmann; род. в 1951 году) – голландско-американский исследователь в области верификации компьютерных протоколов; разработчик инструмента Spin; лауреат ACM Software System Award (2001).

[2] Возможно, в новых версиях Spin разрядность специализированных типов будет увеличена.

Здесь T – тип данных, x_i – имя переменной, t_i – необязательное выражение, называемое *инициализатором* (по умолчанию переменные инициализируются нулями).

Переменные делятся на *глобальные* (*разделяемые*), декларируемые вне процессов, и *локальные*, декларируемые внутри процессов и в них же используемые.

Пример 10.1

Рассмотрим примеры деклараций [Hol03]. Обратите внимание на три момента:

1) в одной декларации можно указать несколько переменных;
2) комментарии обрамляются /* и */ или начинаются с // (как в языке программирования C);
3) при декларации переменных типа **unsigned** задается их разрядность.

```
bit x, y;           /* две 1-битные переменные (значение 0) */
bool turn = true;   // булева переменная (значение true = 1)
unsigned v:5;       /* беззнаковая 5-битная переменная (значение 0) */
unsigned w:3 = 5;   // беззнаковая 3-битная переменная (значение 5)
```

□

Массивы в Promela могут быть только *одномерными* и декларируются следующим образом:

```
T x[N] [ = t];
```

Здесь T – тип элементов, x – имя, N – длина (натуральное число), t – инициализатор:

- либо выражение типа T (в этом случае все элементы получают одинаковое начальное значение);
- либо конструкция вида $\{t_0, \ldots, t_{N-1}\}$, где t_i – выражение, задающее начальное значение i-го элемента (индексация начинается с 0).

Доступ к элементам массива осуществляется привычным образом: $x[i]$, где x – имя массива, а i – индекс.

Указателей и адресной арифметики в Promela нет.

Пример 10.2

Приведем примеры деклараций массивов и скалярных переменных.

```
byte a[10];             // a[0] = ... = a[9] = 0
short b[2] = {-1, 1};   // b[0] = -1, b[1] = 1
int c, d[3] = 1, e = 2; // c = 0, d[0] = d[1] = d[2] = 1, e = 2
```

□

Упомянутый ранее тип **mtype** является аналогом перечислений (**enum**) языка C. Название происходит от англ. *message type* (символы, определяемые

с помощью **mtype**, часто используют для обозначения типов сообщений). Синтаксис следующий:

```
mtype = {c1, ..., cn};
```

Здесь c_i – разные идентификаторы, отличные от ключевых слов.

Допускается несколько определений **mtype**, что эквивалентно одному определению с объединенным множеством символов.

Пример 10.3

Ниже вводятся символьные обозначения для распространенных типов сообщений.

```
// определение типов сообщений
mtype = { req, ack, nack, err };
```

`req` – это запрос, `ack` – подтверждение, `nack` – отрицательное подтверждение, `err` – ошибка передачи.

□

Структурный тип определяется, как показано ниже:

```
typedef S { T1 f1; ... ; Tn fn };
```

Здесь S – имя типа, а T_i и f_i – соответственно тип и имя i-го поля. Заметим: типы полей могут быть как базовыми, так и составными, в частности массивами[1].

Для доступа к полям структуры используется общепринятая нотация: $s.f$, где s – имя переменной структурного типа, а f – имя поля.

Пример 10.4

Определим структурный тип для представления пакетов, передаваемых по сети:

```
// определение структурного типа Packet
typedef Packet {
  mtype type;    // тип сообщения
  short source;  // адрес источника
  short target;  // адрес приемника
  byte  data[4]  // передаваемые данные
};
```

Объявим переменную этого типа:

```
// декларация переменной p типа Packet
Packet p;
```

[1] Эту особенность можно использовать для определения многомерных массивов.

Присвоим значения некоторым полям:

```
// доступ к полям переменной p
p.type    = req;
p.target  = 1;
p.data[0] = 2;
```

□

Выражения Promela похожи на выражения C. Одно из немногих отличий – форма записи *условной операции*:

$(B \text{ -> } t_1 \text{ : } t_2)$.

Вместо символа ? используется ->; кроме того, выражение заключается в скобки.

Другое отличие: присваивания в Promela не возвращают значений и, таким образом, не являются выражениями в полном смысле слова (например, их нельзя использовать в правых частях присваиваний).

Операции (в порядке убывания приоритета) и их ассоциативность (способ расстановки скобок) представлены в табл. 10.2 [Hol03].

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) вычисляется значение *t* согласно приоритетам и ассоциативности операций;
3) вычисленное значение приводится к типу переменной *x* и ей присваивается.

10.1.2. Процессные типы и процессы

Функциональность модели описывается в одном или нескольких *процессных типах*. Активная роль в реализации этой функциональности отводится *процессам* – экземплярам процессных типов.

Определение процессного типа начинается с ключевого слова **proctype**, после которого следуют имя, формальные параметры и тело. Локальные переменные, если они нужны, лучше декларировать в начале тела (синтаксис Promela допускает объявление переменных в любом месте, однако неявно все декларации переносятся в начало [Hol03]):

proctype $P(T_1\ p_1;\ \ldots;\ T_n\ p_n)\ \{\ S_1\ x_1;\ \ldots;\ S_m\ x_m;\ \ldots\ \}$

Заметим, что параметры, как и переменные, можно декларировать группами:

$A\ a_1,\ a_2,\ \ldots;\ B\ b_1,\ b_2,\ \ldots;\ \ldots$

Процессный тип описывает поведение семейства процессов, но не порождает их (если не указан модификатор **active** – об этом ниже).

Точкой входа в Promela-модель является *процесс инициализации* **init**:

```
init { ... }
```

В нем (как и в других местах модели) можно порождать процессы, используя конструкцию

run $P(t_1,\ \ldots,\ t_n)$,

где P – имя процессного типа, t_i – фактические параметры, т. е. выражения, задающие значения формальных параметров.

Процесс завершается при достижении конца тела соответствующего процессного типа, но не раньше, чем завершатся все порожденные им процессы.

Еще один способ запуска процесса – модификатор **active** перед **proctype**. В этом случае процесс создается в начале работы модели (если у процессного типа есть параметры, им присваиваются нулевые значения). Чтобы запустить несколько процессов, можно использовать конструкцию **active** [N], где N – число порождаемых процессов.

У каждого процесса имеется предопределенная локальная переменная `_pid` типа **pid**, хранящая его идентификатор (целое число от 0 до 255).

Пример 10.5

Определим простой процессный тип: процесс печатает строку «hello, world: *n*», где *n* – идентификатор процесса.

```
active [3] proctype main() {
  // функция печати аналогична функции printf в C
  printf("hello, world: %d\n", _pid)
}
```

Модификатор **active** [3], присутствующий в определении, говорит о том, что будет создано 3 процесса.

Для запуска модели в Spin наберите команду

```
spin имя-файла
```

Инструмент выведет на консоль что-то подобное:

```
   hello world: 0
           hello world: 2
       hello world: 1
3 processes created
```

Сообщения от разных процессов выводятся с разными отступами.

□

10.1.3. Средства межпроцессной коммуникации

Основным средством межпроцессной коммуникации в Promela являются *каналы передачи сообщений*. Канал может буферизовать не более чем заданное число сообщений:

- процесс, пишущий в заполненный канал, блокируется до тех пор, пока другой процесс не осуществит чтение[1];
- процесс, читающий из пустого канала, блокируется до тех пор, пока другой процесс не осуществит запись.

Синтаксически каналы являются переменными типа **chan**; декларация начинается с ключевого слова **chan**, после которого указывается имя канала и опционально инициализатор:

chan *c [* = [*N*] **of** {T_1, ..., T_n} *]*;

Здесь *c* – имя канала, *N* – емкость (размер буфера), T_1, ..., T_n – типы компонентов (полей) передаваемых сообщений (каждое сообщение представляет собой кортеж $t_1, \ldots, t_n$, где t_i – значение типа T_i). Если $N = 0$, канал является *синхронным* (см. лекцию 9).

Если канал не инициализирован, он не может быть использован для передачи сообщений. Напомним: **chan** – специальный целочисленный тип. Инициализатор создает канал и возвращает его идентификатор (число от 1 до 255).

---

[1] 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].

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),
   по которому передаются сообщения типа byte */
chan e = [0] of {byte};

// канал емкости 16, по которому передаются идентификаторы каналов
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**(*c*) – число сообщений в буфере канала *c*;
- **empty**(*c*) – то же, что **len**(*c*) == 0;
- **nempty**(*c*) – то же, что **len**(*c*) > 0;
- **full**(*c*) – то же, что **len**(*c*) == *N*, где *N* – емкость канала *c*;
- **nfull**(*c*) – то же, что **len**(*c*) < *N*, где *N* – емкость канала *c*.

Запись в канал осуществляется с помощью конструкции

c!*m*,

где *c* – канал, а *m* – выражение (канал должен быть инициализирован, а тип выражения – совпадать с типом передаваемых по каналу сообщений).

Сообщение *m* добавляется в конец буфера канала *c* (если буфер заполнен, возникает блокировка). У синхронного канала буфера нет: процесс блокируется, пока сообщение не будет прочитано принимающей стороной.

Чтение из канала осуществляется с помощью одной из следующих конструкций:

c?*p* или *c*?<*p*>,

где *c* – канал, а *p* – *образец* (*pattern*) читаемого сообщения (что это такое, будет объяснено чуть позже):

- *c*?*p* – *прием с удалением сообщения из канала (channel read)*:
  - если буфер канала *c* не пуст и первое сообщение соответствует образцу *p*,
    - данные сообщения переносятся в переменные образца;
    - сообщение удаляется из буфера;

- в противном случае
  - процесс блокируется;

❍ *c*?<*p*> – *прием с сохранением сообщения в канале* (*channel poll*):
- то же самое, но сообщение из буфера не удаляется.

Для синхронных каналов вторая операция не определена, а первая работает иначе: сообщение берется не из буфера, а непосредственно от передающей стороны.

Что такое образец? Это кортеж, элементами которого могут быть значения, переменные и символы _. Например, если по каналу передаются пары чисел, допустимы образцы

`1,2` `1,x` `x,y` `x,_` `_,1` `_,_` и другие.

Говорят, что сообщение $m_1,\ldots,m_n$ *соответствует* образцу $p_1,\ldots,p_k$, если $n = k$ и для каждого $i \in \{1, \ldots, n\}$ справедливо следующее: если элемент p_i является значением (константой или результатом вызова функции **eval**), то m_i совпадает с ним. Например,

сообщение `1,2` соответствует образцам `1,y` `x,2` `x,y` `_,_`

Если в образец входят переменные, то при чтении из канала в них записываются значения соответствующих полей сообщения (предполагается, что каждая переменная входит в образец не более одного раза).

Пример 10.7

Приведем примеры операторов передачи и приема сообщений через каналы:

```
c!1          // посылка сообщения: 1 добавляется в конец буфера
             // (блокировка, если буфер полон)
c!ack,p      // посылка составного сообщения (кортежа)
c?_          // удаление из буфера первого сообщения
             // (блокировка, если буфер пуст)
c?x          // чтение первого сообщения в переменную x
             // (сообщение удаляется из буфера)
c?5          // блокировка, если буфер пуст или первое сообщение не 5
             // (если буфер не пуст и 5 - первое сообщение, оно удаляется)
c?eval(x)    // блокировка, если буфер пуст или первое сообщение
             // отлично от значения x в момент вызова функции eval
c?ack,p      // блокировка, если буфер пуст или первый элемент
             // первого сообщения (тип) отличен от ack;
             // чтение второго элемента (данных) в переменную p
             // (сообщение удаляется из буфера)
c?<ack,_>    // то же самое в плане блокировки, но без чтения из канала
c?<_>        // блокировка, если канал пуст
```

□

10.1.4. Управляющие операторы

Основной управляющей конструкцией языка Promela является *оператор выбора* **if**. В нем задается непустое множество *охраняемых действий* (*guarded actions*), т. е. пар вида *условие* → *действие*, из которых выбирается и исполняется только одно:

```
if
:: guard_1 -> action_1
...
:: guard_n -> action_n
fi
```

Более точно семантика оператора выбора определяется так:

1) в текущем состоянии (обозначим его *s*) проверяется *исполнимость* (*executability*) всех условий $guard_i$ и составляется множество *разрешенных* действий:

   $Enable = \{guard_i \rightarrow action_i \mid s \vDash guard_i\}$;

2) если $Enabled = \emptyset$, процесс блокируется (до тех пор, пока в множестве не появится хотя бы одно действие);
3) среди действий множества *Enabled недетерминированно* выбирается одно, и это действие исполняется[1].

В операторе **if** можно использовать специальные условия **else** и **timeout**:

- условие **else** может быть только в последнем действии: оно исполнимо, когда не исполнимы все остальные условия оператора;
- условие **timeout** применяется для отслеживания *тупиков* (*deadlocks*): оно становится исполнимым, когда все процессы блокируются.

Обратите внимание: в качестве условий можно использовать произвольные операторы, включая операторы присваивания и передачи сообщений. *Исполнимость* – это возможность исполнения оператора, т. е. *отсутствие блокировки*. Если оператором является выражение (что допустимо), исполнимость есть истинность (как в языке C, выражение считается истинным, если его значение отлично от нуля). Такие операторы, как **skip**, *x* = *t* и **printf**, исполнимы всегда.

Разделители -> и ; являются взаимозаменяемыми: в качестве условия рассматривается первый оператор последовательности (он может быть единственным).

---

[1] Заметим, что семантика оператора выбора в Promela отличается от семантики аналогичного оператора в языке охраняемых команд Дейкстры (guarded command language) [Dij75], на котором отчасти базируется Promela. В языке Дейкстры если все условия ложны, оператор выбора завершается.

Пример 10.8

Ниже приведены примеры операторов выбора и пояснения к ним.

```
if
/* A1 */ :: (a != b) -> ... // исполняется либо A1,
/* A2 */ :: else     -> ... // либо A2
fi

if
/* B1 */ :: (a >  b) -> ... // если a < b, возникает блокировка
/* B2 */ :: (a == b) -> ...
fi

if
/* C1 */ :: in?x     -> ... // C1 разрешено, если канал in не пуст
/* C2 */ :: out!y    -> ... // C2 разрешено, если канал out не полон
fi
```

□

Еще одной управляющей конструкцией является *оператор повторения* **do**. Синтаксически он похож на оператор выбора: разница лишь в том, что вместо обрамления **if**–**fi** используется **do**–**od**. Семантически это выбор, помещенный внутрь бесконечного цикла.

Для выхода из цикла можно использовать операторы **break** и **goto**:

- **break** передает управление на оператор, следующий за повторением;
- **goto** передает управление на указанную метку.

Пример 10.9

Использование оператора повторения проиллюстрируем на примере алгоритма Евклида.

```
proctype euclid(int a, b) {
  int x = a, y = b;

  do
  :: x > y -> x = x - y
  :: y > x -> y = y - x
  :: else -> break
  od

  printf("gcd(%d,%d)=%d\n", a, b, x)
}
```

□

В новой редакции Promela (Spin версии не ниже 6) для оператора повторения вида

```
T i = a;              // инициализация счетчика цикла
do                    // (T – целочисленный тип)
:: i <= b -> P; i++   // проверка, итерация, инкремент
:: else -> break      // выход из цикла
od
```

имеется более компактная и наглядная форма записи:

```
T i;
for (i : a..b) {  // диапазон значений счетчика
  P               // тело цикла
}
```

В Promela можно определять *атомарные последовательности операторов*:

atomic { op_1; ...; op_n }.

В идеале атомарная цепочка не должна «разрываться» (ее операторы не должны перемешиваться с операторами других процессов), однако если при исполнении цепочки происходит блокировка, управление получает другой процесс. Когда блокировка снимается и исполнение цепочки возобновляется, атомарность по возможности обеспечивается для оставшихся операторов [Hol03].

Пример 10.10

Используем **atomic**, чтобы определить двоичный семафор (mutex).

```
init:  // инициализация мьютекса
  count = 1
...
enter: // захват мьютекса (вход в критическую секцию)
  do
  :: atomic { count > 0 -> count-- }; break
  :: else // ожидание
  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** – позволяет задавать *утверждения* для конкретных точек модели. Оператор имеет следующий синтаксис:

assert(*B*),

где *B* – условие на состояние модели.

Если проверка модели выявляет, что условие может стать ложным (на некотором вычислении), сообщается об ошибке.

Пример 10.11

Ниже приведена простейшая модель, содержащая ошибочное вычисление.

```
init { assert(false) }
```

□

Для описания темпоральных свойств применяются **never**-*клаузы*. Как видно из названия, они описывают недопустимые вычисления модели. Если анализ выявляет, что модель может повести себя так, как описано в клаузе, выдается сообщение об ошибке.

never-клауза работает[1] *синхронно* с моделью (см. лекцию 12): она получает управление перед исполнением каждого действия и сразу после исполнения последнего (если вычисление конечно). Завершение клаузы говорит о распознавании ошибочного поведения.

Пример 10.12

Рассмотрим пару тривиальных примеров.

```
never A { true } // что бы модель ни делала, она ошибочна

never B { P@L }  // ошибка, если в начале работы модели
                 // процесс P находится в точке L
```

Обратите внимание: клауза может иметь имя (оно опционально).

□

Пример 10.13

Определим **never**-клаузу, проверяющую свойство **G***p* [Hol03]. В негативном варианте это свойство выглядит так: **never F**¬*p* – высказывание *p* никогда не становится ложным.

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

Как только условие *p* становится ложным, осуществляется выход из цикла: **never**-клауза завершается, что говорит об ошибке. Пока *p* истинно, цикл продолжается.

□

[1] Мы говорим о *работе*, хотя при проверке модели исполнения (в традиционном понимании) не происходит.

never-клаузы позволяют описывать не только ошибки, выявляемые за конечное число шагов (нарушения *свойств безопасности*), но и ошибки, обнаружить которые можно только на бесконечных вычислениях (нарушения *свойств живости*). Для этого используются метки с префиксом `accept`: если оператор с такой меткой может исполниться бесконечное число раз, фиксируется ошибка.

Те, кто знаком с автоматами Бюхи, наверняка узнали принятое в них условие допустимости ω-слова – бесконечной цепочки символов (см. лекцию 12). Действительно, **never**-клауза есть представление автомата Бюхи, распознающего ошибочное поведение.

Пример 10.14

В следующем примере проверяется свойство **GF***p*. Негативная формулировка имеет вид never **FG**¬*p* – высказывание *p* не становится ложным навсегда.

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

Здесь мы имеем два цикла, расположенных друг за другом. Каждый раз, когда условие *p* становится ложным, есть две возможности: выйти из первого цикла или остаться в нем. При выходе вычисления попадают во второй цикл, помеченный `accept`, который исполняется, пока *p* ложно. Если *p* становится ложным навсегда, в `never`-клаузе допустимо вычисление, проходящее бесконечное число раз через метку `accept`, что говорит об ошибке.

□

В новой редакции Promela (Spin версии не ниже 6) имеются средства спецификации требований с помощью формул LTL [Ben10][1]:

```
ltl name {φ},
```

где *name* – необязательное имя, а φ – формула, задаваемая следующей грамматикой:

φ ::= *p* | **true** | **false** | (φ) | !φ | φ && φ | φ || φ | φ -> φ | φ <-> φ | | []φ | <>φ | φ **U** φ | φ **W** φ | φ **V** φ |.

[1] Spin строит **never**-клаузу для отрицания LTL-формулы (алгоритм рассматривается в лекции 12).

Здесь p – логическое или целочисленное выражение.

Смысл указанных операций приводится в табл. 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*)
&&	конъюнкция
||	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.

Пример 10.15

Ниже приводится LTL-спецификация алгоритма Петерсона взаимного исключения двух процессов (см. лекцию 9). Реализация самого алгоритма оставлена в качестве упражнения.

```
ltl {
  [](P[0]@CRS && P[1]@CRS)  && // взаимное исключение
  [](P[0]@SET -> <>P[0]@CRS) && // живость (процесс 0)
  [](P[1]@SET -> <>P[1]@CRS)    // живость (процесс 1)
}
```

□

10.2. Введение в инструмент Spin

Для анализа Promela-моделей можно использовать инструмент Spin (Simple Promela Interpreter). Он был разработан в 1980-х годах как средство проектирования и верификации компьютерных протоколов [Hol91]. С 1991 года инструмент распространяется бесплатно вместе с исходным кодом. В 2001 году за разработку Spin Джерард Хольцман был удостоен ACM Software System Award.

Нас интересуют функции Spin, связанные с *проверкой моделей* (*model checking*). Слово «модель» здесь допускает двоякую трактовку: *инженерную* (модель как Promela-описание системы) и *логическую* (модель как интерпретация

формулы, в которой эта формула истинна). Под проверкой модели понимается процедура, выявляющая, является ли заданная модель (в инженерном понимании) «моделью» (в логическом понимании), т. е. истинны ли заданные спецификации на модели (на каждом ее вычислении). Теоретические основы метода проверки моделей рассматриваются в следующих лекциях.

Одна из причин популярности Spin – эффективность. За годы существования инструмента в нем были воплощены многие оптимизации: *редукция частичных порядков* (*partial order reduction*), *сжатие битового представления состояний*, *хеширование без разрешения коллизий* и многие другие [Hol03]. Кроме того, чтобы обеспечить высокую скорость работы, для каждой задачи анализа (проверки заданных свойств на заданной модели) генерируется программа на языке C, решающая именно эту задачу.

10.2.1. Функции инструмента

Инструмент Spin содержит следующие компоненты [Hol03]:

- Promela-парсер;
- симулятор моделей;
- генератор программы проверки;
- графическая оболочка.

Типичный сценарий использования Spin выглядит так:

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, когда 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

Рассмотрим некоторые опции инструмента Spin в том порядке, в котором они обычно используются при верификации моделей [Hol03].

Рандомизированная симуляция модели:

```
spin [-p] [-uчисло-шагов] имя-файла
```

Если указана опция `-p`, выводится детальная информация по каждому шагу исполнения; иначе – только пользовательский вывод (**`printf`**).

Опция `-u` ограничивает время симуляции (например, `-u100` ограничивает исполнение 100 шагами).

Генерация, компиляция и запуск программы проверки:

```
spin -a имя-файла
gcc -o pan pan.c
./pan опции
```

Первая команда генерирует C-программу `pan.c`, осуществляющую проверку модели; вторая компилирует ее (это можно сделать любым компилятором); третья запускает.

Программа проверки имеет обширный набор опций. Вот некоторые из них:

- `-a` – включает поиск допускающих циклов (нужно для проверки свойств живости);
- `-f` – включает предположение о справедливости планировщика;
- `-i` – включает поиск кратчайшего контрпримера;
- `-m`*глубина-поиска* – устанавливает максимальную глубину поиска в пространстве состояний модели (по умолчанию 10 000);
- `-N` *имя-клаузы* – задает имя проверяемой **`never`**- или **`ltl`**-клаузы (обрабатывается только одна клауза).

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 НОД двух натуральных чисел. С помощью оператора **`assert`** специфицируйте пред- и постусловия. Используя симулятор Spin, протестируйте модель на некоторых значениях аргументов.
2. Гарантирует ли приведенный ниже монитор проверку свойства *invariant* во всех состояниях модели в режиме симуляции?

```
active proctype monitor() {
  do :: assert(invariant) od
}
```

3. 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 Spin.

4. 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 }
```

5. Реализуйте на языке Promela алгоритм Петерсона (см. лекцию 9), задайте условия корректности, используя **ltl**-клаузы, и проверьте корректность алгоритма с помощью Spin.
6. Задайте условия корректности алгоритма Петерсона, используя **never**-клаузы.
7. Определите **never**-клаузы, распознающие ошибочное поведение, описываемое следующими формулами LTL:
   a) <>*p* (см. лекцию);
   b) []*p*;
   c) <>[]*p* (см. лекцию);
   d) []<>*p*.
8. 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 переходит к обработке запроса, только когда ответит на предыдущий.

Задайте требования к системе, используя оператор **`assert`**, и проверьте корректность модели с помощью Spin (для каких-нибудь фиксированных значений N и M).

9. Для предыдущего задания опишите с помощью формулы LTL свойство живости системы: если клиент посылает запрос, рано или поздно он получит ответ.

   Проверьте это свойство с помощью Spin.

10. 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 тарелок 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.
11. Spin можно использовать для решения логических головоломок. Пусть требуется найти последовательность действий, приводящих некоторую «систему» из заданного начального состояния в заданное конечное состояние (пусть это конечное состояние описывается формулой φ). Поведение системы моделируется процессом, где в бесконечном цикле недетерминированно выбирается одно из допустимых действий. Чтобы «заставить» инструмент искать решение, в качестве проверяемого свойства берется отрицание условия достижения конечного состояния:

    $\neg \mathbf{F}\varphi$ или $\neg(\mathit{safe}\ \mathbf{U}\ \varphi)$,

    где *safe* – инвариант состояния системы (он либо учитывается при выборе действия, либо присутствует в свойстве). Решение головоломки есть контрпример для проверяемого свойства.

    Решите с помощью Spin головоломку «Волк, коза и капуста». Крестьянину нужно перевезти через реку волка, козу и капусту. Лодка небольшая – в ней может поместиться крестьянин, а с ним либо волк, либо коза, либо капуста. Если оставить волка с козой на берегу, волк съест козу; если оставить козу с капустой, коза съест капусту. Как крестьянину перевезти свой груз?

    Для поиска кратчайшего решения используйте опцию `-i` программы проверки (`pan`).

12. Формализуйте операционную семантику языка Promela.

Лекция 11

Моделирование программ и предикатная абстракция

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?

А. Пуанкаре.
Наука и гипотеза

Все обобщения опасны, даже это.

А. Дюма-сын

Рассматриваются вопросы моделирования программ структурами Крипке, в том числе связанные с выбором уровня абстракции данных и гранулярности действий. Обсуждаются возможные аномалии: появление в модели поведения, не присущего 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 программы.

11.1. Представление программ структурами Крипке

Как мы знаем, формулы LTL интерпретируются на бесконечных траекториях – последовательностях пометок, каждая из которых есть множество (истинных) элементарных высказываний (см. лекцию 9). Чтобы иметь возможность проверять LTL-свойства на вычислениях программ, т. е. последовательностях состояний (см. лекцию 3), поступают следующим образом: вводят предикаты над переменными и оценивают их истинность на состояниях; если предикат истинен, соответствующий символ добавляется в пометку. Скажем, если в программе есть переменные x и y, можно определить предикат $p(x, y) \equiv (y > x)$ и добавить p в пометки всех тех состояний, для которых $y > x$. Это позволит проверять (конечно, на конкретных вычислениях) такие свойства, как, например, $\mathbf{G}\neg p$.

Для формальной верификации интерес представляют не одиночные траектории (и тем более не их конечные префиксы), а множество *всех возможных* траекторий. Суть моделирования в контексте верификации программ (software model checking) состоит в построении (конечной) структуры Крипке, аппроксимирующей поведение (множество траекторий) верифицируемой программы. Метод грубой силы – исследовать пространство состояний программы и пометить каждое состояние множеством истинных в нем высказываний – не осуществим на практике: программа может иметь огромное или даже бесконечное число состояний. Требуется *абстракция* – игнорирование тех или иных деталей.

Следующие разделы рассматривают кратко вопросы абстракции данных и действий, а также вопросы адекватности моделирования.

11.1.1. Обобщенные состояния (абстракция данных)

Состояние программы, или конфигурация (см. лекции 2, 5 и 9), – естественная концепция программирования и теории вычислений. Это мгновенный снимок (snapshot) исполнения программы, включающий *состояние данных* (значения переменных) и *состояние управления* (точки исполнения процессов). Пренебрегая некоторыми деталями, состояния можно объединять в классы эквивалентности, называемые *абстрактными* (*обобщенными*) *состояниями*, или *фактор-состояниями*. Формализуем эту идею применительно к структурам Крипке (размеченным системам переходов).

Пусть заданы множество элементарных высказываний AP и структура Крипке $\langle S, S_0, R, L\rangle$, которую мы будем называть *конкретной системой переходов*.

Функцией абстракции называется любое отображение $f\colon S \to \hat{S}$, где $\hat{S}$ – некоторое непустое множество, такое что для всех $s, s' \in S$ имеет место импликация

$$(f(s) = f(s')) \to (L(s) = L(s')).$$

Проще говоря, состояния могут быть объединены в классы эквивалентности, только если они имеют одинаковые пометки (совпадение пометок, однако, не является поводом для объединения состояний).

Абстрактной системой переходов, порожденной функцией абстракции f, называется система $\langle \hat{S}, \hat{S}_0, \hat{R}, \hat{L} \rangle$, где [Bai08]

- $\hat{S}_0 = \{f(s) \mid s \in S_0\}$;
- $\hat{R} = \{(f(s), f(s')) \mid (s, s') \in R\}$;
- $\hat{L}(f(s)) = L(s)$ для всех $s \in S$.

Пример 11.1

При моделировании коммуникационных протоколов можно абстрагироваться от содержимого передаваемых сообщений. Как правило, логика протокола определяется типами сообщений, их порядковыми номерами, но не данными. Удалив из структуры сообщения информационные поля (см. табл. 11.1), можно значительно сократить число состояний модели, сохранив при этом ее адекватность (что понимается под «адекватностью», мы объясним чуть позже).

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 */
	}

□

Важно отметить, что абстракция может приводить к появлению траекторий, не присущих исходной модели (программе). Предположим, что два состояния s_1 и s_2, имеющие одинаковые пометки, «объединяются» в одно абстрактное состояние:

$$L(s_1) = L(s_2) \quad \text{и} \quad f(s_1) = f(s_2).$$

Здесь f – некоторая функция абстракции (см. рис. 11.1).

Пусть в исходной модели имеются следующие пары переходов: $r_1 \to s_1 \to t_1$ и $r_2 \to s_2 \to t_2$ (предположим, что это единственные переходы из r_1, r_2, s_1 и s_2), при этом

$$L(r_1) = a \neq b = L(r_2), \quad L(s_1) = c = L(s_2), \quad L(t_1) = d \neq e = L(t_2).$$

Тогда в абстрактной модели появляются траектории $\ldots ace \ldots$ и $\ldots bcd \ldots$, отсутствующие в исходной модели.

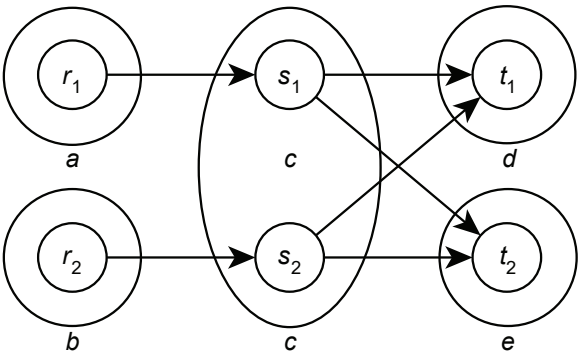


Рис. 11.1 ❖ Возникновение новых траекторий при абстракции

Лишние траектории в абстрактных моделях могут приводить к *ложным сообщениям об ошибках* (*false positives*), или, как их называют в математической статистике, *ошибкам первого рода*[1]. Для каждого найденного примера ошибочного поведения, так называемого *контрпримера*, нужно проверить, реализуем ли он в исходной модели: если нет, это не ошибка[2]. 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).

На рис. 11.2 показаны ситуации, возникающие при абстракции моделей. Тонкие линии соответствуют траекториям системы: попадание хотя бы одной из них в серую область является ошибкой (нарушением свойства безопасности). Жирные линии отражают метод абстракции: это своего рода трубка, аппроксимирующая множество всех возможных траекторий. Сообщение об ошибке выдается, когда эта трубка выходит за пределы белой области: если при этом все траектории лежат внутри белой области, имеет место ложная тревога (график в центре); в противном случае это обычная ошибка (такая ситуация не показана). Если трубка «дырявая», отдельные траектории могут «просочиться» в серую область, даже если трубка целиком находится в белой области, – это пропуск ошибки (график справа). Идеальная ситуация – трубка без дыр внутри зоны безопасности (график слева).

---

[1] Ошибки первого и второго рода – ключевые понятия проверки статистических гипотез. Ошибка первого рода – H_0 (нулевая гипотеза, гипотеза по умолчанию) неверно отвергнута; ошибка второго рода – H_0 неверно принята. В нашем случае в роли H_0 выступает утверждение «в программе нет ошибок» (что, конечно, не согласуется с аксиоматикой Шуры-Буры – см. лекцию 1).

[2] Здесь нужно быть аккуратным: нереализуемость конкретного контрпримера еще не означает, что ошибки нет, – возможно, существует другой контрпример (не выданный по какой-то причине инструментом верификации), который окажется реализуемым.

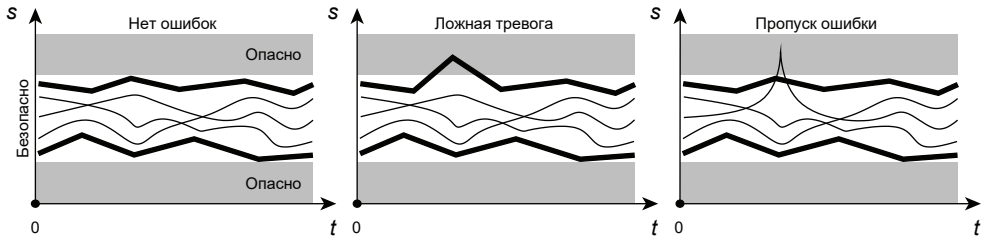


Рис. 11.2 ❖ Абстракция модели: ошибки первого и второго рода

Обратите внимание: при абстракции, основанной на факторизации состояний, пропуски ошибок исключены.

Пример 11.2

В табл. 11.2 приведен фрагмент кода на языке Promela (слева) и результат его абстракции (справа). В абстрактной модели игнорируется переменная *x*: присваивание в *x* удалено, а условия с участием *x* заменены на булевы переменные *p*0 и *p*1, которые могут принимать произвольные значения истинности.

Таблица 11.2. Абстрагирование от переменной x (появление ошибочных вычислений)

Исходная модель	Результат абстракции
`x = f(...);`	`select(p0 : 0 .. 1);` `select(p1 : 0 .. 1);`

`if`
`:: (x % 3 == 0) -> r = 0`
`:: (x % 3 == 1) -> r = 1`
`:: else -> r = 2`
`fi;`
	`if`
	`:: p0 -> r = 0`
	`:: p1 -> r = 1`
	`:: else -> r = 2`
	`fi;`

`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 такая абстрактная модель допускает ошибочные вычисления, невозможные в исходной модели, например:

$r = 0$; **assert**($r == 1$).

Это связано с тем, что в модели не учитывается семантика условий: *p*0 и *p*1 могут одновременно получить значение *true*, что соответствует невыполнимой формуле

$(x \,\%\, 3 == 0)\ \&\&\ (x \,\%\, 3 == 1)$.

Такой ситуации можно избежать, если уточнить модель – исключить при выборе значений истинности *p*0 и *p*1 заведомо недостижимые комбинации:

```
select(p0 : 0 .. 1);        // p0 принимает произвольное значение
select(p1 : 0 .. (1 - p0)); // если p0 = 1, то p1 = 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 (частично или даже полностью), построив при этом адекватную модель со сравнительно небольшим числом состояний.

Более тонкие аспекты абстракции данных рассматриваются в разделе, посвященном предикатной абстракции.

11.1.2. Обобщенные переходы (гранулярность действий)

Понятие состояния не отделимо от понятия перехода – элементарного изменения состояния. Вопрос в том, как переходы модели (атомарные по определению) соотносятся с реальным поведением программы (состоящим, вообще говоря, из множества шагов). Моделирование длительных процессов мгновенными действиями может приводить к пропуску ошибок. С другой стороны, излишняя детализация также вредна: если неделимая операция 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; операция 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-го процесса, в состоянии $x = 0$ может привести к любому из n вариантов: $x = 1, \ldots, x = n$ (см. табл. 11.3).

Таблица 11.3. Варианты исполнения n параллельных присваиваний x := x + 1

Вариант исполнения 1			**Вариант исполнения** n	
$P_{i_1}: t_{i_1} := x$	$t_{i_1} = 0$		$P_{i_1}: x := t_{i_1} + 1$	$t_{i_1} = 0$
оставшиеся процессы, исполняемые в произвольном порядке	…	…	…	$x = 0 + 1 = 1$ … $t_{i_n} = n - 1$
$P_{i_1}: x := t_{i_1} + 1$	$x = 0 + 1 = 1$		$P_{i_n}: x := t_{i_n} + 1$	$x = (n - 1) + 1 = n$

□

11.2. Модели последовательных и параллельных программ

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 программы P на том или ином уровне абстракции и позволяющая выразить свойство φ. В конце раздела мы коснемся вопроса построения модели параллельной программы по моделям составляющих ее процессов.

11.2.1. Целочисленное деление

Начнем с нетипичного для проверки моделей примера. Это вычислительная программа, а именно программа 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 фиксированных $a \geq 0$ и $b > 0$ число состояний конечно. Например, для $a = 5$ и $b = 2$ имеются 4 состояния (включая начальное состояние с нулевыми значениями q и r). Отношение переходов для этого случая показано на рис. 11.3: начальное состояние находится слева; петля в состоянии справа добавлена для того, чтобы сделать отношение полным (см. лекцию 9).

Введем элементарные высказывания:

- $p_1 \equiv (a = b \cdot q + r)$;
- $p_2 \equiv (r \geq 0)$;
- $p_3 \equiv (r < b)$.

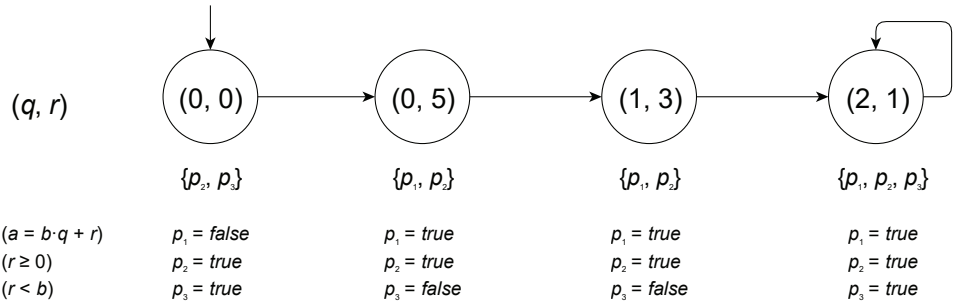


Рис. 11.3 ❖ Структура Крипке, моделирующая программу целочисленного деления для заданных значений входных параметров: $a = 5$ и $b = 2$

Сопоставив состояниям программы значения истинности этих высказываний, получим структуру Крипке (см. рис. 11.3). Поскольку программа детерминирована, структура имеет единственное вычисление (обозначим его π). На нем можно интерпретировать формулы LTL над множеством $AP = \{p_1, p_2, p_3\}$, например следующие:

- $\pi \vDash \mathbf{F}\{p_1 \wedge p_2 \wedge p_3\}$ – когда-нибудь будет достигнуто состояние, в котором переменная q содержит частное от деления q на b, а r – остаток (условие корректности программы);
- $\pi \vDash \mathbf{FG}\{p_1 \wedge p_2 \wedge p_3\}$ – условие корректности, усиленное свойством *стабилизации*;
- $\pi \vDash \mathbf{XG}\{p_1 \wedge p_2\}$ – всегда (за исключением начального состояния) истинен инвариант цикла $(a = q \cdot b + r) \wedge (r \geq 0)$.

Приведенная в примере структура Крипке построена для конкретных значений параметров a и b, поэтому не пригодна для верификации программы в целом. Для верификации *параметризованных моделей* в общем виде (для всех значений параметров) разработаны специальные методы (*parameterized model checking*) [Mcm01].

Например, для программы целочисленного деления можно построить обобщенную систему переходов, объединив промежуточные состояния вычисления в одно (состояние *loop* на рис. 11.4). Для этого нужно доказать, что формула $p_1 \wedge p_2$ является инвариантом цикла. Обратите внимание: на обобщенной структуре свойство $\mathbf{F}\{p_1 \wedge p_2 \wedge p_3\}$ нарушается (это false positive).

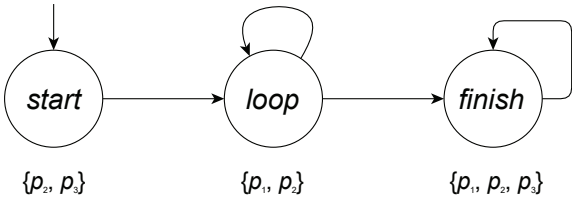


Рис. 11.4 ❖ Обобщенная система переходов программы целочисленного деления

Во избежание недоразумений отметим, что приведенные выше структуры Крипке имеют мало общего с тем, как рассматриваемый код представляется в инструменте Spin.

11.2.2. Алгоритм Петерсона

Рассмотрим алгоритм Петерсона 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 (значения переменных), то на этот раз нас будут интересовать состояния управления (метки операторов). Система переходов одного процесса приведена на рис. 11.5. 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.

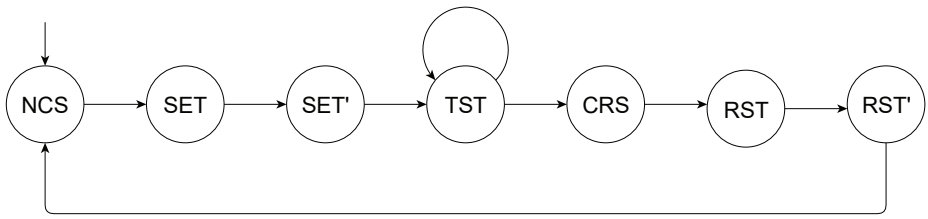


Рис. 11.5 ❖ Система переходов одного процесса в алгоритме Петерсона

Чтобы проверить свойство взаимного исключения $\mathbf{G}\{\neg(p_0 \wedge p_1)\}$, нужно построить модель параллельной программы – *асинхронную композицию* моделей процессов. Состояниями модели программы являются комбинации состояний моделей процессов, а каждый переход соответствует переходу в модели одного из процессов (см. лекцию 9).

Для краткости мы иногда будем опускать слово «модель», понимая под программой или процессом соответствующую модель.

Асинхронная композиция независимых процессов включает все возможные комбинации их состояний; если же между процессами есть зависимости (обычно это так), то некоторые комбинации 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. 11.6 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.

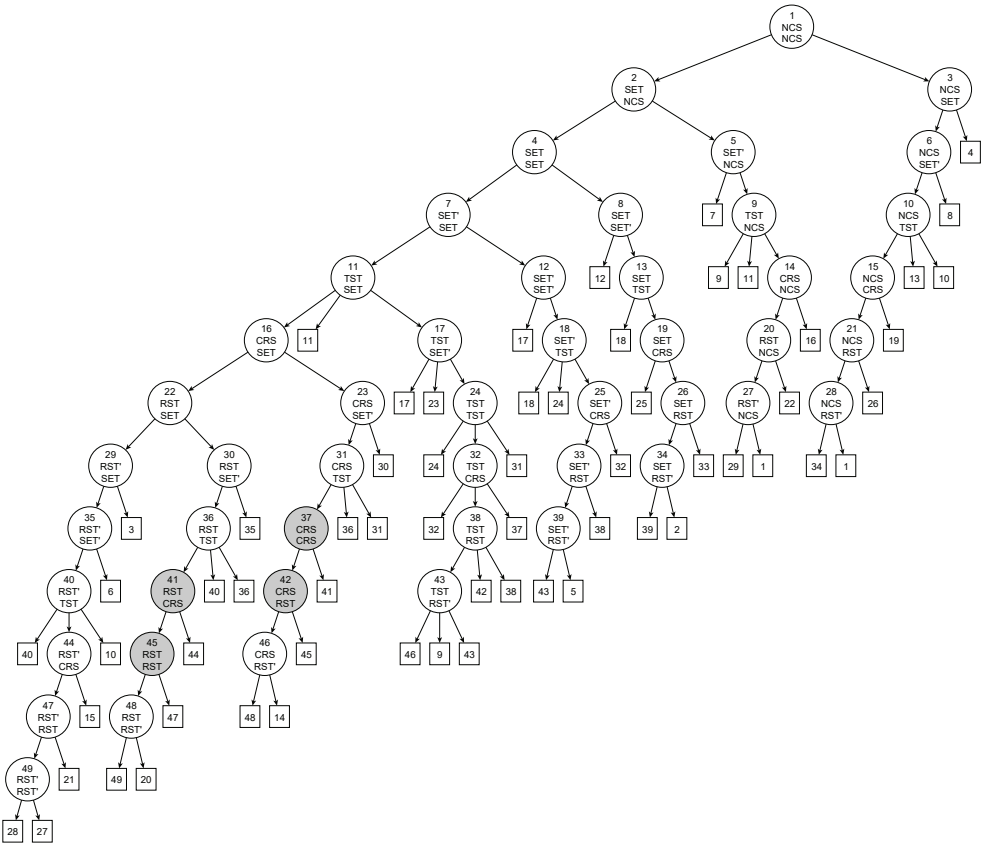


Рис. 11.6 ❖ Асинхронная композиция систем переходов для алгоритма Петерсона

11.2.3. Построение асинхронной композиции систем переходов

Существует два базовых подхода к *исследованию пространства состояний* программы (*state space exploration*), в частности к построению асинхронной композиции процессов:

- *поиск в ширину* (*BFS*, *Breadth-First Search*);
- *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 (исключая состояния 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 табл. 11.4 представлено более формальное описание методов исследования пространства состояний. Для простоты здесь предполагается, что есть только одно начальное состояние (s_0). Множество S содержит посещенные состояния: в начале $S = \{s_0\}$, в конце – множество всех достижимых состояний. В описании используются следующие вспомогательные функции:

- *succ*(s) – множество непосредственных последователей состояния s;
- *choose*(X) – произвольный элемент непустого множества X.

Таблица 11.4. Исследование пространства состояний: поиск в ширину и поиск в глубину

Поиск в ширину	Поиск в глубину
S := {s_0}; *queue* := {s_0}; **while** *queue* ≠ ∅ **do** s := *queue.pop*(); // действие над s (проверка) **for** s' **in** (*succ*(s) \ S) **do** *queue.push*(s') **end**; S := S ∪ *succ*(s) end	S := {s_0}; *stack* := {s_0}; **while** *stack* ≠ ∅ **do** s := *stack.top*(); **if** (*succ*(s) \ S) = ∅ **then** // действие над s (проверка) *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

нарушается в состоянии, недалеко отстоящем 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 следующее (смежное) состояние. По этой причине поиск в глубину широко применяется в *тестировании на основе моделей* (*model-based testing*) – реальную систему, в отличие от модели, трудно перевести в произвольное состояние [Бур03a], [Бур03b] (см. лекцию 16).

11.3. Предикатная абстракция и уточнение абстракции по контрпримерам

Вернемся к построению моделей (как еще говорят, *абстракций*) 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 этой темой рекомендуем обзор [Ман13], содержащий описание метода и обширную библиографию.

Метод CEGAR применяется для установления достижимости заданной точки программы (назовем ее *error*). Идея состоит в следующем. Проверяется абстрактная модель программы. Если анализ выявляет, что модель корректна (точка *error* недостижима), то корректна и исходная программа (ошибки второго рода исключены); в противном случае для модели строится *контрпример* – путь в точку *error*. Если этот путь реализуем в исходной программе, программа некорректна; иначе негативный результат – всего лишь ложная тревога, вызванная абстрагированием от чего-то существенного (от чего именно, выявляется путем анализа контрпримера). Модель уточняется, и к уточненной модели применяется тот же алгоритм. В результате ряда итераций метод CEGAR либо найдет путь в точку *error*, либо докажет ее недостижимость, либо 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.

Пусть p – предикат, заданный на множестве состояний S. Обозначим через $[\![p]\!]$ множество состояний, в которых он истинен:

$$[\![p]\!] = \{s \in S \mid s \vDash p\}.$$

Введем специальные предикаты *false* и *true*:

$[\![false]\!] = \emptyset$ и $[\![true]\!] = S$.

Для заданного конечного множества предикатов $\{p_1, \ldots, p_n\}$ *регионом* называется либо *false*, либо формула вида

$$\bigwedge_{k=1}^{m} p_{i_k}, \quad \text{где } 1 \le i_1 < \cdots < i_m \le n.$$

При $m = 0$ получаем пустую конъюнкцию, т. е. *true*.

Пример 11.4

Выпишем все возможные регионы для предикатов $(x \ge 0)$ и $(x \le 0)$:

- $r_1 \equiv true$;
- $r_2 \equiv (x \ge 0)$;
- $r_3 \equiv (x \le 0)$;
- $r_4 \equiv (x \ge 0) \wedge (x \le 0) \equiv (x = 0)$;
- $r_5 \equiv false$.

□

Метод абстракции, состоящий в «замене» конкретных состояний программы на регионы, называется (*декартовой*) *предикатной абстракцией*. Соответственно, вместо конкретных конфигураций, имеющих вид $\langle l, s\rangle$, где l – состояние управления, а s – состояние данных, при верификации программы рассматриваются *абстрактные конфигурации* – пары вида $\langle l, r\rangle$, где l – состояние управления (оно остается без изменений), а r – регион.

Граф, состояниями которого являются абстрактные конфигурации программы, а дугами – переходы между ними (при исполнении операторов), будем называть *абстрактной системой переходов*.

Для заданной программы P рассмотрим произвольный оператор op. Зафиксируем регион r и определим регион r', в который перейдет r под действием op:

для всех $s \in S$ и $s' \in M[\![op]\!](s)$ из того, что $s \in [\![r]\!]$, следует, что $s' \in [\![r']\!]$.

Пусть $sp(op, r)$ – сильнейшее постусловие оператора op относительно предусловия r (см. лекцию 2). Тогда r' – подходящий регион, если (и только если) общезначима импликация

$$sp(op, r) \to r'$$

Очевидно, что этому условию заведомо удовлетворяет регион *true*, однако интерес представляет «наименьший» из регионов, содержащих в себе множество $M[\![op]\!](s)$.

Чтобы построить этот регион, воспользуемся тождествами

$$\left(p \to \bigwedge_{i=1}^{m} p_{i_k}\right) \equiv \left(\neg p \vee \bigwedge_{i=1}^{m} p_{i_k}\right) \equiv \left(\bigwedge_{i=1}^{m} (\neg p \vee p_{i_k})\right);$$
$$\neg\left(p \to \bigwedge_{i=1}^{m} p_{i_k}\right) \equiv \left(\bigvee_{i=1}^{m} (p \wedge \neg p_{i_k})\right).$$

Таким образом, формула $sp(op, r) \to \bigwedge_{k=1}^{m} p_{i_k}$ общезначима тогда и только тогда, когда невыполнима ни одна из конъюнкций $sp(op, r) \wedge \neg p_{i_k}$.

Исходя из вышесказанного, регион r' конструируется следующим образом:

- если формула $sp(op, r)$ невыполнима, то $r' = false$;
- в противном случае r' имеет вид конъюнкции (возможно, пустой) и строится так:
  - для всех $i \in \{1, \ldots, n\}$:
    - если формула $sp(op, r) \wedge \neg p_i$ невыполнима,
    - предикат p_i добавляется в конъюнкцию.

Напомним: построение сильнейшего постусловия базируется на *SSA-представлении* программы (*Static Single Assignment*), где в каждую переменную может быть не более одного присваивания [Aho06]. В SSA-представлении используются так называемые *версии* переменных (номер версии инкрементируется при присваивании).

Сильнейшее постусловие программы P (не содержащей циклов) относительно предусловия φ определяется соотношением, показанным ниже:

$$sp(P, \varphi) = (\varphi[оригиналы := версии_н] \wedge ssa(P))[версии_к := оригиналы].$$

Здесь $ssa(P)$ – SSA-представление программы P (рассматриваемое как логическая формула), *оригиналы* – переменные программы P, $версии_н$ и $версии_к$ – соответственно начальные и конечные версии переменных.

Обратите внимание: множество регионов образуют полное частично упорядоченное множество, а описанное правило вычисления регионов обладает свойством монотонности (при расширении начального региона конечный регион может только расшириться). Таким образом, предикатная абстракция есть разновидность абстрактной интерпретации (см. лекцию 7).

11.3.2. Уточнение абстракции по контрпримерам (CEGAR)

Сразу выбрать подходящий уровень абстракции модели непросто. 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

этого метода применительно к предикатной абстракции. Напомним: мы проверяем недостижимость точки *error*.

0. Текущее множество предикатов полагается пустым:
   a) имеем два региона: *true* и *false*.
1. Строится абстрактная система переходов для текущего множества предикатов.
2. Проверяется корректность абстрактной модели:
   a) если модель корректна, то 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 $sp(op(\pi), true)$[1], где $op(\pi)$ – цепочка операторов, соответствующая вычислению π.
4. Проверяется выполнимость формулы $sp(op(\pi), true)$:
   a) если формула выполнима, то исходная программа некорректна и π – контрпример;
   b) иначе исполняется шаг 5.
5. Уточняется текущее множество предикатов: в множество добавляются предикаты, построенные на основе цепочки операторов $op(\pi)$. Переход к шагу 1.

Поясним, как уточняется множество предикатов, по которым строится модель. Для логики первого порядка известен следующий примечательный результат, известный как интерполяционная теорема Крейга[2] [Cra57]:

Если общезначима импликация $\varphi \to \psi$, то существует формула ρ, такая что

- общезначимы импликации $\varphi \to \rho$ и $\rho \to \psi$;
- все нелогические символы ρ (свободные переменные, а также неинтерпретируемые функциональные и предикатные символы) являются общими для φ и ψ.

Формула ρ называется *интерполянтом Крейга*, или просто *интерполянтом*.

Теорему Крейга можно переформулировать так: если невыполнима конъюнкция $\varphi \wedge \psi$, то существует формула ρ (все нелогические символы которой являются общими для φ и ψ), такая что общезначима импликация $\varphi \to \rho$ и невыполнима конъюнкция $\rho \wedge \psi$. Грубо говоря, ρ в общих для φ и ψ терминах поясняет, в чем суть противоречия между ними.

Пример 11.5

Пусть $\varphi \equiv (f(a) = b \wedge p(f(a)))$ и $\psi \equiv ((b = c) \to p(c))$ [Rüm].

---

[1] Здесь в качестве предусловия используется *true*, поскольку считается, что начальное состояние может быть любым. Если начальное состояние известно, берется конъюнкция всех тех предикатов, которые в нем истинны.

[2] Вильям Крейг (William Craig; 1918–2016) – американский философ; известен двумя фундаментальными результатами в области логики: теоремой (ре)аксиоматизации (1953) и интерполяционной теоремой (1957).

Нетрудно проверить, что формула $p(b)$ является интерполянтом φ и ψ:

$(f(a) = b \wedge p(f(a))) \rightarrow p(b)$ и $p(b) \rightarrow ((b = c) \rightarrow p(c))$.

□

Мы не рассматриваем здесь процедуру построения интерполянта для логики первого порядка; желающие могут обратиться, например, к книге [Har09]. Для логики высказываний интерполянт можно построить с помощью следующей рекурсивной процедуры ($atoms(\chi)$ обозначает множество элементарных высказываний, входящих в формулу χ)[1]:

```
Функция Interpolate
Вход: формулы φ и ψ, такие что φ → ψ
Выход: интерполянт ρ формул φ и ψ

if (atoms(φ) \ atoms(ψ)) = ∅ then // все высказывания φ входят в ψ
  return φ
else
  p := choose(atoms(φ) \ atoms(ψ));
  φ' := φ[p := true] ∨ φ[p := false];
  return Interpolate(φ', ψ)
end
```

Корректность этой процедуры следует из общезначимости импликаций $\varphi \rightarrow \varphi'$ и $\varphi' \rightarrow \psi$, где $\varphi' \equiv \varphi[p := true] \vee \varphi[p := false]$, а p – произвольный атом из $atoms(\varphi) \setminus atoms(\psi)$.

Для удобства далее под $sp(P, \varphi)$ понимается сильнейшее постусловие P относительно φ, выраженное в терминах версий переменных, т. е.

$\varphi[оригиналы := версии_{н}] \wedge ssa(P)$.

Пусть $op(\pi) = \{op_i\}_{i=1}^{n}$, где π – контрпример модели, но не исходной программы (шаг 4.b). В этом случае формула $sp(op(\pi), true)$ невыполнима. 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 конъюнкции $(\psi_1 \wedge \ldots \wedge \psi_n)$ равносильна общезначимости импликации

$\psi_1 \rightarrow \neg(\psi_2 \wedge \ldots \wedge \psi_n)$.

По теореме Крейга 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 част-

[1] Интерполянт можно также построить путем резолютивного опровержения формулы $\varphi \wedge \neg\psi$, отрицания $\varphi \rightarrow \psi$ (см. лекцию 8) [Rüm]. Пусть φ и $\neg\psi$ представлены в клаузальной форме. Каждой клаузе C сопоставляется *частичный интерполянт* $\rho(C)$: если C – клауза φ, $\rho(C) \equiv g(C)$, где $g(C)$ – подклауза C (возможно, пустая), содержащая все литералы C с общими переменными φ и ψ (и только их); иначе $\rho(C) \equiv true$. При построении опровержения: если резолюция C_1 и C_2 осуществляется по внутренней переменной φ, то $\rho(Res(C_1, C_2)) = \rho(C_1) \vee \rho(C_2)$; иначе $\rho(Res(C_1, C_2)) = \rho(C_1) \wedge \rho(C_2)$. Искомый интерполянт есть $\rho(\square)$.

ности, это означает, что в ρ_1 входит не более одной версии каждой переменной. Обозначим через p'_1 предикат, полученный из ρ_1 заменой версий на оригинальные переменные. Таким образом,

$\rho_1 = p'_1\theta_1$, где θ_1 – подстановка вида [оригиналы := версии].

Нетрудно показать, что если множество предикатов, по которым строится абстракция, содержит p'_1, то p'_1 содержится в регионе r'_1, в который переходит регион $r'_0 = true$ 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, общезначимость импликации $\psi_1 \rightarrow \rho_1$, учитывая, что $\psi_1 = sp(op_1, true)$ и $\rho_1 = p'_1\theta_1$, эквивалентна невыполнимости

$sp(op_1, true) \wedge \neg p'_1\theta_1$.

В свою очередь, это означает, что p'_1 входит в регион r'_1 (см. правило вычисления регионов).

Рассмотрим оставшиеся операторы: $op(\pi^{(2)}) = \{op_i\}_{i=2}^{n}$. Условие $sp(op(\pi^{(2)}), r'_1)$ невыполнимо, поскольку

$sp(op(\pi^{(2)}), r'_1) = r'_1\theta_1 \wedge sp(op(\pi^{(2)}), true) =$

$(p'_1 \wedge r_1)\theta_1 \wedge (\psi_2 \wedge \ldots \wedge \psi_n) = \rho_1 \wedge (\psi_2 \wedge \ldots \wedge \psi_n) \wedge r_1\theta_1$,

а конъюнкция $\rho_1 \wedge (\psi_2 \wedge \ldots \wedge \psi_n)$, как утверждает теорема Крейга (в альтернативной формулировке), невыполнима.

Применив к $sp(op(\pi^{(2)}), r'_1)$ 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 отличен от *false*, процесс продолжается.

Итак, в цикле ($k = 1, 2, \ldots$) для каждой формулы

$sp(op(\pi^{(k)}), r'_{k-1}) = r'_{k-1}\theta_{k-1} \wedge (\psi_k \wedge \ldots \wedge \psi_n)$

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 $sp(op(\pi^{(n)}), r'_{n-1}) = sp(op_n, r'_{n-1})$ невыполнима (как и все предшествующие формулы этого вида). По правилу вычисления региона (тривиальный случай) имеем $r'_n = false$.

Уточнение множества предикатов состоит в добавлении в него построенных предикатов $p'_1, \ldots, p'_k$. Для уточненной абстракции вычисление π уже не является контрпримером (собственно, это было показано выше).

На практике, например в инструменте CPAchecker [CPA], процесс уточнения абстракции отличается от описанного выше [Ман13]. Для противоречивой конъюнкции $(\psi_1 \wedge \ldots \wedge \psi_n)$ за одно обращение к *интерполирующему решателю*, например SMTInterpol [SMTi], строятся все необходимые интерполянты $\rho_1, \ldots, \rho_{n-1}$ – формулы, удовлетворяющие следующим свойствам:

- общезначимы $\psi_1 \rightarrow \rho_1$, $(\psi_1 \wedge \psi_2) \rightarrow \rho_2$, …, $(\psi_1 \wedge \ldots \wedge \psi_{n-1}) \rightarrow \rho_{n-1}$;

- невыполнимы $\rho_1 \wedge (\psi_2 \wedge \ldots \wedge \psi_n)$, $\rho_2 \wedge (\psi_3 \wedge \ldots \wedge \psi_n)$, ..., $\rho_{n-1} \wedge \psi_n$;
- для любого $k \in \{1, \ldots, n-1\}$ все нелогические символы ρ_k являются общими для $(\psi_1 \wedge \ldots \wedge \psi_k)$ и $(\psi_{k+1} \wedge \ldots \wedge \psi_n)$.

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
  NCS_i: // некритическая секция
  SET_i: (y_i, s) := (1, i); // атомарное присваивание
  TST_i: while (y_1-i ≠ 0) ∧ (s = i) do skip end;
  CRS_i: // критическая секция
  RST_i: y_i := 0
end
```

7. Постройте системы переходов для следующих Promela-моделей и исследуйте их на наличие тупиков:

```
chan c = [1] of {byte}
active [2] proctype P() {
  do
  :: c!0
  :: c?_
  od
}
```

```
chan c = [1] of {byte}
active [2] proctype P() {
  do
  :: true -> c!0
  :: true -> c?_
  od
}
```

   Для модели слева рассмотрите также случай, когда *c* – синхронный канал.
8. Постройте множество всех возможных регионов для предикатов $x \geq 5$ и $x \leq 10$.

9. Используя декартову предикатную абстракцию, постройте модель (абстрактную систему переходов) 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 $x \leq 10$.

10. Для предыдущей задачи примените метод CEGAR.
11. 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).
12. 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.

13. 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 регулярные ω-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, т. е. 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 конечными автоматами и автоматами 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. Контрольный автомат

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 Крип-

ке M (обозначение: $M \vDash \varphi$), если φ истинна на любом вычислении M. Обратно: формула φ ложна на структуре M (обозначение: $M \nvDash \varphi$), если M имеет хотя бы одно вычисление π, такое что $\pi \nvDash \varphi$ (вычисления такого типа называются *контрпримерами*). В общих чертах проверку моделей можно рассматривать как поиск 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 строится *оракул*, или *монитор*, – компонент тестовой системы, функцией которого является вынесение вердикта о корректности или некорректности поведения программы. Оракул перехватывает стимулы, подаваемые на программу, и выдаваемые ей 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 (неопределенный, если быть точным).

Адаптируем этот подход к проверке моделей. Поскольку модель программы имеет форму системы переходов – по сути, автомата, – имеет смысл и оракул представить в таком виде. В результате получим *контрольный автомат*.

Контрольный автомат функционирует *параллельно* и *синхронно* с проверяемой моделью: если траектория модели не соответствует спецификации, он переходит в специальное состояние «ошибка» и остается в нем навсегда [Кар10]. Про тестирование мы подробнее поговорим в лекции 16.

В Promela контрольные автоматы описываются с помощью **never**-клауз (см. лекцию 10). Клаузы **ltl** автоматически преобразуются к клаузам **never**; для этого используется специальный алгоритм, рассматриваемый в следующей лекции.

Пусть в нашем распоряжении есть контрольный автомат. Как его использовать для проверки модели? Перебрать все возможные траектории и проверить каждую из них не получится: траекторий бесконечно много, причем каждая из них бесконечна. Можно ограничиться проверкой всех конечных префиксов, но это, очевидно, не гарантирует полноты верификации.

12.2. Конечные автоматы и регулярные языки

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. 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 этот вопрос дает теория ω-автоматов и регулярных ω-языков. Прежде чем переходить к этой теме, рассмотрим случай «классических» регулярных языков.

12.2.1. Описание языков конечными автоматами

Вспомним основные определения и обозначения.

Языком над конечным и непустым алфавитом Σ называется любое (возможно, бесконечное) множество конечных *цепочек* (*слов*), составленных из *символов* (*букв*) алфавита Σ.

Множество всех слов в алфавите Σ обозначается Σ^*, пустая цепочка – ε. Итак, $\mathcal{L}$ – язык над алфавитом Σ, если $\mathcal{L} \subseteq \Sigma^*$.

Известны формализмы, позволяющие описывать бесконечные языки конечным образом: формальные грамматики, конечные автоматы, регулярные выражения [Сер12]. Языки, задаваемые конечными автоматами, называются *автоматными языками*. Так как конечные автоматы имеют ту же выразительную силу, что и регулярные выражения, будем считать термины «автоматный язык» и «регулярный язык» синонимами.

Детерминированным конечным автоматом (*ДКА*) называется пятерка $\langle S, \Sigma, s_0, \delta, \mathcal{F}\rangle$, где

- S – множество *состояний*;
- Σ – *входной алфавит*;
- $s_0 \in S$ – *начальное состояние*;
- $\delta\colon S \times \Sigma \to S$ – *функция переходов*[1];
- $\mathcal{F} \subseteq S$ – множество *допускающих состояний*.

[1] Функция переходов, вообще говоря, является *частичной*, т. е. может быть не определенной для некоторых комбинаций состояний и входных символов.

Недетерминированным конечным автоматом (*НКА*) называется пятерка $\langle S, \Sigma, S_0, \delta, \mathcal{F}\rangle$, где $S_0 \subseteq S$ – множество *начальных состояний*, а $\delta: S \times \Sigma \to 2^S$ – *функция переходов*[1]. Смысл остальных элементов точно такой же, как в ДКА.

Все множества, указанные в этих двух определениях, предполагаются конечными. Множество S_0 может быть пустым, однако в этом случае язык, распознаваемый автоматом, также будет пустым (см. определение ниже).

Далее, если не оговорено противное, будем иметь дело с НКА, называя их при этом *конечными автоматами*, или просто *автоматами*.

Говорят, что автомат A *допускает* или *распознает* слово (траекторию конечной длины) $w = x_0 x_1 \ldots x_{n-1} \in \Sigma^*$, если w «переводит» A из одного из начальных состояний в одно из допускающих состояний. Формально это определяется следующим образом: существует вычисление $\pi = \{s_i\}_{i=0}^{n}$, такое что

- $s_0 \in S_0$;
- $s_i \in \delta(s_{i-1}, x_{i-1})$ для всех $1 \le i \le n$;
- $s_n \in \mathcal{F}$.

Множество всех слов, допускаемых автоматом A, называется *языком*, *допускаемым* (*распознаваемым*) A (обозначение: $\mathcal{L}_A$). Язык называется *регулярным*, если существует допускающий его автомат.

Пример 12.1

На рис. 12.1 изображены два автомата A и B над алфавитом $\{a, b\}$. Их начальные состояния изображены слева, 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:

$$\mathcal{L}_A = \{wa \mid w \in \{a, b\}^*\} \quad \text{и} \quad \mathcal{L}_B = \{(ab)^n, (ba)^n \mid n \in \mathbb{N}\}.$$

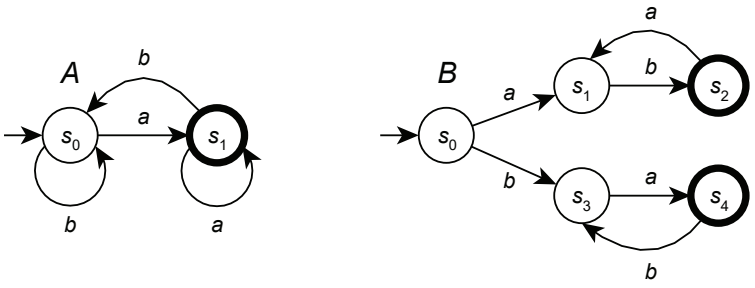


Рис. 12.1 ❖ Примеры конечных автоматов

□

[1] В некоторых работах по теории формальных языков в НКА допускаются переходы по *пустому символу* $\epsilon \notin \Sigma$; в этом случае функция переходов имеет вид $\delta: S \times (\Sigma \cup \{\epsilon\}) \to 2^S$. Заметим: по НКА с ϵ-переходами можно построить эквивалентный ему НКА без ϵ-переходов; более того, по НКА можно построить эквивалентный ему ДКА [Сер12].

12.2.2. Прямое произведение конечных автоматов

Прямым произведением 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:

$$\delta_{AB}((s_A, s_B), x) = \delta_A(s_A, x) \times \delta_B(s_B, x).$$

Проще говоря, прямое произведение двух автоматов – это автомат, моделирующий их параллельную и синхронную работу:

- в момент старта оба автомата находятся в начальных состояниях;
- на автоматы подаются одни и те же входные символы, при этом каждый из них осуществляет переходы в соответствии со своей функцией переходов;
- если в конце эксперимента и первый, и второй автоматы находятся в допускающих состояниях, слово считается распознанным.

Пример 12.2

На рис. 12.2 изображено прямое произведение автоматов A и B из предыдущего примера. В автомате $A \otimes B$ имеется одно начальное состояние (s_0, s_0) и одно достижимое допускающее состояние (s_1, s_4).

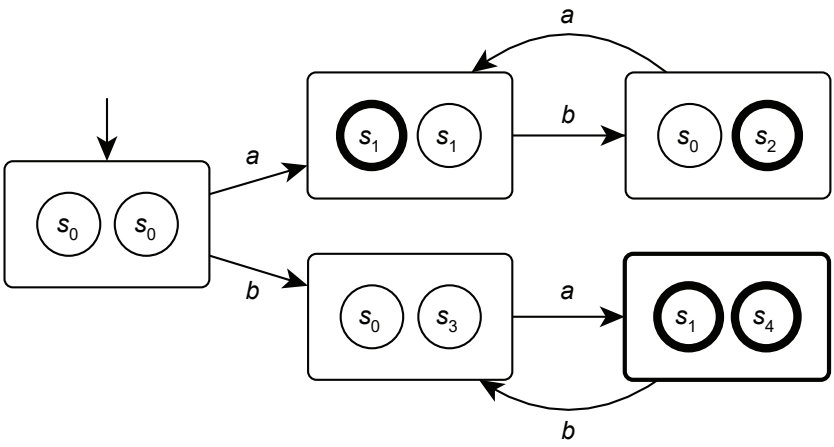


Рис. 12.2 ❖ Прямое произведение конечных автоматов

Нетрудно видеть, что автомат $A \otimes B$ допускает слова, имеющие вид $(ba)^n$, где $n \geq 1$, т. е. именно те слова, которые допускаются как автоматом A, так и автоматом B.

□

Наблюдение, сделанное в примере 12.2, имеет универсальный характер. Класс автоматных языков замкнут относительно операции пересечения: для любой пары конечных автоматов существует конечный автомат, распознающий пересечение их языков. Пересечение языков, распознаваемых автоматами A и B, совпадает с языком, распознаваемым прямым произведением $A \otimes B$ [Кар10]:

$$\mathcal{L}_A \cap \mathcal{L}_B = \mathcal{L}_{A \otimes B}.$$

12.2.3. Проверка регулярного языка 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 прямым произведением $M \otimes C_{\neg\varphi}$. 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: процесс проверки языка на пустоту можно организовать так, что если язык не пуст, предъявляется контрпример – путь из начального состояния в допускающее. Для автомата $M \otimes C_{\neg\varphi}$ этот путь задает вычисление модели M, в котором нарушается свойство φ.

12.3. Автоматы Бюхи и регулярные ω-языки

Классическая теория формальных языков построена для слов конечной длины. Вычисления реагирующих систем и моделирующих их структур Крипке

являются *бесконечными* (потенциально такие системы могут работать бесконечно долго); стало быть, для их верификации необходимы инструменты, позволяющие оперировать с бесконечными цепочками и их множествами: *ω-словами* и *ω-языками* соответственно[1] [Kap10].

Множество всех ω-слов над алфавитом Σ обозначается Σ^{ω}. Любое его подмножество называется ω-языком над алфавитом Σ. Заметим: алфавит по-прежнему конечный.

Аналогом конечных автоматов для ω-языков являются так называемые *ω-автоматы*. К ним относятся автоматы Бюхи[2] [Buc60], а также автоматы Мюллера[3] [Mul63] и автоматы Рабина[4] [Rab69]. Разумеется, разные типы автоматов имеют разные условия допустимости слов [Tho96]. Нас главным образом интересуют автоматы Бюхи – им посвящен следующий раздел.

12.3.1. Описание ω-языков автоматами Бюхи

Структурно автоматы Бюхи идентичны обычным конечным автоматам:

- *детерминированный автомат Бюхи* – $\langle S, \Sigma, s_0, \delta, \mathcal{F}\rangle$;
- *недетерминированный* – $\langle S, \Sigma, S_0, \delta, \mathcal{F}\rangle$.

Если не оговорено противное, автоматы являются недетерминированными. Для краткости будем называть их *автоматами Бюхи*, или просто *автоматами*.

Обозначим через $\inf(\pi)$ множество всех элементов последовательности π, встречающихся в ней бесконечное число раз.

Говорят, что ω-слово $w = x_0x_1 \ldots \in \Sigma^{\omega}$ *допускается* автоматом Бюхи $\langle S, \Sigma, S_0, \delta, \mathcal{F}\rangle$, 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 $i \geq 1$,

причем

- $\inf(\pi) \cap \mathcal{F} \neq \emptyset$: хотя бы одно допускающее состояние встречается бесконечное число раз.

---

[1] Вместо терминов «ω-слово» и «ω-язык» в литературе можно встретить термины «сверхслово» и «сверхъязык» соответственно.

[2] Юлиус Рихард Бюхи (Julius Richard Büchi; 1924–1984) – швейцарский логик и математик; известен тем, что показал связь между теорией автоматов и монадической логикой второго порядка с одной функцией следования (S1S, Second-order logic with 1 Successor).

[3] Дэвид Юджин Мюллер (в некоторых переводах – Маллер) (David Eugene Muller; 1924–2008) – американский ученый-информатик; пионер асинхронной схемотехники; специалист по теории автоматов и теории кодирования (коды Рида–Маллера, 1954).

[4] Михаэль Ошер Рабин (Michael Oser Rabin; род. в 1931 году) – израильский математик и информатик; автор и соавтор многих известных результатов, включая обобщение теоремы Бюхи на случай более одной функции следования (1969), вероятностный тест на простоту (тест Миллера–Рабина, 1980), алгоритм поиска образца в строке (алгоритм Рабина–Карпа, 1987); лауреат премии Тьюринга (1976).

Множество всех ω-слов, допускаемых автоматом A, называется *ω-языком, допускаемым A* (обозначение: $\mathcal{L}_A$); ω-язык называется *регулярным*, если существует допускающий его *недетерминированный* автомат Бюхи.

В отличие от конечных автоматов, не для всякого недетерминированного автомата Бюхи существует эквивалентный ему детерминированный автомат, поэтому в приведенном определении условие недетерминированности является существенным[1].

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?

- $\mathcal{L}_A$ – все ω-слова в алфавите $\{a, b\}$, не заканчивающиеся бесконечной последовательностью b^ω.
- $\mathcal{L}_B$ состоит из двух цепочек: $(ab)^\omega$ и $(ba)^\omega$.

□

12.3.2. Прямое произведение автоматов Бюхи

Класс регулярных языков замкнут относительно операции пересечения. То же справедливо в отношении регулярных ω-языков, но с небольшой оговоркой.

Пусть A и B – автоматы Бюхи. Наша цель – построить композицию $A \otimes B$, такую что цепочка допускается $A \otimes B$, если (и только если) она допускается как автоматом A, так и автоматом B, т. е. существует вычисление (с соответствующей траекторией), которое

- бесконечное число раз проходит через допускающие состояния автомата A и
- бесконечное число раз проходит через допускающие состояния автомата B.

Заметим: от вычисления не требуется, чтобы оно проходило через допускающие состояния автоматов A и B *одновременно*. Таким образом, ошибочно считать, что $\mathcal{F}_{A\otimes B} = \mathcal{F}_A \times \mathcal{F}_B$.

Чтобы допустить неодновременность прохода через допускающие состояния, введем более удобный для композиции тип автоматов.

Обобщенным автоматом Бюхи называется пятерка $\langle S, \Sigma, S_0, \delta, \boldsymbol{\mathcal{F}}\rangle$, где $\boldsymbol{\mathcal{F}} = \{\mathcal{F}_1, \ldots, \mathcal{F}_k\}$ – семейство множеств допускающих состояний.

Говорят, что ω-слово $w = x_0x_1 \ldots \in \Sigma^\omega$ *допускается* обобщенным автоматом Бюхи, если существует последовательность состояний $\pi = \{s_i\}_{i=0}^{\infty}$, такая что

---

[1] Для автоматов Мюллера и Рабина это не так: классы детерминированных и недетерминированных автоматов имеют одинаковую выразительную силу, такую же, как у недетерминированных автоматов Бюхи [Tho96].

- $s_0 \in S_0$;
- $s_i \in \delta(s_{i-1}, x_{i-1})$ для всех $i \geq 1$;
- $\inf(\pi) \cap \mathcal{F}_j \neq \emptyset$ для всех $1 \leq j \leq k$.

Обычный автомат Бюхи можно считать обобщенным автоматом Бюхи, у которого семейство множеств допускающих состояний состоит из одного множества.

Прямым произведением обобщенных автоматов Бюхи

$$A = \langle S_A, \Sigma, S_{A0}, \delta_A, \boldsymbol{\mathcal{F}}_A = \{\mathcal{F}_{A1}, \ldots, \mathcal{F}_{Ak}\}\rangle \quad \text{и} \quad B = \langle S_B, \Sigma, S_{B0}, \delta_B, \boldsymbol{\mathcal{F}}_B = \{\mathcal{F}_{B1}, \ldots, \mathcal{F}_{Bm}\}\rangle$$

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 $\boldsymbol{\mathcal{F}}_{AB} = \{(\mathcal{F}_{A1} \times S_B), \ldots, (\mathcal{F}_{Ak} \times S_B), (S_A \times \mathcal{F}_{B1}), \ldots, (S_A \times \mathcal{F}_{Bm})\}$.

Справедливо следующее утверждение: для любого обобщенного автомата Бюхи существует эквивалентный ему автомат Бюхи.

Пусть $G = \langle S, \Sigma, S_0, \delta, \boldsymbol{\mathcal{F}} = \{\mathcal{F}_1, \ldots, \mathcal{F}_k\}\rangle$ – обобщенный автомат Бюхи, в котором $k \geq 1$ (очевидно, что если $k = 0$, то, заменив пустое множество $\boldsymbol{\mathcal{F}}$ на $\{S\}$, получим эквивалентный автомат). Построим k «копий» автомата G: состояниями в новом автомате (назовем его B) являются пары (s, i), где s – состояние исходного автомата, а i – номер копии.

Начальные и допускающие состояния имеются только в первой копии: множества S_0 и $\mathcal{F}_1$ соответственно (точнее, $S_0 \times \{1\}$ и $\mathcal{F}_1 \times \{1\}$).

Из состояний множества $\mathcal{F}_i$, принадлежащих i-й копии, где $1 \leq i < k$, переходы ведут в «следующие» состояния $(i + 1)$-й копии:

если $s \in \mathcal{F}_i$ и $s' \in \delta(s, x)$, то $(s', i + 1) \in \delta_B((s, i), x)$.

Для k-й копии такие переходы ведут в первую копию (см. рис. 12.3). Остальные переходы осуществляются внутри копий.

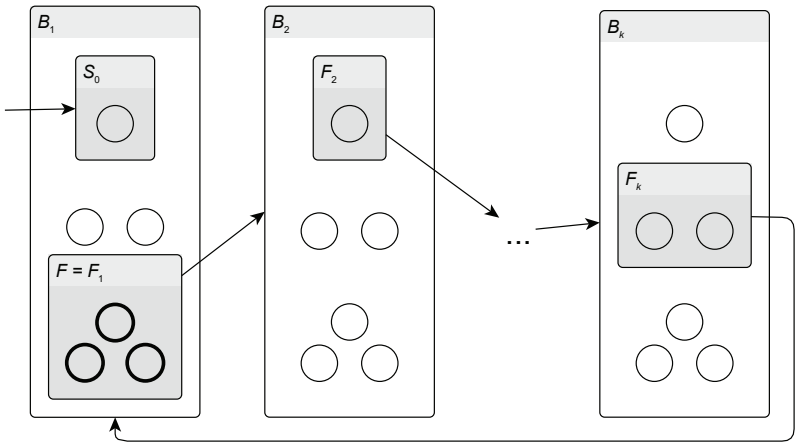


Рис. 12.3 ❖ Построение автомата Бюхи по обобщенному автомату Бюхи

Формально автомат B определяется следующим образом:

$$B = \langle S \times \{1, \ldots, k\}, \Sigma, S_0 \times \{1\}, \delta_B, \mathcal{F}_1 \times \{1\} \rangle,$$

где функция переходов δ_B имеет вид:

$$\delta_B\big((s, i), x\big) = \begin{cases} \{(s', i) \mid s' \in \delta(s, x)\}, & \text{если } s \notin \mathcal{F}_i \\ \{(s', (i \,\%\, k) + 1) \mid s' \in \delta(s, x)\} & \text{иначе} \end{cases}.$$

Легко убедиться, что автомат B эквивалентен обобщенному автомату G.

Пусть $w \in \mathcal{L}_B$. Тогда, по определению, существует вычисление $\pi_B = \{(s_j, i_j)\}_{j=0}^{\infty}$ с траекторией w, такое что $\inf(\pi_B) \cap (\mathcal{F}_1 \times \{1\}) \neq \emptyset$. Заметим: $\pi_G = \{s_j\}_{j=0}^{\infty}$ – вычисление автомата G с той же самой траекторией. Чтобы показать, что $w \in \mathcal{L}_G$, достаточно доказать, что $\inf(\pi_G) \cap \mathcal{F}_i \neq \emptyset$ для всех $1 \le i \le k$.

Рассмотрим произвольное состояние $(s_1, 1) \in \inf(\pi_B) \cap (\mathcal{F}_1 \times \{1\})$. После каждого посещения $(s_1, 1)$ автомат B переходит во вторую копию и может вернуться обратно только после последовательного прохода через все копии (для повторного посещения $(s_1, 1)$ может потребоваться несколько проходов). Это возможно, если для каждого $2 \le i \le k$ при нахождении в i-й копии автомат B рано или поздно переходит в состояние из множества $\mathcal{F}_i$. Бесконечное число проходов и конечность множеств $\mathcal{F}_i$ гарантируют, что в каждом из них найдется по состоянию, посещаемому бесконечное число раз, что и требовалось доказать.

Схожие рассуждения можно применить и при доказательстве в обратную сторону.

12.3.3. Проверка регулярного ω-языка 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* (*accepting cycle*). Заметим, что для обобщенного автомата Бюхи допускающий цикл должен содержать хотя бы по одному состоянию из каждого допускающего множества.

В самом деле, пусть s_F – допускающее состояние, а π_C – цикл, проходящий через s_F (считаем, что π_C начинается с s_F). Тогда вычисление $\pi = \pi' \cdot (\pi_C)^{\omega}$, где π' – путь из какого-нибудь начального состояния в состояние, предшествующее s_F, содержит 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 существует и допускающий цикл: в качестве такового 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 (КСС) графа состояний.

КСС ориентированного графа называется максимальный по включению вершин подграф, в котором любая пара вершин s и t соединена (ориентированным) путем:

- либо $s = t$ (пустой путь);
- либо существует путь из s в t и существует путь из t в s.

Сформулированное выше условие непустоты языка можно переформулировать так: язык, допускаемый автоматом Бюхи, не пуст тогда и только тогда, когда в графе состояний автомата достижима нетривиальная (отличная от одиночной вершины без петель) КСС, содержащая хотя бы одно допускающее состояние (из каждого допускающего множества, если речь об обобщенном автомате Бюхе).

Известен эффективный алгоритм построения множества достижимых КСС – алгоритм Тарьяна[1]; он основан на поиске в глубину и имеет вычислительную сложность $O(m)$, где m – число дуг графа [Tar72], [Kac03].

Второй подход к поиску допускающих циклов основан на *вложенном поиске в глубину* (*NDFS*, *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*), когда допускающие циклы ищутся не после построения 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* тогда и только тогда, когда 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).

Идея 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: 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 [Hol03].

---

[1] Роберт Эндре Тарьян (Robert Endre Tarjan; род. в 1948 году) – американский ученый-информатик; автор и соавтор множества структур данных (куча Фибоначчи, расширяющееся дерево) и алгоритмов на графах (построение множества КСС ориентированного графа, поиск ближайшего общего предка вершин дерева); лауреат премии Тьюринга (1986).

В табл. 12.1 представлен псевдокод алгоритма. Для наглядности предполагается, что начальное состояние одно (это s_0). Множество S содержит посещенные состояния – пары вида (s, i). В псевдокоде используются следующие вспомогательные предикаты и функции:

- *accepting*(s) – состояние s является допускающим;
- *succ*(s) – множество непосредственных последователей состояния s;
- $succ(s, i) \equiv \{(s', i) \mid s' \in succ(s)\}$;
- *choose*(X) – произвольный элемент непустого множества X.

Таблица 12.1. Вложенный поиск в глубину (поиск допускающего цикла в автомате Бюхи)

Основной поиск	**Вложенный поиск – вызов** $ndfs(s_F)$

`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 найден`
`  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 ∪ {(sF,1)};`
	`ndfs_stack := {sF}; // свой стек`
	`while ndfs_stack ≠ ∅ do`
	`  s := ndfs_stack.top();`
	`  if succ(s) ∩ dfs_stack ≠ ∅ then`
	`  return true // цикл найден`
	`  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`

12.4. Теоретико-автоматный подход к проверке моделей

Траектории структуры Крипке над множеством элементарных высказываний AP образуют ω-язык над алфавитом 2^{AP}. Этот ω-язык является регулярным – 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 $(s, L(s))$:

$$\delta(s, L(s)) = \{s' \mid (s, s') \in R\}.$$

Обратите внимание: в автомате B_M все состояния являются допускающими.

Пример 12.5

На рис. 12.4 приведена структура Крипке M и соответствующий автомат Бюхи B [Кар10].

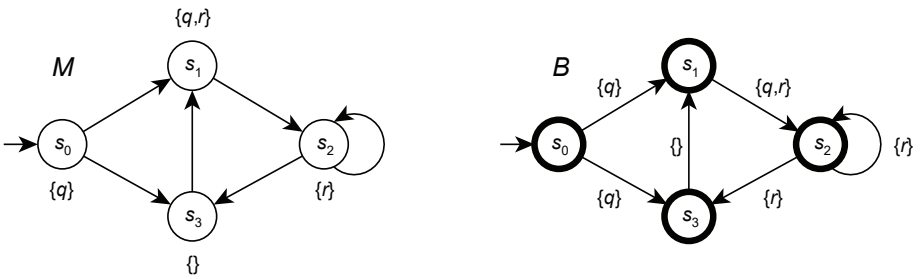


Рис. 12.4 ❖ Структура Крипке и соответствующий автомат Бюхи

Можно заметить, что пометки «переместились» с состояний на переходы. □

Более сложная задача – по формуле φ построить (обобщенный) автомат B_φ, допускающий те и только те траектории, на которых формула φ истинна (если формула описывает условие корректности модели, то контрольный автомат 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 известные нам формулы, часто возникающие на практике:

- **G***p* – свойство *безопасности* (*safety*): *нечто «плохое» никогда не произойдет (всегда все «хорошо»)*;
- **F***p* – свойство *живости* (*liveness*): *нечто «хорошее» когда-нибудь произойдет*;
- **GF***p* – свойство *справедливости* (*fairness*): *некоторое событие будет возникать периодически (время от времени)*;
- **FG***p* – свойство *стабилизации* (*stabilization*): *некоторое свойство станет истинным навсегда*.

Пусть $AP = \{p\}$ и, соответственно, $\Sigma = 2^{AP} = \{\emptyset, \{p\}\}$. Построим для этих формул автоматы Бюхи, распознающие траектории, на которых формулы истинны.

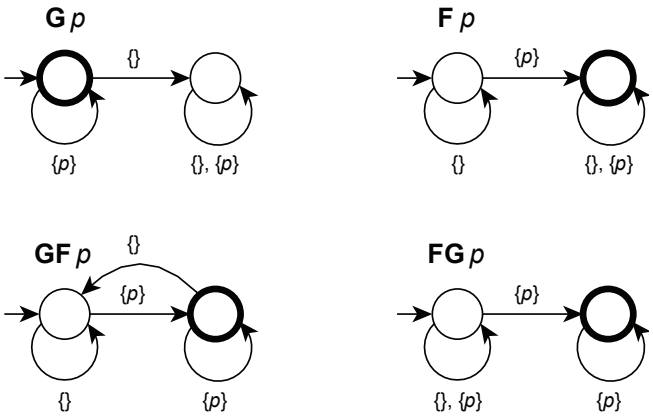


Рис. 12.5 ❖ Автоматы Бюхи для некоторых формул LTL

□

Теперь мы готовы рассмотреть общую схему проверки моделей для LTL. Описываемый способ, называемый *теоретико-автоматным подходом*, был предложен в 1985 году Мойше Варди[1] и Пьером Вольпером[2] [Var86]. На входе – модель M (структура Крипке) и спецификация φ (формула LTL над тем же множеством элементарных высказываний, что и модель). Проверка M на соответствие φ осуществляется следующим образом:

1) строится автомат B_M, распознающий все траектории модели M (и только их);
2) строится контрольный автомат $B_{\neg\varphi}$, распознающий все траектории, на которых ложна формула φ (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, $B_M \otimes B_{\neg\varphi}$ – обобщенный автомат Бюхи, но поскольку в модельном автомате B_M все состояния являются допускающими, множество допускающих состояний $B_M \otimes B_{\neg\varphi}$ определяется контрольным автоматом $B_{\neg\varphi}$: 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 как $O(|S| \cdot 2^{|\varphi|})$, где $|S|$– число состояний в модели M, а $|\varphi|$ – число символов в фор-

---

[1] Мойше Варди (Moshe Vardi; род. в 1954 году) – израильский ученый-информатик; известен своими работами в области проверки моделей, разрешения ограничений, теории баз данных; лауреат премии Гёделя (2000).

[2] Пьер Вольпер (Pierre Wolper; род. в 1955 году) – бельгийский ученый-информатик; известен в первую очередь как соавтор теоретико-автоматного подхода к проверке моделей; лауреат премии Гёделя (2000).

муле φ. Происхождение множителя $2^{|\varphi|}$ станет понятным из следующей лекции – это оценка числа состояний в автомате $B_{\neg\varphi}$.

12.5. Вопросы и упражнения

1. В чем отличие автоматов Бюхи от классических конечных автоматов?
2. Дайте определения автомата Бюхи и обобщенного автомата Бюхи. Опишите схему преобразования обобщенного автомата Бюхи к обычному. Завершите начатое в лекции доказательство эквивалентности такого преобразования.
3. Докажите, что класс регулярных ω-языков замкнут относительно операций объединения и дополнения.
4. 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.
5. Постройте 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).
6. Постройте автомат Бюхи над алфавитом $\{a, b\}$, допускающий все цепочки с конечным числом вхождений символа a и бесконечным числом вхождений символа b. Выразите этот язык формулой LTL.
7. Известно, что для любого НКА существует эквивалентный ему, т. е. допускающий тот же язык, ДКА. Верно ли, что для любого недетерминированного автомата Бюхи существует эквивалентный ему детерминированный автомат Бюхи? (Подсказка: рассмотрите язык из предыдущего упражнения.)
8. Приведите пример ω-языка, не являющегося регулярным.
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. Представьте автоматы Бюхи для формул $\mathbf{G}p$, $\mathbf{F}p$, $\mathbf{GF}p$ и $\mathbf{FG}p$ (см. лекцию) на языке Promela в виде **never**-клауз.
11. Постройте автоматы Бюхи, допускающие все траектории (и только их), на которых истинны 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) $\mathbf{G}\{p \rightarrow \mathbf{X}q\}$;

e) $\mathbf{G}\{p \rightarrow \mathbf{F}q\}$;
f) $\mathbf{G}\{p \rightarrow (p\ \mathbf{U}\ q)\}$.

12. Предложите алгоритм отыскания КСС ориентированного графа. Оцените его сложность. Напишите программы (на удобном вам языке программирования), реализующие предложенный алгоритм и алгоритм Тарьяна. Проведите их экспериментальное сравнение.
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

Ситуация, изучаемая нами в этой главе, напоминает диалог между «заказчиком», предъявляющим условия к поведению автомата, и «исполнителем», в задачу которого и входит построение подходящего автомата.

Б. А. Трахтенброт, Я. М. Барздинь.
Конечные автоматы
(поведение и синтез)

Рассматривается ключевой этап теоретико-автоматного подхода к проверке моделей для логики LTL – синтез автомата Бюхи, распознающего все траектории, на которых истинна заданная формула LTL (и только их). Результатом синтеза является недетерминированный обобщенный автомат Бюхи, число состояний которого экспоненциально зависит от длины формулы. Описанная процедура может быть использована и для проверки выполнимости формулы: если язык, допускаемый построенным автоматом, не пуст, то формула выполнима (существует траектория, на которой она истинна).

13.1. Постановка задачи и предварительные замечания

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}, распознающий язык φ – множество всех траекторий, на которых формула φ истинна (и только их). В известном

смысле LTL выступает в роли «метаязыка заказчика», на котором формулируются требования к поведению синтезируемого автомата [Тра70]. Сразу отметим, что задача имеет решение: для любой формулы LTL такой автомат существует. Обратное, вообще говоря, неверно: не для всякого автомата Бюхи найдется эквивалентная ему формула LTL[1].

Рассматриваемый алгоритм построения автомата Бюхи основан на метафоре *реализации обязательств* [Кар10]. Поясним, что имеется в виду. Если в формуле нет темпоральных операторов, ее обязательства реализуются в настоящем: в текущем состоянии автомат допускает только те пометки (означивания элементарных высказываний), которые делают формулу истинной (что происходит дальше, не важно). У формулы с темпоральными операторами появляются отложенные обязательства. Например:

- **F***p* обязуется *когда-нибудь в будущем* удовлетворить свойство *p* (если, конечно, оно не удовлетворено сейчас);
- **G***q* обязуется удовлетворять свойство *q всегда*, в каждый момент времени, начиная с текущего.

Это накладывает ограничения на функционирование соответствующих автоматов:

- $B_{\mathbf{F}p}$ не перейдет в допускающее состояние, пока не возникнет *p*;
- в $B_{\mathbf{G}q}$ невозможны переходы, для которых $q = false$.

Обязательства сложной формулы выражаются в терминах ее подформул. Так, формула $\chi\ \mathbf{U}\ \psi$ требует, чтобы подформула χ была истинна, пока ложна подформула ψ. Как можно представить информацию о такого рода ограничениях в автомате? Указав в каждом состоянии, какие подформулы истинны, а какие ложны (очевидно, что при этом не должно возникать противоречий), и «запрограммировав» переходы в соответствии с семантикой темпоральных операторов.

Множество всех подформул формулы φ, включая саму формулу φ, вместе с их отрицаниями[2] называется *замыканием* формулы φ и обозначается $closure(\varphi)$.

---

[1] По выразительности автоматы Бюхи соответствуют S1S – монадической логике второго порядка с одной функцией следования: ее сигнатура имеет вид ($\{0^{(0)}, succ^{(1)}\}$, $\{=^{(2)}, \in^{(2)}\}$); квантификацию можно проводить по индивидуальным переменным и множествам (одноместным предикатам). Как показал Бюхи, каждой формуле S1S можно сопоставить регулярный ω-язык (в алфавите $\{0, 1\}^m$, где m – число свободных переменных, включая индивидуальные переменные и множества), ω-слова которого кодируют модели этой формулы, и наоборот – любой регулярный ω-язык (в алфавите $\{0, 1\}^m$) является S1S-определимым в том смысле, что существует S1S-формула с m свободными переменными, модели которой кодируются ω-словами этого языка [Muk12]. Соответствие между моделью $(x_1, \ldots, x_k; x_{k+1}, \ldots, x_m)$ и словом $a_0a_1\ldots$ определяется так: для всех $i \in \mathbb{N}_0$ и $j \in \{1, \ldots, m\}$ имеет место эквивалентность $(a_i[j] = 1) \leftrightarrow \begin{cases} x_j = i, & \text{если } j \le k \\ i \in X_j, & \text{если } j > k \end{cases}$.

[2] Формулы $\neg\neg\psi$ и ψ считаются совпадающими.

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 (точнее, истинны на вычислении, начинающемся с этого состояния). Начнем с элементарных высказываний и постепенно будем добавлять в разметку все более сложные подформулы, как показано в табл. 13.1.

Таблица 13.1. Разметка состояний истинными подформулами

Состояние s	**Пометка** $L(s)$	**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: достаточно знать значения истинности элементарных высказываний. Чтобы установить истинность подформул, содержащих темпоральные операторы, приходится «заглядывать в будущее». Например, находясь в состоянии s_0 или s_1 и не зная, что будет дальше, нельзя сказать, истинна формула φ или ложна. В состоянии s_2 необходимая информация появляется: $s_2 \vDash (p \wedge q)$. Далее, 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 **U** в лекции 9). Так как формула φ входит в разметку начального состояния, она истинна на траектории:

$$(\{p\}\{q\}\{p, q\}\{q\}\emptyset)^{\omega} \vDash (p \vee q)\ \mathbf{U}\ (p \wedge q).$$

□

В описанном примере, установив истинность формулы φ в состоянии s_2, мы возвратились в предшествующие состояния и пополнили их разметку. На это можно взглянуть по-другому: переход в состояние s_2 (и вообще в любое состояние, размеченное φ) возможен только из состояний с определенной разметкой: в частности, в нее должны входить $(p \vee q)$ и φ. Так учитывается семантика оператора **U**.

Приводимый ниже алгоритм строит недетерминированный обобщенный автомат Бюхи (для краткости мы будем называть его автоматом Бюхи, или просто автоматом). Описание во многом базируется на книге [Kap10]; дополнительная информация может быть найдена в статье [Wol02].

Для простоты мы будем рассматривать формулы LTL, в которых используются две логические связки, ¬ (отрицание) и ∨ (дизъюнкция), и два темпоральных оператора, **X** (*neXt time*) и **U** (*Until*). Это не ограничивает общности рассуждений, поскольку названные связки и операторы образуют базис LTL (см. лекцию 9).

13.2. Состояния автомата

Состояниями автомата B_φ являются множества подформул формулы φ, возможно с отрицаниями, т. е. $S \subseteq 2^{closure(\varphi)}$. Подформулы, составляющие одно состояние (или «мир», обращаясь к семантике возможных миров), должны быть *согласованными*. Состояния также должны быть *максимально насыщенными* по включению подформул: это позволяет однозначно определить, возможен тот или иной переход или нет (об этом идет речь в следующем разделе).

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 будем называть *условиями локальной согласованности*.

Множество всех атомов формулы φ (элементарных множеств замыкания φ) обозначается символом $atoms(\varphi)$.

Элементарные множества связаны с множествами Линденбаума[1] и отчасти множествами Хинтикки[2], используемыми в табличных методах и теории моделей (в множествах Хинтикки сформулированные условия выполняются в одну сторону: если $(\chi \vee \psi) \in s$, то $\chi \in s$ или $\psi \in s$ и т. д.).

Определение можно очевидным образом расширить на другие логические связки и темпоральные операторы.

Почему в условиях локальной согласованности не упоминается темпоральный оператор **X**? Дело в том, что у формул вида $\mathbf{X}\psi$ нет обязательств в настоящем (они «отложены на завтра»). Поэтому каждое из четырех множеств, приведенных ниже, является согласованным:

$$\{\psi, \mathbf{X}\psi\}, \quad \{\psi, \neg\mathbf{X}\psi\}, \quad \{\neg\psi, \mathbf{X}\psi\}, \quad \{\neg\psi, \neg\mathbf{X}\psi\}.$$

Построение множества атомов заданной LTL-формулы φ удобно организовать в форме следующей процедуры:

1) выписать все входящие в φ элементарные высказывания и подформулы вида $\mathbf{X}p$, предварительно выполнив преобразования:

$$\mathbf{X}\neg\psi \equiv \neg\mathbf{X}\psi, \quad \mathbf{X}(\chi \vee \psi) \equiv \mathbf{X}\chi \vee \mathbf{X}\psi, \quad \mathbf{X}(\chi\ \mathbf{U}\ \psi) \equiv \mathbf{X}\chi\ \mathbf{U}\ \mathbf{X}\psi;$$

2) перебрать все комбинации значений истинности выписанных условий:
   a) для каждой из них построить множество истинных подформул φ, полученных из этих условий с помощью классических логических связок (истинные условия тоже входят в это множество);
3) для каждой подформулы φ вида $(\chi\ \mathbf{U}\ \psi)$ (в порядке от вложенных к внешним) проверить вхождение ψ в каждое из построенных на шаге 2 множеств s:
   a) если $\psi \in s$, добавить $(\chi\ \mathbf{U}\ \psi)$ в S;
   b) в противном случае:
      i) если $\chi \in s$, построить копию s с добавлением $(\chi\ \mathbf{U}\ \psi)$;
      ii) иначе оставить s без изменений.

Следует отметить, что добавление в одно из множеств подформулы вида $(\chi\ \mathbf{U}\ \psi)$ может «спровоцировать» добавление в него «классических» подформул, содержащих $(\chi\ \mathbf{U}\ \psi)$. Например, для формулы

$$\mathbf{F}p \vee q \equiv (\mathit{true}\ \mathbf{U}\ p) \vee q$$

---

[1] Адольф Линденбаум (Adolf Lindenbaum; 1904–1941) – польский математик и логик; известен как автор леммы о возможности расширения любой значимой теории первого порядка до полной; один из основоположников современной алгебраической логики (алгебра Линденбаума–Тарского); погиб во время холокоста.

[2] Каарло Яакко Юхани Хинтикка (Kaarlo Jaakko Juhani Hintikka; 1929–2015) – финский философ; наиболее известен разработкой дистрибутивных нормальных форм (1953), модельных множеств (1955) и теоретико-игровой семантики (1973); иностранный член Российской академии наук (1999); лауреат премии Рольфа Шока (2005).

при $p = true$ и $q = false$ получим элементарное множество

$\{p, (true\ \mathbf{U}\ p), (true\ \mathbf{U}\ p) \vee q\}$.

Таким образом, шаги 2 и 3 выполняются итерационно.

Пример 13.3

Для формулы $\varphi \equiv (p \vee q)\ \mathbf{U}\ (p \wedge q)$ существуют шесть элементарных множеств подформул: они показаны в табл. 13.2 (как мы договаривались, подформулы с отрицаниями не указываются, но подразумеваются).

Таблица 13.2. Атомы формулы $(p \vee q)\ \mathbf{U}\ (p \wedge q)$

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*	$\{p, q, (p \vee q), (p \wedge q)\}$	$s_6 = \{p, q, (p \vee q), (p \wedge q), \varphi\}$

□

13.3. Переходы автомата

Формулы без темпоральных операторов никак не ограничивают переходы автомата. Для темпоральных формул ситуация иная: соседние состояния (обозначим их s и s') должны удовлетворять следующим требованиям:

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 $(\chi\ \mathbf{U}\ \psi) \in s'$.

Данные требования будем называть *условиями реализации обязательств* (обозначение: $s \to s'$). Разумеется, их можно определить и для других темпоральных операторов.

Обратите внимание на связку «тогда и только тогда»: в частности, если в $closure(\varphi)$ есть подформула $\mathbf{X}\psi$, то переход в состояние, содержащее ψ, возможен *только* из состояний, содержащих $\mathbf{X}\psi$.

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 только один темпоральный оператор):

$$\varphi \in s \leftrightarrow \begin{cases} (p \wedge q) \in s \\ (p \vee q) \in s. \\ \varphi \in s' \end{cases}$$

Для пары состояний s_2 и s'_2 имеем: $\varphi \notin s_2$; соответственно,

$$((p \wedge q) \notin s_2) \wedge (((p \vee q) \notin s_2) \vee (\varphi \notin s'_2)) \equiv \varphi \notin s'_2.$$

Таким образом, $\delta(s_2) = \{s_1, s_2, s_4\}$, где

- $s_1 = \emptyset$;
- $s_2 = \{p, (p \vee q)\}$;
- $s_4 = \{q, (p \vee q)\}$.

Для пары состояний s_3 и s'_3 имеем: $\varphi \in s_3$; соответственно,

$$((p \wedge q) \in s_3) \vee (((p \vee q) \in s_3) \wedge (\varphi \in s'_3)) \equiv \varphi \notin s'_3.$$

Таким образом, $\delta(s_3) = \{s_3, s_5, s_6\}$, где

- $s_3 = \{p, (p \vee q), \varphi\}$;
- $s_5 = \{q, (p \vee q), \varphi\}$;
- $s_6 = \{p, q, (p \vee q), (p \wedge q), \varphi\}$.

□

Каждый переход из состояния s имеет пометку $x(s)$ – множество элементарных высказываний, входящих в s:

$$x(s) = (s \cap AP).$$

Таким образом, все переходы из одного состояния имеют одинаковые пометки.

13.4. Начальные и допускающие состояния автомата

Начальными состояниями автомата B_φ являются те атомы формулы φ, которые содержат саму формулу φ. Таким способом задается цель автомата – распознать траектории, на которых истинна φ.

Обязательства подформул без темпоральных операторов реализуются в состояниях, содержащих эти подформулы, поэтому в искомом автомате нет состояний, где эти обязательства не выполнены (либо в состоянии нет подформулы, либо ее обязательства исполняются «на месте»). Похожая ситуация имеет место с подформулами вида $\mathbf{X}\psi$: отличие лишь в том, что их обязательства реализуются не в состояниях с $\mathbf{X}\psi$, а в следующих за ними. Таким образом, при определении допускающих состояний можно не учитывать подформулы без темпоральных операторов и подформулы вида $\mathbf{X}\psi$.

Для каждой подформулы вида $(\chi\ \mathbf{U}\ \psi)$ автомат B_φ, являющийся, вообще говоря, обобщенным автоматом Бюхи, содержит отдельное допускающее множество $\mathcal{F}_{\chi \mathbf{U} \psi}$. Это множество включает все состояния, в которых нет $(\chi\ \mathbf{U}\ \psi)$ или есть ψ:

$s \in \mathcal{F}_{\chi \mathbf{U} \psi}$ тогда и только тогда, когда $(\chi\ \mathbf{U}\ \psi) \notin s$ или $\psi \in s$.

Пример 13.5

Начальными состояниями автомата B_φ для формулы $\varphi \equiv (p \vee q)\ \mathbf{U}\ (p \wedge q)$ являются s_3, s_5 и s_6 (см. табл. 13.2). Допускающие состояния приведены в табл. 13.3.

Таблица 13.3. Допускающие состояния автомата Бюхи B_φ для формулы $\varphi \equiv (p \vee q)\ \mathbf{U}\ (p \wedge q)$

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 $(p \wedge q) \in s_6$

Обратите внимание: в формуле φ есть только одна подформула с $\mathbf{U}$; соответственно, в B_φ только одно допускающее множество.

□

13.5. Алгоритм построения автомата

Подведем итог. Пусть AP – множество элементарных высказываний, а φ – формула LTL над множеством AP. Обобщенный автомат Бюхи $B = \langle S, \Sigma, S_0, \delta, \boldsymbol{\mathcal{F}} \rangle$, допускающий те и только те траектории, на которых истинна формула φ, строится так:

- $S = atoms(\varphi)$;
- $\Sigma = 2^{AP}$;
- $S_0 = \{s \in S \mid \varphi \in s\}$;
- $\delta: S \times \Sigma \to 2^S$ 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

$$\mathcal{F}_{\chi \mathbf{U} \psi} = \{s \in S \mid ((\chi \mathbf{U} \psi) \notin s) \vee (\psi \in s)\}.$$

Описанный метод синтеза не гарантирует минимальности автомата – вопросы оптимизации выходят за рамки этой книги. Нетрудно показать, что число состояний автомата B_φ оценивается величиной $O(2^{|\varphi|})$, где $|\varphi|$ – длина формулы φ (число символов в ее записи).

Доказательство корректности алгоритма можно найти в [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[1]
  - $s_1 = \emptyset$;
  - $s_2 = \{p, (p \vee q)\}$;
  - $s_3 = \{p, (p \vee q), \varphi\}$;
  - $s_4 = \{q, (p \vee q)\}$;
  - $s_5 = \{q, (p \vee q), \varphi\}$;
  - $s_6 = \{p, q, (p \vee q), (p \wedge q), \varphi\}$;
- $\Sigma = 2^{AP} = \{\emptyset, \{p\}, \{q\}, \{p, q\}\}$;
- $S_0 = \{s \in S \mid \varphi \in s\} = \{s_3, s_5, s_6\}$;
- $\mathcal{F} = \mathcal{F}_\varphi = \{s \in S \mid (\varphi \notin s) \vee ((p \wedge q) \in s)\} = \{s_1, s_2, s_4, s_6\}$[2];
- $\delta: S \times \Sigma \to 2^S$:
  - $\delta(s_1, \emptyset) = \{s_1, \ldots, s_6\}$ – все состояния;
  - $\delta(s_2, \{p\}) = \{s_1, s_2, s_4\}$ – состояния, не содержащие φ;
  - $\delta(s_3, \{p\}) = \{s_3, s_5, s_6\}$ – состояния, содержащие φ;
  - $\delta(s_4, \{q\}) = \{s_1, s_2, s_4\}$;
  - $\delta(s_5, \{q\}) = \{s_3, s_5, s_6\}$;
  - $\delta(s_6, \{p, q\}) = \{s_1, \ldots, s_6\}$.

Граф состояний построенного автомата показан на рис. 13.1.

---

[1] Еще раз подчеркнем, что подформулы с отрицаниями для краткости опущены.

[2] В автомате B_φ одно множество допускающих состояний – это обычный автомат Бюхи.

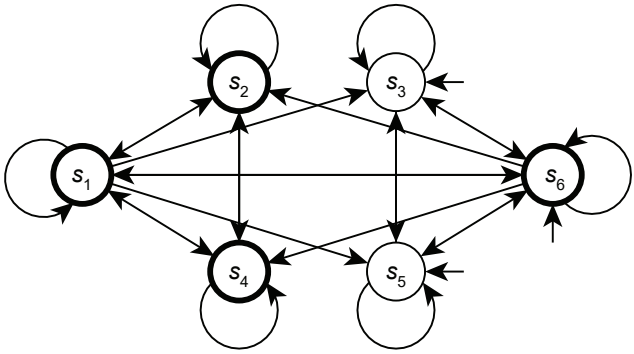


Рис. 13.1 ❖ Автомат Бюхи B_φ для формулы $\varphi \equiv (p \vee q)\ \mathbf{U}\ (p \wedge q)$

□

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 $\varphi \equiv \mathbf{G}\{p \rightarrow \mathbf{X}q\}$. Прежде всего заметим, что[1]:

$$\varphi \equiv \mathbf{G}\{p \rightarrow \mathbf{X}q\} \equiv \neg\mathbf{F}\neg(p \rightarrow \mathbf{X}q) \equiv \neg(true\ \mathbf{U}\ \neg(p \rightarrow \mathbf{X}q)) \equiv \neg(true\ \mathbf{U}\ (p \wedge \neg\mathbf{X}q));$$

$$\neg\varphi \equiv true\ \mathbf{U}\ (p \wedge \neg\mathbf{X}q).$$

- $S = atoms(\varphi) = \{s_1, \ldots, s_{14}\}$ – см. табл. 13.4[2].
- $\Sigma = 2^{AP} = \{\emptyset, \{p\}, \{q\}, \{p, q\}\}$.
- $S_0 = \{s \in S \mid \varphi \in s\} = \{s_1, s_4, s_6, s_9, s_{11}, s_{13}\}$ – состояния, не содержащие $\neg\varphi$.
- $\mathcal{F} = \mathcal{F}_{\neg\varphi} = \{s \in S \mid (\neg\varphi \notin s) \vee ((p \wedge \neg\mathbf{X}q) \in s)\} = \{s_1, s_3, s_4, s_6, s_8, s_9, s_{11}, s_{13}\}$.
- δ: $S \times \Sigma \rightarrow 2^S$:
  - $\delta(s_1, \emptyset) = \{s_1, s_6, s_9\}$ – состояния, не содержащие ни $\neg\varphi$, ни q;
  - $\delta(s_2, \emptyset) = \{s_2, s_3, s_7, s_{10}\}$ – состояния, содержащие $\neg\varphi$, но не содержащие q;
  - $\delta(s_3, \{p\}) = \{s_1, s_2, s_3, s_6, s_7, s_9, s_{10}\}$ – состояния, не содержащие q;
  - $\delta(s_4, \{q\}) = \delta(s_1, \emptyset)$;
  - $\delta(s_5, \{q\}) = \delta(s_2, \emptyset)$;
  - $\delta(s_6, \emptyset) = \{s_4, s_{11}, s_{13}\}$ – состояния, содержащие q, но не содержащие $\neg\varphi$;
  - $\delta(s_7, \emptyset) = \{s_5, s_8, s_{12}, s_{14}\}$ – состояния, содержащие $\neg\varphi$ и q;
  - $\delta(s_8, \{p, q\}) = \delta(s_3, \{p\})$;
  - $\delta(s_9, \{p\}) = \delta(s_6, \emptyset)$;
  - $\delta(s_{10}, \{p\}) = \delta(s_7, \emptyset)$;
  - $\delta(s_{11}, \{q\}) = \delta(s_6, \emptyset)$;
  - $\delta(s_{12}, \{q\}) = \delta(s_7, \emptyset)$;
  - $\delta(s_{13}, \{p, q\}) = \delta(s_6, \emptyset)$;
  - $\delta(s_{14}, \{p, q\}) = \delta(s_7, \emptyset)$.

---

[1] Поскольку нами не были определены условия локальной согласованности и реализации обязательств для темпорального оператора **G**, формула преобразуется к эквивалентной, содержащей только **X** и **U**.

[2] Обратите внимание: во все состояния (атомы) входит подформула *true*.

Таблица 13.4. Состояния автомата Бюхи B_φ для формулы $\varphi \equiv \mathbf{G}\{p \to \mathbf{X}q\}$

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*	$\{true, p, q, (p \wedge \neg\mathbf{X}q)\}$	$s_8 = \{true, p, q, (p \wedge \neg\mathbf{X}q), \neg\varphi\}$
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*	$\{true, p, q, \mathbf{X}q\}$	$s_{13} = \{true, p, q, \mathbf{X}q\}$
				$s_{14} = \{true, p, q, \mathbf{X}q, \neg\varphi\}$

Граф состояний построенного автомата показан на рис. 13.2.

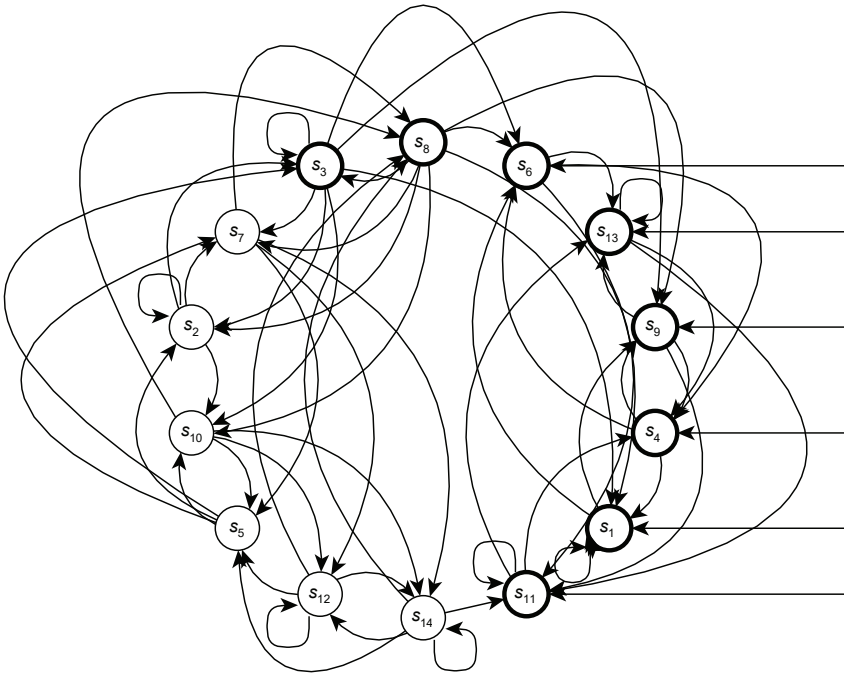


Рис. 13.2 ❖ Автомат Бюхи B_φ для формулы $\varphi \equiv \mathbf{G}\{p \to \mathbf{X}q\}$

□

13.6. Общая схема проверки моделей для логики LTL

Синтез автомата Бюхи по формуле LTL – ключевой этап теоретико-автоматного подхода к проверке моделей. Общая схема метода приведена на рис. 13.3. Входными данными для инструмента верификации 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, и в ней ищется допускающий цикл:

- если допускающего цикла нет, модель корректна;
- в противном случае в модели есть ошибка (предъявляется контрпример).

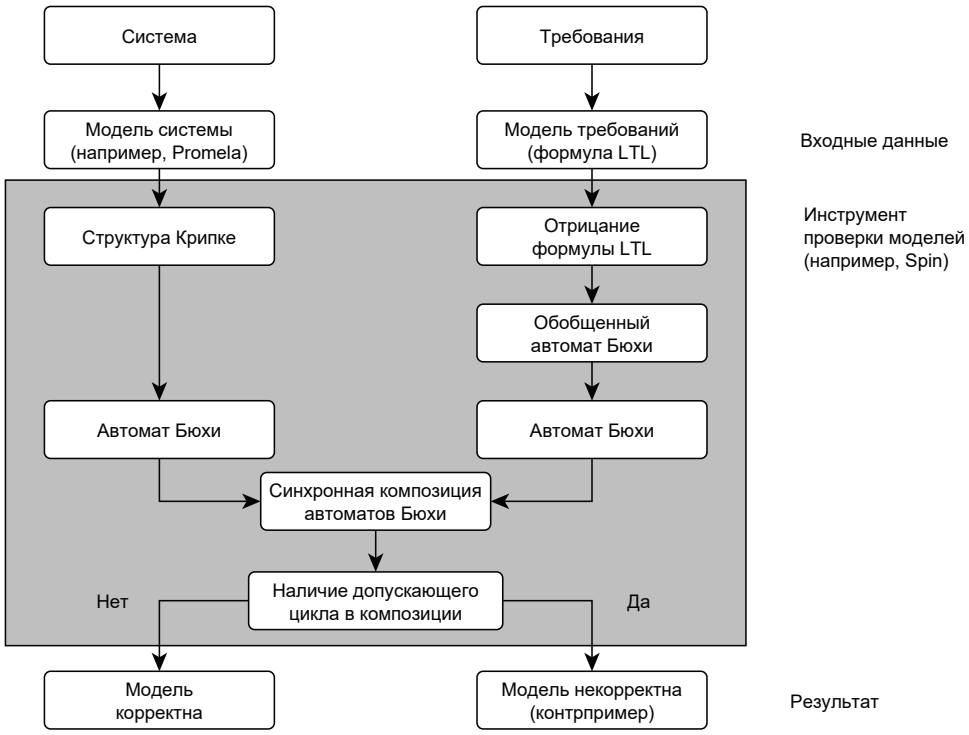


Рис. 13.3 ❖ Общая схема проверки моделей для логики LTL

Если быть более точным, по формуле строится обобщенный автомат Бюхи. Его можно и не преобразовывать в обычный автомат (как это показано на схеме), однако в этом случае придется искать цикл, проходящий через все допускающие множества.

13.7. Вопросы и упражнения

1. Дайте определения замыкания формулы и элементарного множества подформул.
2. Выпишите условия реализации обязательств для темпоральных операторов **X** и **U**.
3. Для темпоральных операторов **F** и **G** определите условия локальной согласованности, условия реализации обязательств и условия на допускающие состояния.
4. Пусть φ – некоторая формула LTL, 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. Выпишите атомы для следующих формул LTL:
   a) p **U** (**X**$\neg q$);
   b) p **U** ($\neg q$ **U** (**X**p));
   c) **G**$\{p \rightarrow$ **FX**$\neg q\}$.
7. Используя алгоритм, рассмотренный на лекции, постройте автоматы Бюхи для следующих формул LTL:
   a) p ($AP = \{p\}$);
   b) **X**p;
   c) **G**p;
   d) **F**p;
   e) **GF**p;
   f) **FG**p;
   g) p **U** q ($AP = \{p, q\}$);
   h) p **U** (q **U** r) ($AP = \{p, q, r\}$).
8. Постройте автоматы Бюхи для следующих формул LTL ($AP = \{p, q\}$):
   a) $p \wedge$ (**XG**$\neg q$);
   b) **F**$\{p \rightarrow$ **X**$\neg q\}$;
   c) **GF**$\{\neg p \wedge$ **X**$q\}$;
   d) **FG**$\{$**X**$p \wedge q\}$;
   e) ($p \wedge \neg$**X**q) **U** ($\neg p \wedge q$);
   f) (**FG**p) $\rightarrow$ (**X**q);
   g) **G**$\{p \rightarrow (p$ **U** $q)\}$.
9. Предложите способы оптимизации автомата Бюхи, конструируемого по формуле LTL.
10. Напишите программу (на удобном вам языке программирования), синтезирующую автомат Бюхи по заданной формуле LTL.
11. 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 или контрпример.

Лекция 14

Символьная 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 символьная проверка моделей для логики LTL. Отличие от классического подхода, рассмотренного ранее, состоит в том, что модель (множество состояний и отношение переходов), а также автомат Бюхи, построенный по формуле LTL, представляются неявно, в символьной форме. Общая идея метода остается прежней, но алгоритмы поиска в графах состояний заменяются на манипуляции с их символьными представлениями, что частично решает проблему комбинаторного взрыва. Ключевую роль здесь играют способы символьного представления множеств и отношений. Рассматривается один из таких способов, позволяющий создавать эффективные символьные алгоритмы, – двоичные решающие диаграммы (BDD, Binary Decision Diagrams).

14.1. Символьные методы верификации

Традиционные методы проверки моделей, включая рассмотренный ранее теоретико-автоматный подход (см. лекции 12 и 13), называются *методами с явным представлением состояний* (*explicit-state model checking*): каждое состояние и каждый переход модели некоторым образом кодируются и при анализе модели хранятся в памяти компьютера. Проблема в том, что число состояний в модели параллельной системы может превысить любые мыслимые значения, скажем число атомов во Вселенной[1]. Соответственно, верификация, основанная на исследовании пространства состояний, будет либо упираться в нехватку памяти, либо длиться «вечно».

Представители «классической школы» проверки моделей прибегают к разного рода «трюкам», оптимизирующим представление состояний в памяти. В инструменте Spin, например, используется сжатие данных (с потерями и без): *хеширование* (*bitstate hashing*) [Hol98], *рекурсивное индексирование* (*recursive indexing*) [Hol97] и другие подходы. Применяются техники параллельного обхода графов состояний [Бур11], что позволяет сократить время верификации на 2 порядка и более (в зависимости от числа компьютеров). Задействуются и более продвинутые методы: *редукция частичных порядков* (*partial order reduction*) [God94] и *редукция симметрии* (*symmetry reduction*) [Cla99]. Краткий обзор способов борьбы с комбинаторным взрывом можно найти в [Кам15].

Альтернативу методам с явным представлением состояний составляют *символьные*, или, как их еще называют, *символические, методы* (*symbolic methods*). Суть в том, что множество состояний задается не путем перечисления элементов, а неявно, как решение некоторой системы уравнений. Пространство состояний системы, огромное или даже бесконечное, обладает определенной регулярностью, что позволяет выразить его в компактной форме[2].

Пусть χ_M – описание (символьное представление) пространства состояний модели M, а χ_φ – описание требований φ. Для простоты будем считать, что φ – инвариант, т. е. свойство, которое должно быть истинным в каждом состоянии. Проверка модели в этом случае есть проверка общезначимости импликации $\chi_M \rightarrow \chi_\varphi$, или, что то же самое, невыполнимости конъюнкции $\chi_M \wedge \neg\chi_\varphi$. Для решения этой задачи применяются SAT- и SMT-решатели (см. лекцию 8).

---

[1] По современным представлениям, число атомов в наблюдаемой части Вселенной – это величина порядка 10^{79}–10^{81}. Для сравнения: минимальное количество неповторяющихся шахматных партий оценивается как 10^{118} (так называемое *число Шеннона*). Число возможных состояний в моделях программ может быть гораздо больше.

[2] К слову сказать, код программы – неявное и экономное представление ее пространства состояний.

Символьная проверка моделей была разработана в конце 1980-х – начале 90-х годов [Bur90]. Один из первых методов был представлен в диссертации Кеннета Макмиллана[1] [Mcm92] и реализован в инструменте SMV (Symbolic Model Verifier). Современными «реинкарнациями» SMV являются NuSMV [NuSMV] и nuXmv [Cav14].

Изначально символьная проверка моделей была ориентирована на логику CTL (Computation Tree Logic), мы же, следуя теоретико-автоматному подходу, рассмотрим ее разновидность для LTL [Roz10].

14.1.1. Символьное представление множеств и отношений

Пусть S – счетное множество. Для символьного представления подмножеств множества S введем *нумерацию*: сопоставим каждому элементу $x \in S$ уникальное число

$$[\![x]\!] \in \mathbb{N}_0.$$

Любая арифметическая формула φ_X с одной свободной переменной (например, алгебраическое уравнение) является описанием некоторого подмножества $X \subseteq S$[2]:

$x \in X$ тогда и только тогда, когда $\vDash \varphi_X([\![x]\!])$.

Если множество S конечно, можно поступить проще: *закодировать* его элементы двоичными векторами – точками пространства $\mathbb{B}^n$, 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 табл. 14.1 показано соответствие между операциями над множествами и операциями над характеристическими функциями.

---

[1] Кеннет Макмиллан (Kenneth McMillan) – американский исследователь в области формальных методов; известен своими работами по символьной проверке моделей и композиционной верификации; автор инструмента SMV; лауреат премии Париса Канеллакиса за теоретические и практические достижения (1998).

[2] Множество натуральных чисел (подмножество $\mathbb{N}_0$), которое может быть описано формулой языка арифметики первого порядка, называется *арифметическим множеством*.

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	$\bar{X}$	$\chi_{\bar{X}} \equiv \neg\chi_X$
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	$X \setminus Y$	$\chi_{X \setminus Y} \equiv \chi_X \wedge \neg\chi_Y$

Отношения также могут быть представлены с помощью характеристических функций. Пусть $R \subseteq S \times S$ – бинарное отношение. Если двоичная кодировка элементов S требует n бит, то χ_R – характеристическая функция R – есть булева функция от $2n$ переменных:

$\vec{x} = (x_1, \ldots, x_n)$ и $\vec{x}' = (x'_1, \ldots, x'_n)$.

Некоторые операции над бинарными отношениями представлены в табл. 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	$Im(R) = \{y \in S \mid \exists x((x, y) \in R)\}$	$\chi_{Im(R)}(\vec{x}') = \exists\vec{x}(\chi_R(\vec{x}, \vec{x}')) = \bigvee_{\vec{x} \in \mathbb{B}^n} \chi_R(\vec{x}, \vec{x}')$
Домен отношения	$Dom(R) = \{x \in S \mid \exists y((x, y) \in R)\}$	$\chi_{Dom(R)}(\vec{x}) = \exists\vec{x}'(\chi_R(\vec{x}, \vec{x}')) = \bigvee_{\vec{x}' \in \mathbb{B}^n} \chi_R(\vec{x}, \vec{x}')$
Ограничения домена и образа		
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 и прообраз множества относительно отношения		
Образ	$R(X) = Im(X \lhd R)$	$\chi_{R(X)}(\vec{x}') = \exists\vec{x}(\chi_X(\vec{x}) \wedge \chi_R(\vec{x}, \vec{x}'))$
Прообраз	$R^{-1}(Y) = Dom(R \rhd Y)$	$\chi_{R^{-1}(X)}(\vec{x}) = \exists\vec{x}'(\chi_R(\vec{x}, \vec{x}') \wedge \chi_Y(\vec{x}'))$

14.1.2. Символьное представление структур Крипке

Теперь перейдем к более сложным объектам, а именно к структурам Крипке. Еще раз напомним, что для заданного множества элементарных высказываний AP структурой Крипке называется четверка $\langle S, S_0, R, L\rangle$, где

- S – (конечное) множество состояний;
- $S_0 \subseteq S$ – множество начальных состояний;
- $R \subseteq S \times S$ – отношение переходов;
- $L: S \to 2^{AP}$ – функция разметки.

Для символьного представления структуры Крипке 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 $[\![p]\!] = \{s \in S \mid p \in L(s)\}$. В результате получаем четверку

$$\langle code, \chi_0, \chi_R, \{\chi_p\}_{p\in AP}\rangle.$$

Пример 14.1

Пусть $AP = \{p, q, r\}$. Построим символьное представление простой явно заданной структуры Крипке $M = \langle S, S_0, R, L\rangle$, где [Кар10]

- $S = \{s_0, s_1, s_2\}$;
- $S_0 = \{s_0\}$;
- $R = \{(s_0, s_1), (s_0, s_2), (s_1, s_0), (s_1, s_2), (s_2, s_2)\}$;
- $L = \{s_0 \mapsto \{p, q\}, s_1 \mapsto \{q, r\}, s_2 \mapsto \{r\}\}$.

На рис. 14.1 приведено графическое изображение этой структуры.

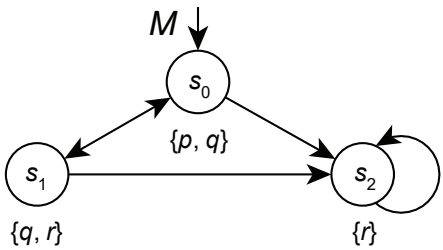


Рис. 14.1 ❖ Пример структуры Крипке

Для представления состояний достаточно двух булевых переменных (назовем их x_1 и x_2). Возможная кодировка приведена в табл. 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	$x_1 \wedge \neg x_2$

Для выбранной кодировки характеристическая функция $\chi_0(x_1, x_2)$ множества начальных состояний S_0 задается 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 табл. 14.4.

Таблица 14.4. Характеристическая функция отношения переходов

Таблица истинности χ_R					**Символьное представление** χ_R	
x_1	x_2	x'_1	x'_2	χ_R	**Совершенная ДНФ**	**Упрощенная формула**
0	0	0	0	0		
0	0	0	1	1	$\neg x_1 \wedge \neg x_2 \wedge \neg x'_1 \wedge \neg x'_2$	$\neg x_1 \wedge \neg x_2 \wedge (x'_1 \oplus x'_2)$
0	0	1	0	1	$\neg x_1 \wedge \neg x_2 \wedge x'_1 \wedge \neg x'_2$	
0	0	1	1	0		
0	1	0	0	1	$\neg x_1 \wedge x_2 \wedge \neg x'_1 \wedge \neg x'_2$	
0	1	0	1	0		$\neg x_1 \wedge x_2 \wedge \neg x'_2$
0	1	1	0	1	$\neg x_1 \wedge x_2 \wedge x'_1 \wedge \neg x'_2$	
0	1	1	1	0		
1	0	0	0	0		
1	0	0	1	0		
1	0	1	0	1	$x_1 \wedge \neg x_2 \wedge x'_1 \wedge \neg x'_2$	$x_1 \wedge \neg x_2 \wedge x'_1 \wedge \neg x'_2$
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		

Характеристические функции, задающие разметку L, определяются согласно табл. 14.5.

Таблица 14.5. Характеристические функции, задающие разметку состояний

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	$\{s_1, s_2\}$	$x_1 \oplus x_2$

□

Следует подчеркнуть, что этот пример является искусственным: обычно модель программы представляется не как граф состояний, а как текст на формализованном языке, и при построении характеристических функций нет нужды выписывать громоздкие таблицы истинности.

14.1.3. Символьные алгоритмы

В основе символьных методов проверки моделей лежат так называемые *символьные алгоритмы* – вычислительные процедуры, оперирующие не с явно заданными множествами объектов и отношениями между ними, а с их символьными представлениями. Сам способ представления нам пока не важен: это могут быть формулы, схемы, диаграммы.

Проиллюстрируем понятие символьного алгоритма на примере анализа достижимости состояний в графе [Кар10].

Пусть $\langle S, S_0, R\rangle$ – ориентированный граф (точнее, система переходов, моделирующая программу). Задача состоит в построении множества всех состояний, достижимых из начальных, т. е. множества $R^*(S_0)$, где R^* – рефлексивно-транзитивное замыкание отношения R. Алгоритм решения этой задачи показан в табл. 14.6.

Таблица 14.6. Обычный и символьный алгоритмы построения множества достижимых состояний

Обычный алгоритм	Символьный 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} := $\chi_{old} \vee \chi_{R(old)}[\mathbf{x}' := \mathbf{x}]$; **while** $\chi_{new} \neq \chi_{old}$ **do** // неэквивалентность χ_{old} := χ_{new}; χ_{new} := $\chi_{old} \vee \chi_{R(old)}[\mathbf{x}' := \mathbf{x}]$ **end**

В левом столбце представлен обычный алгоритм поиска в ширину (в обобщенной теоретико-множественной форме); в правом – его символьный аналог, где операции над множествами и отношениями заменены операциями над их характеристическими функциями. Подстановка (переобозначение переменных) $[\vec{x}' := \vec{x}]$ приводит представление образа множества к исходным переменным. Результатами работы обычного и символьного алгоритмов являются S_{new} и χ_{new} соответственно.

Заметим: сравнение $\chi_{new} \neq \chi_{old}$ в символьном алгоритме построения множества достижимых состояний – это проверка *эквивалентности* двух символьных представлений (например, двух логических формул), а не их посимвольного совпадения. Вообще говоря, это непростая задача: для логики высказываний она является NP-полной [Pap82], а в общем случае – алгоритмически неразрешимой (см. лекцию 8). В конце лекции мы рассмотрим

способ представления булевых функций, для которого эта операция является тривиальной.

14.1.4. Ограниченная проверка моделей и *k*-индукция

Существует упрощенный вариант символьной проверки моделей, в котором строится не замыкание отношения переходов, а итерация ограниченной глубины:

$$R^{(k)} = \bigcup_{i=0}^{k} R^i.$$

Этот подход, известный как *ограниченная проверка модели* (*BMC*, *Bounded Model Checking*), был предложен в начале 2000-х годов Армином Биром[1] [Bie03]. Метод, вообще говоря, не является полным, но позволяет эффективно обнаруживать ошибки.

Пусть проверяемое свойство имеет вид $\mathbf{G}\varphi$. На каждом шаге $i \in \{0, \ldots, k\}$ проверяется выполнимость формулы $\chi_i \wedge \neg\chi_{\varphi_i}$, где χ_i – характеристическая функция множества состояний, достижимых из начальных за i переходов, а χ_{φ_i} – представление свойства φ через переменные, кодирующие состояния на i-м шаге:

$$\chi_i \equiv \chi_0(\vec{x}_0) \wedge (\vec{x}_1 = R(\vec{x}_0)) \wedge \ldots \wedge (\vec{x}_i = R(\vec{x}_{i-1}));$$

$$\chi_{\varphi_i} \equiv \chi_\varphi[\vec{x} := \vec{x}_i].$$

Если формула оказывается выполнимой, в модели есть ошибка.

Пример 14.2

Проиллюстрируем ограниченную проверку моделей примером 2-битного счетчика:

```
unsigned counter:2 = 0; // 2-битный счетчик

do
:: counter++            // инкремент по модулю 4
od

ltl {[](counter != 3)}  // свойство не выполняется
```

[1] Армин Бир (Armin Biere; род. в 1967 году) – европейский исследователь в области формальных методов; автор ограниченной проверки моделей; лауреат IBM Faculty Award (2012) и других премий.

Пусть x и y – соответственно младший и старший биты переменной *counter*. Тогда начальное состояние и отношение переходов задаются формулами

$$\neg x_0 \wedge \neg y_0 \quad \text{и} \quad (x_i = \neg x_{i-1}) \wedge (y_i = x_{i-1} \oplus y_{i-1}).$$

Верифицируемое свойство: $\neg x_i \vee \neg y_i$ (хотя бы один из битов счетчика равен 0).

Для $k = 2$ проверяем $(\neg x_0 \wedge \neg y_0) \wedge \left\{ \begin{array}{c} x_1 = \neg x_0 \\ y_1 = x_0 \oplus y_0 \end{array} \right\} \wedge \left\{ \begin{array}{c} x_2 = \neg x_1 \\ y_2 = x_1 \oplus y_1 \end{array} \right\} \wedge \left[\begin{array}{c} x_0 \wedge y_0 \\ x_1 \wedge y_1 \\ x_2 \wedge y_2 \end{array} \right]$.

Эта формула невыполнима – на заданной глубине ошибки не проявляются. Аналогичная формула для $k = 3$ оказывается выполнимой, что является признаком ошибки.

□

Часто метод BMC применяют в контексте *k-индукции*:

- если для некоторого $k \in \mathbb{N}_0$
  - **базис индукции**: свойство φ истинно во всех состояниях множества $R^{(k)}(S_0) = \bigcup_{i=0}^{k} R^i(S_0)$ и
  - **индуктивный переход**: для любого $X \subseteq S$ из истинности φ во всех состояниях $R^{(k)}(X)$ следует истинность φ во всех состояниях $R^{k+1}(X)$, то
- свойство φ истинно во всех достижимых состояниях.

Проверка базиса индукции – в чистом виде ограниченная проверка модели. Для проверки индуктивного перехода используются SAT- и SMT-решатели.

14.2. Символьная проверка моделей для LTL

Метод символьной 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 символьными. Принимая на вход модель M и LTL-формулу φ, метод осуществляет следующие шаги:

1) построение символьного представления $\mathcal{R}(B_M)$ автомата B_M;
2) построение символьного представления $\mathcal{R}(B_{\neg\varphi})$ автомата $B_{\neg\varphi}$;
3) построение символьного представления $\mathcal{R}(B_M \otimes B_{\neg\varphi})$ автомата $B_M \otimes B_{\neg\varphi}$;
4) проверка языка $\mathcal{L}_{B_M \otimes B_{\neg\varphi}}$ на пустоту.

14.2.1. Символьное представление модели программы

По большому счету, модель M, подаваемая на вход, и так является символьным представлением[1]. Задача состоит в том, чтобы из одного представления построить другое, более подходящее для верификации.

Напомним: *конфигурацией* программы называется пара $\langle l, s\rangle$, где l – состояние управления (метка текущего оператора или множество меток, если речь идет о параллельной программе), а s – состояние данных (означивание переменных).

Пусть задана начальная метка l_0 и предусловие χ_0: начальным состоянием данных может быть любое состояние s, такое что $s \vDash \chi_0$.

Переходы модели можно представить символьно, в виде формул. Например, тот факт, что при исполнении оператора присваивания $x_0 := t$ конфигурация $\langle l, s\rangle$, где l – метка этого оператора, а s – произвольное состояние данных, переходит в конфигурацию $\langle l', s'\rangle$, где l' – метка следующего оператора, а $s' = s[x_0 := s(t)]$, описывается формулой [Кар10]

$$(pc = l) \wedge \left(\bigwedge_{x \in V} (x = s(x))\right) \wedge (pc' = l') \wedge (x_0' = t) \wedge same(V \setminus \{x_0\}).$$

Здесь pc – специальная переменная, хранящая метку текущего оператора (от англ. *program counter* – счетчик программы), $s(x)$ – значение переменной x в состоянии s, pc' и x_0' – переменные, содержащие «следующие» значения pc и x_0 соответственно, V – множество переменных и

$same(V') \equiv \bigwedge_{x \in V'}(x' = x)$, где $V' \subseteq V$.

В табл. 14.7 приведены правила построения символьного представления while-программ. $\mathcal{R}(l\colon P, l')$ есть формула, описывающая отношение переходов программы P с начальной меткой l и конечной меткой l' (считается, что в конце исходной программы есть метка l_ω, сигнализирующая о ее завершении).

Таблица 14.7. Правила построения символьного представления while-программ

(1)	$\mathcal{R}(l\colon \textbf{skip}, l')$	$(pc = l) \wedge (pc' = l') \wedge same(V)$
(2)	$\mathcal{R}(l\colon x := t, l')$	$(pc = l) \wedge (pc' = l') \wedge (x' = t) \wedge same(V \setminus \{x\})$
(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)	$\mathcal{R}(l\colon \textbf{if } B \textbf{ then } l_1\colon P_1 \textbf{ else } l_2\colon P_2 \textbf{ end}, l')$	$((pc = l) \wedge B \wedge (pc' = l_1) \wedge same(V)) \vee \mathcal{R}(l_1\colon P_1, l') \vee$ $((pc = l) \wedge \neg B \wedge (pc' = l_2) \wedge same(V)) \vee \mathcal{R}(l_2\colon P_2, l')$
(5)	$\mathcal{R}(l\colon \textbf{while } B \textbf{ do } l_1\colon P_1 \textbf{ end}, l')$	$((pc = l) \wedge B \wedge (pc' = l_1) \wedge same(V)) \vee \mathcal{R}(l_1\colon P_1, l) \vee$ $((pc = l) \wedge \neg B \wedge (pc' = l') \wedge same(V))$

[1] Инструменты проверки моделей работают не со структурами Крипке, представленными явно, а с описаниями моделей на специальных языках, например Promela (см. лекцию 10) или SMV (см. лекцию 15).

Эти правила интуитивно понятны и не требуют особых пояснений. Заметим лишь, что в них нет аналогов конъюнкции $\bigwedge_{x \in V}(x = s(x))$, имеющейся в формуле, приведенной выше: точная идентификация состояния – это как раз то, от чего в символьных методах верификации хотят уйти. Достаточно иметь неявное описание начальной конфигурации – формулу $(pc = l_0) \wedge \chi_0$ – и представление отношения переходов.

Символьное представление программы может быть пополнено следующим дизъюнктивным членом:

$(pc = l_\omega) \wedge (pc' = l_\omega) \wedge same(V)$, где l_ω – завершающая метка программы.

Без него при синхронной композиции процессов завершение одного из них «остановит» другие (построение символьного представления композиции процессов рассматривается после примера).

Пример 14.3

Обратимся к алгоритму Петерсона взаимного исключения двух процессов (см. лекцию 9). Для простоты и критическая, и некритическая секции состоят из одного оператора **skip**.

```
BGN_i: while true do
  NCS_i: skip; // некритическая секция
  SET_i: flag_i := 1; SET'_i: turn := i;
  TST_i: while (flag_1-i = 1) ∧ (turn = i) do SKP_i: skip end;
  CRS_i: skip; // критическая секция
  RST_i: flag_i := 0
end
END_i:
```

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 символьное представление процессов в соответствии с правилами, приведенными в табл. 14.7. Результат показан в табл. 14.8.

Таблица 14.8. Символьное представление процессов

Метка	Символьное представление
BGN_i	$((pc_i = BGN_i) \wedge true \wedge (pc'_i = NCS_i) \wedge same(V)) \vee$
NCS_i	$((pc_i = NCS_i) \wedge (pc'_i = SET_i) \wedge same(V)) \vee$
SET_i	$((pc_i = SET_i) \wedge (pc'_i = SET_i) \wedge (flag'_i = 1) \wedge same(V \setminus \{flag_i\})) \vee$
SET'_i	$((pc_i = SET'_i) \wedge (pc'_i = TST_i) \wedge (turn' = i) \wedge same(V \setminus \{turn\})) \vee$
TST_i	$((pc_i = TST_i) \wedge ((flag_{1-i} = 1) \wedge (turn = i)) \wedge (pc'_i = SKP_i) \wedge same(V)) \vee$
SKP_i	$((pc_i = SKP_i) \wedge (pc'_i = TST_i) \wedge same(V)) \vee$
	$((pc_i = TST_i) \wedge \neg((flag_{1-i} = 1) \wedge (turn = i)) \wedge (pc'_i = CRS_i) \wedge same(V))$

Таблица 14.8 *(окончание)*

Метка	Символьное представление
CRS_i	$((pc_i = CRS_i) \wedge (pc_i' = RST_i) \wedge same(V)) \vee$
RST_i	$((pc_i = RST_i) \wedge (pc_i' = BGN_i) \wedge (flag_i' = 0) \wedge same(V \setminus \{flag_i\})) \vee$
	$((pc_i = BGN_i) \wedge \neg true \wedge (pc_i' = END_i) \wedge same(V)) \vee$
END_i	$((pc_i = END_i) \wedge (pc_i' = END_i) \wedge same(V))$

□

Если все переменные имеют конечные домены (как, например, в алгоритме Петерсона), можно воспользоваться двоичной кодировкой состояний. Самый простой способ – упорядочить переменные и сконкатенировать соответствующие им двоичные векторы.

Пусть $P = P_1 \,||\, \cdots \,||\, P_n$ – параллельная программа, в которой для каждого процесса P_i, где $i \in \{1, \ldots, n\}$, определено символьное представление отношения переходов $\mathcal{R}(P_i)$[1]. Как построить $\mathcal{R}(P)$ – символьное представление программы P? Ответ зависит от используемой модели вычислений (см. лекцию 9):

- $\mathcal{R}(P) = \bigwedge_{i=1}^{n} \mathcal{R}(P_i)$ для синхронной композиции (прямого произведения);
- $\mathcal{R}(P) = (p \in \{1, \ldots, n\}) \wedge \left(\bigwedge_{i=1}^{n} \begin{Bmatrix} (p = i) \to \mathcal{R}(P_i) \\ (p \neq i) \to (pc_i' = pc_i) \end{Bmatrix} \right)$ для асинхронной.

Переменная p в представлении асинхронной композиции – это номер процесса, выбранного планировщиком для исполнения.

В случае синхронной композиции имеется нюанс, связанный с ограничениями *same*. Правила, как они заданы сейчас, не рассчитаны на параллельную работу операторов. Так, синхронное исполнение $x := x + 1$ и $y := y + 1$ описывается заведомо ложной формулой

$$((x' = x + 1) \wedge (y' = y)) \wedge ((y' = y + 1) \wedge (x' = x)).$$

Чтобы избежать этой аномалии, используются другие ограничения:

- для каждой переменной x
  - составляется условие $def(x)$, сигнализирующее о присваивании в эту переменную: обычно оно имеет вид $(pc_{i_1} = l_{i_1}) \vee \ldots \vee (pc_{i_k} = l_{i_k})$;
  - к символьному представлению отношения переходов композиции добавляется конъюнктивный член

    $$\neg def(x) \to (x' = x).$$

Отсутствие одновременных присваиваний в переменную x выражается формулой

$$\bigwedge_{i \neq j} \neg\left(def_i(x) \wedge def_j(x)\right),$$

где $def_i(x)$ – условие присваивания в x внутри процесса с номером i.

[1] Очевидно, что у каждого процесса счетчик *pc* свой.

14.2.2. Символьное представление модели требований

На прошлой лекции мы рассмотрели метод синтеза автомата Бюхи по формуле LTL. Вместо того чтобы сконструировать автомат в явном виде, можно построить его символьное представление. Суть метода от этого не меняется (в качестве упражнения: предложите способ кодировки состояний автомата, упрощающий «программирование» условий реализации обязательств).

14.2.3. Проверка соответствия модели программы и модели требований

Символьное представление синхронной композиции $B_M \otimes B_{\neg\varphi}$ строится по правилам, указанным в разделе 14.2.1. Отсутствие допускающих циклов в автомате проверяется путем вычисления *оболочки компонент сильной связности* (символьно проще вычислить оболочку компонент, чем их совокупность).

Оболочкой КСС ориентированного графа (системы переходов) называется множество достижимых состояний, входящих в нетривиальные КСС. Напомним: компонента называется *нетривиальной*, если она содержит хотя бы один переход, т. е. не является одиночным состоянием без петель.

Нас интересуют не все КСС, а только те, которые содержат допускающие состояния автомата (будем называть такие компоненты *допускающими*): в автомате есть допускающий цикл тогда и только тогда, когда оболочка допускающих компонент не пуста.

Пусть S – множество состояний синхронной композиции $B_M \otimes B_{\neg\varphi}$, $S_0 \subseteq S$ – множество начальных состояний, $R \subseteq S \times S$ – отношение переходов. Приведенный ниже алгоритм вычисления оболочки допускающих КСС базируется на работах [Roz10] и [Kes98]. Для удобства мы считаем, что $B_M \otimes B_{\neg\varphi}$ содержит одно множество допускающих состояний $\mathcal{F}$.

```
S_old := ∅;
S_new := R*(S_0);                    // оболочка - все достижимые состояния
while S_new ≠ S_old do
  S_old := S_new;
  S_new := R*(S_new ∩ F);            // оставляются состояния, достижимые
                                     // из допускающих состояний оболочки
  while S_new ≠ (S_new ∩ R(S_new)) do
    S_new := S_new ∩ R(S_new)        // оставляются состояния, имеющие
  end                                // предшественников в оболочке
end;
verdict := (S_new = ∅)               // если оболочка пуста, допускающих
                                     // циклов нет (модель корректна)
```

Сначала вычисляется множество достижимых состояний $R^*(S_0)$ – это своего рода «протооболочка», охватывающая все достижимые КСС, в том числе

тривиальные и недопускающие. Далее в цикле (пока возможно) эта протооболочка сужается:

- удаляются состояния, недостижимые из ее допускающих состояний (такие состояния заведомо не принадлежат допускающим КСС);
- удаляются состояния, в которые нельзя попасть из других состояний протооболочки (это избавляет от состояний, не входящих в КСС, а также тривиальных КСС).

Для построения контрпримера можно использовать следующий алгоритм, находящий кратчайший путь между состояниями из множества S_{source} и состояниями из множества S_{target} (предполагается, что путь существует).

```
π := ∅;                                  // сначала путь пуст
S_start := S_source;                     // пока фронт начальных состояний
while (S_start ∩ S_target) = ∅ do        // не достиг конечных состояний
  S_end := R^-1(S_target);
  while (S_start ∩ S_end) = ∅ do         // поиск в ширину от конца к началу
    S_end := R^-1(S_end)                 // до достижения начальных состояний
  end;
  s := choose(S_start ∩ S_end);          // выбор состояния для перехода
  π := π · {s};                          // удлинение пути
  S_start := R(s)                        // продвижение фронта начальных состояний
end
```

Алгоритм применяется к множествам S_0 и $S_{new} \cap \mathcal{F}$ (см. предыдущий алгоритм). Далее выбирается произвольное состояние $s \in S_{start} \cap (S_{new} \cap \mathcal{F})$, после чего алгоритм применяется к множествам $R(s)$ и $\{s\}$ (так ищется цикл через s).

14.3. Двоичные решающие диаграммы

Рассмотрим важный для многих приложений способ представления булевых функций – *двоичные решающие диаграммы* (*BDD*, *Binary Decision Diagrams*).

BDD – это корневой ориентированный ациклический граф, нелистовые (внутренние) вершины которого помечены переменными. Из каждой нелистовой вершины исходят две дуги: одна соответствует значению 0 переменной, помечающей эту вершину; другая – значению 1. Листовых вершин две: $\boxed{0}$ и $\boxed{1}$.

Структуры вроде двоичных решающих диаграмм исследуются уже много лет[1], однако впервые их потенциал был раскрыт Рэнделом Брайан-

[1] Первой публикацией по двоичным решающим диаграммам, не считая пионерских работ Виктора Ивановича Шестакова (1907–1987) и Клода Шеннона по символьному анализу релейных схем, является статья Ли [Lee59]. Важной вехой стала работа Шелдона Акерса [Ake78], в которой был введен термин BDD. Также можно отметить русскоязычную статью эстонского исследователя Раймунда Убара [Уба76], описывающую схожую структуру данных (названную *альтернативным графом*) и ее применение к тестированию цифровых схем.

том[1] [Bry86], предложившим фиксировать порядок переменных и устранять избыточность путем «склеивания» одинаковых подграфов (почему это важно, мы увидим чуть позже).

14.3.1. Синтаксис и семантика диаграмм

Дадим формальное определение.

Зафиксируем множество переменных V. *Двоичной решающей диаграммой*, или просто *диаграммой*, над множеством V называется пятерка $\langle N \cup \{\boxed{0}, \boxed{1}\}, n_0, L, low, high\rangle$, где

- $N \cup \{\boxed{0}, \boxed{1}\}$ – множество вершин ($N \cap \{\boxed{0}, \boxed{1}\} = \varnothing$);
- $n_0 \in N \cup \{\boxed{0}, \boxed{1}\}$ – корень;
- $L: N \to V$ – функция разметки;
- $low: N \to N \cup \{\boxed{0}, \boxed{1}\}$ – отображение, задающее *low*-дуги;
- $high: N \to N \cup \{\boxed{0}, \boxed{1}\}$ – отображение, задающее *high*-дуги.

Вершины $\boxed{0}$ и $\boxed{1}$ называются *листовыми*; вершины множества N – *внутренними*.

Для внутренней вершины n введем следующие обозначения:

- $then(n)$ – поддиаграмма с корнем $high(n)$;
- $else(n)$ – поддиаграмма с корнем $low(n)$.

Через $root(F)$ будем обозначать корень диаграммы F. Если $root(F) \in N$, то

- $L(F) \equiv L(root(F))$;
- $then(F) \equiv then(root(F))$;
- $else(F) \equiv else(root(F))$.

Диаграммы с корнями $\boxed{0}$ и $\boxed{1}$ называются *тривиальными* и обозначаются *ZERO* и *ONE* соответственно.

Семантика диаграммы F, т. е. представляемая ей булева функция (обозначение: $[\![F]\!]$), определяется следующим образом:

- если $root(F) = \boxed{0}$, то $[\![F]\!] \equiv 0$ – константа 0;
- если $root(F) = \boxed{1}$, то $[\![F]\!] \equiv 1$ – константа 1;
- иначе $[\![F]\!] \equiv (L(F) \wedge [\![then(F)]\!]) \vee (\neg L(F) \wedge [\![else(F)]\!])$.

Чтобы вычислить значение функции, нужно начиная с корня в зависимости от значений переменных, помечающих вершины, двигаться по *low*- или *high*-дугам до тех пор, пока не достигнем листовой вершины: очевидно, $\boxed{0}$ соответствует значению 0, а $\boxed{1}$ – 1.

---

[1] Рэндел Брайант (Randal Bryant; род. в 1952 году) – американский ученый-информатик; наиболее известен своей работой по представлению булевых функций с помощью BDD (1986); лауреат многочисленных премий, в том числе премии Париса Канеллакиса за теоретические и практические достижения (1998) и премии Фила Кауфмана (2009).

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 14.4

Диаграмма, показанная на рис. 14.2 [Yua06], соответствует следующей ДНФ:

$$(\neg x_1 \wedge \neg y_1 \wedge z_1) \vee (\neg x_1 \wedge y_1 \wedge z_1) \vee (x_1 \wedge \neg y_2 \wedge z_1) \vee (x_1 \wedge y_2 \wedge \neg z_2) \vee (x_1 \wedge y_2 \wedge z_2).$$

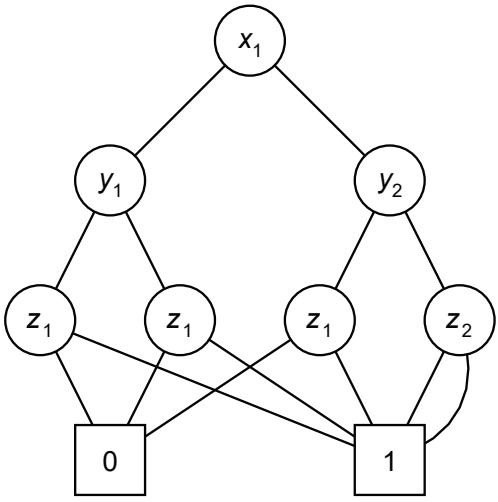


Рис. 14.2 ❖ Пример двоичной решающей диаграммы

□

14.3.2. Сокращенные упорядоченные диаграммы

Данное выше определение BDD допускает неоднозначность представления функций, что затрудняет проверку их эквивалентности. В этом разделе мы рассмотрим специальный вид диаграмм, в котором за счет фиксации порядка переменных и удаления избыточности этот недостаток устранен (это именно то, что было предложено Брайантом).

Диаграмма называется *упорядоченной*, если на множестве переменных задан линейный порядок $\prec$ и для любых двух нелистовых вершин n и n' справедливо соотношение

$L(n) \prec L(n')$, если n – предок n'.

Диаграмма называется *сокращенной*, если в ней отсутствуют *избыточные вершины* и *изоморфные подграфы*.

Вершина n называется *избыточной*, если $low(n) = high(n)$.

Две диаграммы F и G называются изоморфными, если

- либо $root(F) = root(G)$[1],
  - в частности оба корня – $\boxed{0}$ или оба корня – $\boxed{1}$;
- либо $L(F) = L(G)$, при этом
  - $then(F)$ и $then(G)$ изоморфны,
  - $else(F)$ и $else(G)$ изоморфны.

Любая диаграмма может быть приведена к сокращенной форме путем применения следующих двух правил (пока хотя бы одно из них применимо)[2]:

1) если в диаграмме есть два изоморфных подграфа:
   a) выбрать один из них и перенаправить ведущие в него дуги в соответствующие вершины другого подграфа;
   b) удалить выбранный подграф;
2) если в диаграмме есть избыточная вершина:
   a) перенаправить ведущие в нее дуги в ее последователя (он один);
   b) удалить вершину.

Когда идет речь о двоичных решающих диаграммах, обычно имеются в виду *сокращенные упорядоченные двоичные решающие диаграммы* (*ROBDD*, *Reduced Ordered Binary Decision Diagrams*). Для краткости мы будем называть их просто диаграммами, или ROBDD.

ROBDD обладают важным свойством: представление с их помощью функций является *каноническим* – две функции равны тогда и только тогда, когда их диаграммы совпадают (являются изоморфными). Предполагается, что в сравниваемых диаграммах используется единый порядок переменных.

Это свойство можно также использовать для проверки выполнимости и общезначимости формул логики высказываний (см. лекцию 8): в первом случае ROBDD должна отличаться от *ZERO*, во втором – совпадать с *ONE*.

Пример 14.5

Применим правила сокращения к диаграмме из предыдущего примера. На рис. 14.3 слева показана исходная диаграмма; в центре – результат применения правила 1; справа – результат применения правила 2.

В исходной диаграмме есть три взаимно изоморфных подграфа: их корни помечены переменной z_1, *low*-дуги ведут в $\boxed{0}$, *high*-дуги – в $\boxed{1}$. Два из них удаляются с перенаправлением входящих дуг в вершины оставшегося подграфа.

В полученной диаграмме (в центре) есть две избыточные вершины: одна помечена переменной y_1, вторая – z_2. Эти вершины удаляются (входившие в них дуги перенаправляются в их последователей: z_1 и $\boxed{1}$ соответственно).

---

[1] Здесь предполагается, что диаграммы с одинаковыми корнями обязательно совпадают.

[2] Имеется альтернативный вариант второго правила: если $high(n) = \boxed{0}$, то все входящие в n дуги перенаправляются в $low(n)$, а вершина n удаляется. Получаемые диаграммы называются *двоичными решающими диаграммами с подавлением нулей* (*ZDD*, *Zero-suppressed Decision Diagrams*). Они были предложены Шиничи Минато (Shin-ichi Minato) в 1993 году.

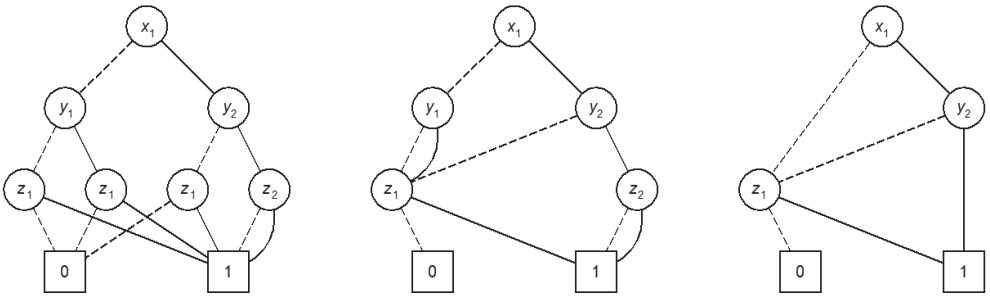


Рис. 14.3 ❖ Построение сокращенной двоичной решающей диаграммы

□

Отметим, что размер ROBDD существенно зависит от порядка переменных. Известны эвристики для выбора «хорошего» порядка, но в этой книге они не рассматриваются.

14.3.3. Синтез диаграмм и манипуляции с ними

Построение двоичной решающей диаграммы по булевой функции базируется на *разложении Шеннона*[1]:

$$f(x_1, \ldots, x_n) = (x_i \wedge f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)) \vee (\neg x_i \wedge f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)).$$

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, для которой
  - $L(F) = x$;
  - $then(F) = T$;
  - $else(F) = E$.

Пусть f и g – булевы функции, а $\circ$ – некоторая бинарная операция (конъюнкция, дизъюнкция или другая). Применим разложение Шеннона по переменной x:

$$f \circ g = (x \wedge (f_{|x=1} \circ g_{|x=1})) \vee (\neg x \wedge (f_{|x=0} \circ g_{|x=0})).$$

Используя это равенство, определим процедуру *Apply*, конструирующую диаграмму $F \circ G$ (функции $f \circ g$) по диаграммам F и G (функций f и g соответственно) [Bry86]:

$$Apply(F \circ G) = Reduce(Compose(x, Apply(F_{|x=1} \circ G_{|x=1}), Apply(F_{|x=0} \circ G_{|x=0}))).$$

---

[1] По сути, это условный оператор (мультиплексор «2 в 1»): $x_i \; ? \; f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) : f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$.

Запись $D_{|x=\sigma}$ означает диаграмму, полученную из D путем следующего преобразования: для каждой вершины n, такой что $L(n) = x$, дуги, входящие в n, перенаправляются в

$$\begin{cases} low(n), & \text{если } \sigma = 0 \\ high(n), & \text{если } \sigma = 1 \end{cases}.$$

В качестве переменной x, по которой производится разложение в определении *Apply*, используется «минимальная» из переменных, помечающих корни F и G. В этом случае имеет место равенство

$$D_{|x=\sigma} = \begin{cases} then(D), & \text{если } x = L(D) \text{ и } \sigma = 1 \\ else(D), & \text{если } x = L(D) \text{ и } \sigma = 0. \\ D, & \text{если } x \prec L(D) \end{cases}$$

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.

Мы рассмотрели синтез ROBDD для бинарных операций. Построить диаграмму отрицания некоторой функции можно, перенаправив дуги из вершины $\boxed{0}$ в вершину $\boxed{1}$ и наоборот.

Важный момент, касающийся программной реализации ROBDD. Для хранения узлов диаграмм обычно используют таблицу, ключом в которой выступает тройка (x, T, E), где x – переменная, а T и E – указатели на дочерние узлы. Процедура $Compose(x, T, E)$ реализована следующим образом:

- если $T = E$, возвратить T (это *Reduce*);
- проверить, есть ли в таблице узел с ключом (x, T, E):
  - если нет, создать узел (рекурсивно) и поместить его в таблицу;
- возвратить указатель на узел.

Такая организация, называемая *структурным хешированием* (*structural hashing*), не только ускоряет построение ROBDD, но и упрощает сравнение: две диаграммы изоморфны тогда и только тогда, когда совпадают указатели на их корни.

Кроме того, для оптимизации манипуляций с ROBDD применяется кеширование диаграмм часто используемых формул.

Пример 14.6

Рассмотрим по шагам построение ROBDD для функции $(x_1 \wedge x_2) \vee \neg x_3$. Предположим, что переменные упорядочены в порядке возрастания индексов: $x_1 \prec x_2 \prec x_3$.

Применяя рекурсивно процедуру *Apply*, получим:

- $Apply((x_1 \wedge x_2) \vee \neg x_3) = Compose(x_1, Apply(x_2 \vee \neg x_3), Apply(\neg x_3))$:
  - $Apply(x_2 \vee \neg x_3) = Compose(x_2, ONE, Apply(\neg x_3))$:
    - $Apply(\neg x_3) = Compose(x_3, ZERO, ONE)$;
  - $Apply(\neg x_3) = Compose(x_3, ZERO, ONE)$:
    - возвращается указатель на ранее построенный узел.

Диаграмма синтезируется в обратном порядке: сначала создается узел для $\neg x_3$, потом – для $x_2 \vee \neg x_3$, наконец, для $(x_1 \wedge x_2) \vee \neg x_3$ (при этом переиспользуется узел для $\neg x_3$). Процесс изображен на рис. 14.4.

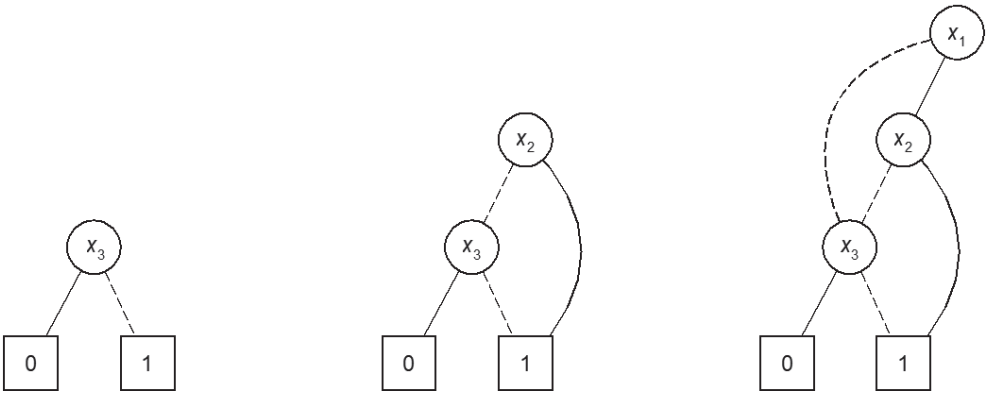


Рис. 14.4 ❖ Синтез двоичной решающей диаграммы по формуле

□

Пример 14.7

Пусть F – ROBDD функции $x_2 \vee \neg x_3$ (см. рис. 14.4, диаграмма в центре), а G – ROBDD функции x_3. Построим диаграмму $F \wedge G$:

$$Apply(F \wedge G) = Compose(x_2, Apply(F_{|x_2=1} \wedge G_{|x_2=1}), Apply(F_{|x_2=0} \wedge G_{|x_2=0})) = \cdots.$$

Шаги процедуры показаны на рис. 14.5.

□

Отметим, что квантификация (∃), используемая для построения образа и прообраза бинарного отношения, допускает эффективную реализацию с помощью двоичных решающих диаграмм (см. упражнения). Переобозначение переменных при вычислении образа (см. символьный алгоритм в табл. 14.6) делается легко, если порядок переменных в $\vec{x}'$ соответствует порядку переменных в $\vec{x}$.

Существует несколько свободно распространяемых библиотек для работы с двоичными решающими диаграммами. Одна из них – CUDD [Cudd], используемая в инструментах проверки моделей NuSMV [NuSMV] и nuXmv [Cav14].

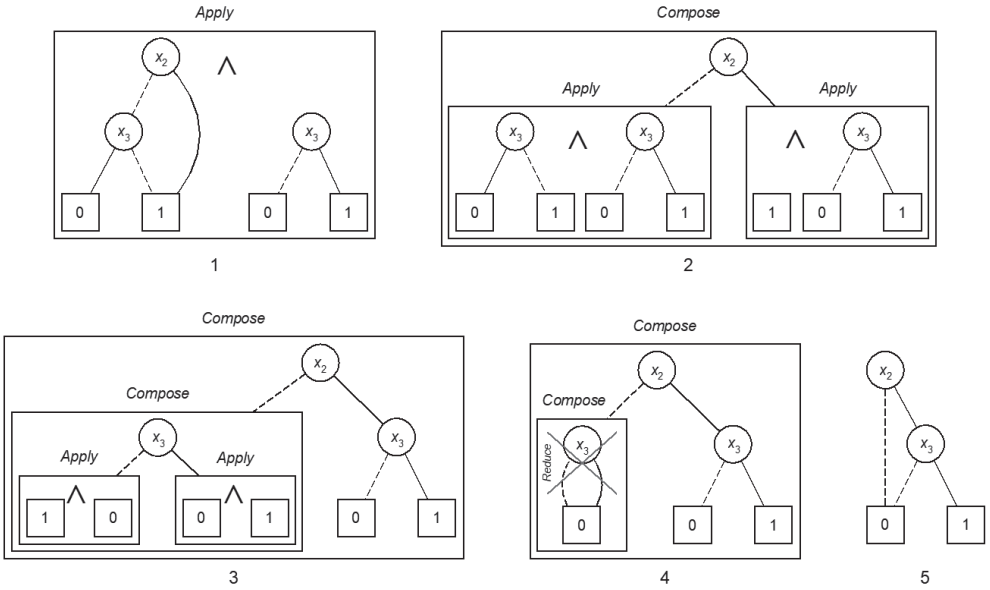
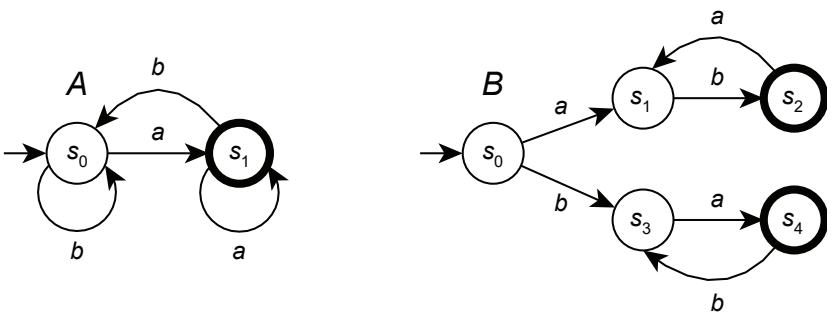


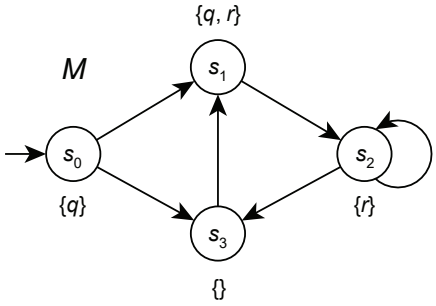
Рис. 14.5 ❖ Композиция двоичных решающих диаграмм

14.4. Вопросы и упражнения

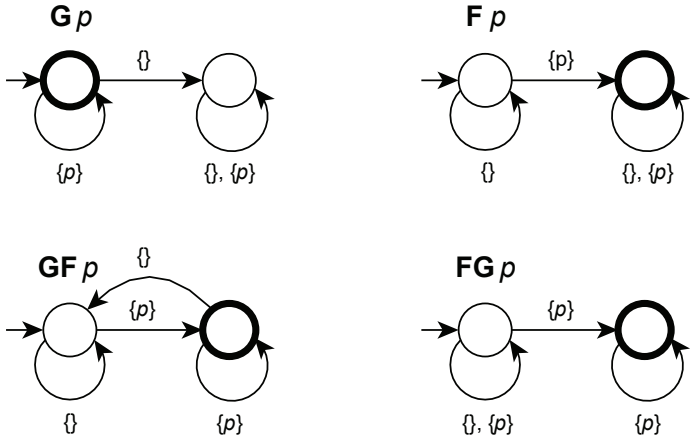
1. В чем отличие символьных методов проверки моделей от классических?
2. Постройте символьные представления для следующих конечных автоматов (под символьным представлением автомата понимаются логические формулы, характеризующие множество состояний, множество начальных состояний, множество допускающих состояний, функцию переходов):



3. Постройте символьное представление для следующей структуры Крипке:



4. Постройте символьные представления для следующих автоматов Бюхи:



5. Постройте символьное представление программы, реализующей алгоритм Евклида (программы *P* из лекции 1).
6. Постройте символьное представление параллельной программы $P = P_1 \,||\, P_2$, где процессы P_1 и P_2 определяются, как показано ниже, а параллелизм предполагается асинхронным [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 *A*0 и *A*1, и постройте символьное представление синхронной композиции процессов P_1 и P_2.

8. Задайте двоичную кодировку состояний (управления и данных) для алгоритма Петерсона и постройте характеристическую функцию отношения переходов.
9. Постройте двоичные решающие диаграммы (ROBDD) для всех возможных булевых функций двух переменных.
10. Предложите способ реализации следующих операций на ROBDD:
    a) $f(x_1, \ldots, x_n)_{|x_i=\sigma} = f(x_1, \ldots, x_{i-1}, \sigma, x_{i+1}, x_n)$, где $\sigma \in (0, 1)$;
    b) $\exists x_i f(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, x_n) \vee f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, x_n)$;
    c) $\forall x_i f(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, x_n) \wedge f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, x_n)$.
11. Определите заключительные ветви рекурсии процедуры *Apply* для следующих булевых операций:
    a) $\oplus$ (сумма по модулю 2);
    b) $\rightarrow$ (импликация).
12. 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) $x_1 \prec x_2 \prec x_3 \prec y_1 \prec y_2 \prec y_3$.
13. 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).
14. Постройте двоичную решающую диаграмму для 3-битного сумматора.
15. Докажите, что алгоритмическая сложность $Apply(F \circ G)$ составляет $O(|F| \cdot |G|)$.
16. Расширьте символьный алгоритм построения оболочки достижимых КСС на случай обобщенных автоматов Бюхи.
17. Напишите программу (на удобном вам языке программирования), синтезирующую ROBDD по заданной формуле логики высказываний.

Лекция 15

Инструменты символьной проверки моделей семейства SMV

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

А. Маслоу

Рассматриваются основные возможности инструментов семейства SMV (Symbolic Model Verifier) и лежащего в их основе языка моделирования: представление параллельных систем в виде иерархически организованных модулей, спецификация требований на языке темпоральной логики и собственно символьная проверка моделей, базирующаяся на двоичных решающих диаграммах (BDD) и техниках проверки выполнимости (SAT). Описанные средства могут быть использованы на практике для разработки и анализа компьютерных протоколов, моделей программ и цифровой аппаратуры.

15.1. Введение в язык SMV

Инструменты семейства SMV (Symbolic Model Verifier) – как первоначальный вариант CMU SMV [Mcm92], разработанный Кеннетом Макмилланом в Университете Карнеги–Меллона (CMU, Carnegie Mellon University), так и сравнительно новые реализации NuSMV [NuSMV] и nuXmv [Cav14], созданные

в Центре Бруно Кесслера (FBK, Fondazione Bruno Kessler), – предназначены для символьной проверки моделей параллельных систем. Все перечисленные инструменты реализуют *неограниченную* (*unbounded*) проверку с использованием двоичных решающих диаграмм (BDD-based model checking); в дополнение к этому NuSMV и nuXmv могут осуществлять *ограниченную* (*bounded*) проверку с использованием SAT-решателей (SAT-based model checking). Входной язык (его обычно так и называют – SMV) является (плюс-минус) общим для всех инструментов семейства[1]. В этой лекции мы будем ориентироваться на NuSMV версии 2.6 [Cav15].

Язык SMV близок по духу к языкам описания аппаратуры (HDL, Hardware Description Languages), таким как Verilog и VHDL [Har12]. Основной структурной единицей в нем является *модуль*, воплощающий в себе конечный автомат. SMV-модель есть иерархическая система *экземпляров* модулей – *синхронная* композиция конечных автоматов. В отличие от Promela (см. лекцию 10), где допускается динамическое порождение процессов, конфигурация системы в SMV является статической.

15.1.1. Типы данных, переменные и выражения

Базовыми типами данных языка SMV являются **`boolean`** (значения истинности), **`integer`** (целые числа) и **`word`** (слова, или двоичные векторы, интерпретируемые как знаковые или беззнаковые числа) – см. табл. 15.1.

Составных типа два: *массивы* и *множества*. Кроме того, в языке есть *перечисления*.

Таблица 15.1. Базовые типы данных языка SMV

Тип данных	Множество значений	Битовая длина
`boolean`	{FALSE, TRUE}	1
`integer`	[INT_MIN + 1, INT_MAX]	зависит от реализации, обычно 32
`word`[*n*] **`unsigned word`**[*n*]	$[0, 2^n - 1]$	$n \geq 1$
`signed word`[*n*]	$[-2^{n-1}, 2^{n-1} - 1]$	$n \geq 1$

Тип **`boolean`** имеет два значения: `FALSE` и `TRUE` (в отличие от Promela, это не то же самое, что `0` и `1`). Литералами типа **`integer`** являются непустые последовательности десятичных цифр, перед которыми может ставиться минус (обратите внимание: запись целых чисел возможна только в десятичной форме). Синтаксис констант типа **`word`** приведен ниже:

`0[s/u]base[width]_value`

[1] Небольшое отличие nuXmv от NuSMV (в плане языка) – поддержка типа **real** [Boz19]. В nuXmv применяется SMT-решатель MathSAT5 [MatS], что позволяет верифицировать системы с бесконечным числом состояний.

Смысл обозначений здесь следующий:

- *s/u* – признак знаковости/беззнаковости слова:
  - *s* – знаковое слово (*signed*);
  - *u* – беззнаковое слово (*unsigned*);
  - отсутствие признака равносильно наличию *u*;
- *base* – система счисления, используемая в записи значения:
  - *b* (или *B*) – двоичная (*binary*);
  - *o* (или *O*) – восьмеричная (*octal*);
  - *d* (или *D*) – десятичная (*decimal*);
  - *h* (или *H*) – шестнадцатеричная (*hexadecimal*);
- *width* ≥ 1 – разрядность слова (битовая длина):
  - обязательна при использовании десятичной системы;
  - для других систем может выводиться из записи значения;
- *value* – запись значения в заданной системе счисления:
  - допускается разделитель _ (символ подчеркивания);
  - значение должно принадлежать диапазону:
    - $[0, 2^{width-1} - 1]$, если указаны *s* и *d*;
    - $[0, 2^{width} - 1]$ иначе.

Пример 15.1

Ниже приведены примеры слов с пояснениями. Обратите внимание: однострочные комментарии в SMV начинаются с `--`, а многострочные обрамляются `/--` и `--/`.

```
0sb5_10111    /-- знаковое (s) слово длины 5
                  в двоичной (b) записи --/
0uo6_37       /-- беззнаковое (u) слово длины 6
                  в восьмеричной (o) записи --/
0d11_9        /-- беззнаковое слово длины 11
                  в десятичной (d) записи --/
0sh12_a9      /-- знаковое (s) слово длины 12
                  в шестнадцатеричной (h) записи --/

0ud8_11       -- беззнаковое слово длины 8
0ub8_1011     -- старшие биты заполняются 0
0b_00001011   -- длина не указывается
0b_0000_1011  -- используется разделитель _
0h_0b         -- длина не указывается
0h8_b         -- старшие биты заполняются 0
```

□

Синтаксис объявления переменных заимствован у языка Pascal:

```
x : T;  -- объявление переменной x типа T
```

Идентификатор (имя переменной или другой сущности модели) есть любая последовательность символов из алфавита {`A–Z a–z 0–9 _ $ # -`}, начи-

нающаяся с латинской буквы или символа подчеркивания (и отличная от ключевого слова).

Пример 15.2

Рассмотрим примеры деклараций переменных.

```
flag  : boolean;  -- булева переменная
index : integer;  -- целочисленная переменная
turn  : word[1];  -- беззнаковое слово длины 1
```

□

Перечислимый тип определяется, как показано ниже:

```
{c_1, ..., c_n}  -- определение перечислимого типа
```

Здесь c_i – разные идентификаторы (символьные константы).

Если быть точным, так определяется *символьный перечислимый тип*. В SMV также есть *целочисленные* и *смешанные* (*integer-and-symbolic*) *перечисления*: в первом случае в фигурных скобках указываются целые числа, во втором – числа вперемежку с символьными константами.

Пример 15.2

Ниже объявляется переменная `status` перечислимого типа (статус процесса).

```
-- объявление переменной, хранящей статус процесса
status : { stopped, running, waiting };
```

□

Заметим, что в SMV нет конструкции именования типа (наподобие **typedef** в языке C): чтобы объявить несколько переменных одного перечислимого или иного нетривиального типа, придется дублировать код.

Перейдем к составным типам данных: массивам и множествам.

При определении типа-массива указываются диапазон значений индекса и тип элемента. В примере ниже это *min*..*max* и *T* соответственно (*min* и *max* – целые числа, причем $min \le max$):

```
array min..max of T  -- определение массива
```

Значения-массивы записываются путем перечисления (через запятую) значений элементов, при этом вся совокупность значений заключается в квадратные скобки.

Пример 15.3

Приведем несколько типов-массивов вместе с примерами их значений.

```
/-- массив из 4 элементов булева типа;
    пример значения: [ FALSE, TRUE, FALSE, TRUE ] --/
array 0..3 of boolean

/-- массив из 3 элементов перечислимого типа;
    пример значения: [ running, waiting, stopped ] --/
```

```
array -1..1 of { stopped, running, waiting }

/-- массив из 3 элементов, каждый из которых
    является массивом из 2 целых чисел;
    пример значения: [ [1, 9], [-4, 5], [8, 0] ] --/
array 1..3 of array 1..2 of integer
```

□

Последний тип данных, который мы рассмотрим, – это множество. Отметим, что этот тип можно использовать в ограниченном числе мест (в частности, нельзя определять переменные-множества). В зависимости от типа элемента множества бывают булевыми, целочисленными, символьными и смешанными (SMV не допускает множеств слов, множеств массивов и тем более множеств множеств).

Множества записываются в фигурных скобках (в отличие от массивов, порядок записи их элементов не важен). Для записи множеств также используются константные диапазоны (вроде тех, что задают границы значений индекса в массивах) и операция **union**.

SMV неявно приводит одиночные значения к множествам-синглтонам: например, `0 union 1` – это то же самое, что и `{0} union {1}`, т. е. `{0, 1}`.

Пример 15.4

Ниже представлены примеры множеств с комментариями.

```
{FALSE, TRUE}                    -- множество из двух булевых значений
FALSE union TRUE                 -- то же самое
{1, 2, 3, 4}                     -- множество из четырех целых чисел
1 union {2} union {3, 4}         -- то же самое
1..4                             -- то же самое
{stopped, running, waiting}      -- множество из трех констант
stopped union {running, waiting} -- то же самое
```

□

Поддерживаемые операции (в порядке убывания приоритета) и их ассоциативность (способ расстановки скобок) приведены в табл. 15.2 [Cav15].

Операции имеют понятную семантику, отметим лишь некоторые особенности:

- в языке SMV отсутствует неявное приведение значений между типами **boolean**, **integer**, **unsigned word**[*n*] и **signed word**[*n*] (в любой комбинации);
- бинарные операции над словами (кроме сдвига и конкатенации) требуют, чтобы слова имели одинаковую знаковость и разрядность[1];
- побитовые операции не применимы к целым числам;
- результат операции выбора битов или конкатенации – всегда беззнаковое слово (другие операции над словами знаковость не меняют).

[1] Для арифметических операций есть послабление: операнд меньшей разрядности расширяется (знаково или беззнаково – в зависимости от типа) до разрядности другого операнда.

Таблица 15.2. Операции языка SMV, их приоритет и ассоциативность

Операции	Ассоциативность	Комментарий
`[] [ : ]`	слева направо	Индексирование массива, выбор битов слова
`!`	справа налево	Логическое и побитовое отрицание
`::`	слева направо	Конкатенация слов
`-`	справа налево	Унарный минус
`* / mod`	слева направо	Умножение, деление, взятие остатка от деления
`+ -`	слева направо	Сложение, вычитание
`<< >>`	слева направо	Сдвиг влево, сдвиг вправо
`union`	слева направо	Объединение множеств
`in`	слева направо	Вложение множеств, принадлежность элемента множеству
`= != < <= > >=`	слева направо	Сравнение на равно / не равно и меньше/больше
`&`	слева направо	Логическое и побитовое И
`| xor xnor`	слева направо	ИЛИ, исключающее ИЛИ, исключающее ИЛИ-НЕ
`? :`	слева направо	Условная операция
`<->`	слева направо	Логическая и побитовая эквивалентность
`->`	справа налево	Логическая и побитовая импликация

Другие тонкости: в записи конкатенации слева пишутся старшие биты, справа – младшие; при выборе битов слова сначала указывается номер старшего бита, потом – младшего; биты нумеруются начиная с 0.

Пример 15.5

Приведем примеры, иллюстрирующие операции выбора битов и конкатенации.

```
0sb8_10101100[4:1]    -- 0ub4_0110 (результат беззнаковый)
0ub8_10101100[4:1]    -- результат тот же
0sb4_1010::0sb4_1100  -- 0ub8_1010_1100 (результат беззнаковый)
0sb4_1010::0ub4_1100  -- результат тот же
0ub4_1010::0sb4_1100  -- результат тот же
0ub4_1010::0ub4_1100  -- результат тот же
```

□

15.1.2. Встроенные функции

В SMV есть встроенные функции для приведения типов и компактной записи некоторых операций (см. табл. 15.3).

Таблица 15.3. Встроенные функции языка SMV

Функция	Описание
abs(*t*)	абсолютная величина *t*
min(t_1, t_2)	минимум из двух величин t_1 и t_2
max(t_1, t_2)	максимум из двух величин t_1 и t_2
count(b_1,…,b_n)	число истинных логических выражений из b_1,…,b_n
sizeof(*w*)	длина слова *w* (целое число)
word1(*b*)	приведение значения истинности к типу **unsigned word**[1]
bool(*t*)	приведение 1-битного беззнакового слова или целого числа *t* к типу **boolean**
toint(*t*)	приведение *константного* слова (знакового или беззнакового)[1] или значения истинности *t* к типу **integer**
uwconst(*C*, *n*)	приведение целочисленной *константы C* к типу **unsigned word**[*n*]
swconst(*C*, *n*)	приведение целочисленной *константы C* к типу **signed word**[*n*]
unsigned(*w*)	приведение слова *w* к беззнаковому слову той же длины
signed(*w*)	приведение слова *w* к знаковому слову той же длины
extend(*w*, Δ*n*)	расширение слова *w* (Δ*n* – число дополнительных битов): – если слово беззнаковое, добавляются нули; – иначе – дублируется знаковый бит
resize(*w*, *n*)	изменение длины слова *w* (*n* – новая длина): – если *n* = *m*, где *m* = **sizeof**(*w*), возвращается слово *w*; – если *n* > *m* – **extend**(*w*, *n*-*m*); – если *n* < *m*: • если слово беззнаковое – *w*[*n*-1:0]; • иначе – **signed**(*w*[*m*-1:*m*-1]::*w*[*n*-2:0])

Пример 15.6

Ниже представлены примеры явного приведения типов.

```
word1(FALSE)          -- 0ub1_0
word1(TRUE)           -- 0ub1_1
bool(0ub1_0)          -- FALSE
bool(0ub1_1)          -- TRUE
bool(0)               -- FALSE
bool(123)             -- TRUE (как отличное от нуля целое)
toint(FALSE)          -- 0
toint(TRUE)           -- 1
toint(0uh4_F)         -- 15
toint(0sh4_F)         -- -1
uwconst(15, 4)        -- 0ub4_1111
swconst(-1, 4)        -- 0sb4_1111
unsigned(0sh4_F)      -- 0uh4_F (слово становится беззнаковым)
```

[1] Как отмечается в [Cav15], применение функции **toint** к неконстантным словам возможно, но может негативно сказаться на производительности инструмента. Вообще, использование типа **integer** не приветствуется.

```
signed(0uh4_F)        -- 0sh4_F (слово становится знаковым)
extend(0ub4_1010, 1)  -- 0ub5_0_1010 (добавляется нулевой бит)
extend(0sb4_1010, 1)  -- 0sb5_1_1010 (дублируется знаковый бит)
resize(0ub4_1010, 3)  -- 0ub3_010 (обрезается старший бит)
resize(0sb4_1010, 3)  -- 0sb3_110 (переносится знаковый бит)
```

□

15.1.3. Модули и их экземпляры

Как отмечалось в самом начале, основной структурной единицей в SMV является модуль – параметризованный фрагмент кода, допускающий тиражирование (создание экземпляров). SMV-модель представляет собой иерархию экземпляров модулей, в корне которой находится модуль `main` без параметров. Этот модуль – своего рода точка входа, с которой начинается «разворачивание» (elaboration) модели: подстановка кода вложенных модулей с заменой формальных параметров на фактические.

Определение модуля начинается с ключевого слова **MODULE**, за которым следуют имя модуля, формальные параметры и тело. В свою очередь, тело содержит следующие элементы:

- секция **VAR**:
  - переменные;
  - подмодули;
- секция **ASSIGN**:
  - начальные значения переменных;
  - описание осуществляемых вычислений;
- секция **DEFINE**:
  - макроопределения;
- секция **INIT**:
  - ограничение на начальное состояние;
- секция **INVAR**:
  - инвариант состояния;
- секция **TRANS**:
  - ограничение на переходы.

Выглядит это так:

```
MODULE M(p_1, ..., p_n)     -- имя модуля и формальные параметры
  VAR
    x_1 : T_1; x_2 : T_2;...  -- переменные
    m_1 : M_1(...); ...       -- подмодули с фактическими параметрами
  ASSIGN
    init(x_1) := s_1; ...     -- начальные значения
    next(x_1) := t_1; ...     -- следующие значения (переходы)
    x_2 := r_2; ...           -- текущие значения (комбинационная логика)
```

```
DEFINE
  y1 := f1; ...          -- макроопределения
INIT
  φ(x1, ...)             -- ограничение на начальное состояние
INVAR
  ψ(x1, ...)             -- инвариант состояния
TRANS
  δ(x1, next(x1), ...) -- ограничение на переходы
```

Отметим, что порядок секций не фиксирован, некоторые секции могут отсутствовать, а некоторые – встречаться несколько раз.

Некоторые секции мы считаем несущественными и поэтому не рассматриваем (это не относится к секциям спецификации требований, о которых мы поговорим позже).

Пример 15.7

Самой простой моделью на SMV является

```
MODULE main  -- пустая модель (ничего не делает)
```

Обратите внимание: у модуля без параметров скобки не пишутся.
□

Пример 15.8

Рассмотрим немного более сложное описание.

```
MODULE main
  VAR
    request : boolean;
    state   : {ready, busy};
  ASSIGN
    init(state) := ready;
    next(state) :=
      (state = ready & request) ? busy : ready;
```

В секции **VAR** объявляются две переменные:

- `request` типа **boolean** – признак запроса;
- `state` перечислимого типа {`ready, busy`} – состояние системы.

Далее в секции **ASSIGN** описываются начальное состояние и отношение переходов. Для этого используются следующие конструкции:

- **init**(x) – начальное значение переменной x;
- **next**(x) – значение переменной x в следующий момент времени (в терминологии цифровой аппаратуры – на следующем *такте*);
- просто x – текущее значение переменной x.

При инициализации переменная `state` получает значение `ready`. По значениям переменных `state` и `request` вычисляется значение, которое будет у переменной `state` на следующем такте:

- если в состоянии `ready` приходит запрос, система переходит в состояние `busy`;
- в противном случае следующим состоянием становится `ready`.

На значение переменной `request` не накладывается никаких ограничений: на каждом такте это может быть как `FALSE`, так и `TRUE`.

□

В SMV разрешены *ссылки вперед* – обращения к сущностям (переменным, модулям, макроопределениям), объявленным ниже по тексту, – но с некоторыми запретами:

- запрещены циклические зависимости между модулями:

```
MODULE A(...)                MODULE B(...)
  VAR                          VAR
    b : B(...);                  a : A(...); -- ошибка!
```

- запрещены *комбинационные циклы* – зависимости переменных от самих себя внутри одного такта:

```
x := y; -- ошибка!           x := y;       -- OK
y := x; -- y зависит от y    next(y) := x; -- есть задержка
```

Кроме того, в каждый момент времени допускается только одно присваивание в переменную, например:

```
init(x) := s;              init(x) := s;           next(x) := s;
next(x) := t; -- OK        x := r; -- ошибка!      x := r; -- ошибка!
```

Нужно понимать, что

- `init(x) := s` – присваивание, осуществляемое в момент времени 0;
- `next(x) := t` – в моменты времени 1, 2, ...;
- `x := r` – в моменты времени 0, 1, ...

Остановимся подробнее на многомодульных моделях. Если модуль параметризован, то при его инстанциации (создании экземпляра) должны быть указаны фактические параметры. В качестве таковых могут выступать переменные родительского модуля или выражения. Инстанциация состоит в подстановке тела модуля в родительский модуль с заменой формальных параметров на фактические (и переименованием внутренних переменных во избежание конфликта имен). В результате многократного применения этой процедуры получается «плоская» (flat) модель.

Пример 15.9

Обратимся к примеру счетчика, составленного из нескольких 1-битных ячеек [CavTu]. Код модуля `counter_cell`, описывающего одну ячейку, приведен ниже.

```
MODULE counter_cell(carry_in)
  VAR
    value : boolean;
  ASSIGN
    init(value) := FALSE;
    next(value) := value xor carry_in;
  DEFINE
    carry_out := value & carry_in;
```

У этого модуля один параметр и одна внутренняя переменная:

- `carry_in` – входной бит переноса;
- `value` – значение соответствующего разряда счетчика.

В начальный момент времени переменная `value` имеет значение `FALSE`; очередное значение получается путем сложения по модулю 2 текущего значения с входным битом переноса:

```
next(value) := value xor carry_in;
```

В секции **`DEFINE`** определяется выходной бит переноса:

```
carry_out := value & carry_in;
```

Это не переменная, а *макроопределение* – обозначение для выражения справа.

В модуле верхнего уровня `main` в секции **`VAR`** создаются три ячейки (экземпляры модуля `counter_cell`): `bit0`, `bit1` и `bit2`.

```
MODULE main
  VAR
    bit0 : counter_cell(TRUE);
    bit1 : counter_cell(bit0.carry_out);
    bit2 : counter_cell(bit1.carry_out);
```

На вход ячейки `bit2` подается выходной бит переноса ячейки `bit1`, на вход `bit1` – выходной бит переноса `bit0`, на вход `bit0` – `TRUE` (постоянный инкремент).

Обратите внимание: доступ к элементам подмодулей осуществляется с помощью *составных идентификаторов* – последовательностей имен, разделенных точками.

□

Пример 15.10

Функциональность модулей можно описывать неявно, в форме ограничений. Так, модуль из примера 15.8 допускает следующее альтернативное определение:

```
INIT
  state = ready
TRANS
  (state = ready & request) <-> next(state) = busy
```

Обратите внимание: в конце секций **INIT** и **TRANS** точка с запятой не ставится.

□

Вообще, запись

```
ASSIGN
  init(x) := s;
  y := r;
  next(z) := t;
```

эквивалентна

```
INIT
  x in s -- в общем случае s, r и t являются множествами
INVAR
  y in r
TRANS
  next(z) in t
```

15.1.4. Конструкция выбора

Для записи сложных условных выражений используется *конструкция выбора* **case**:

```
case
  cond_1: expr_1;
  ...
  cond_n: expr_n;
esac
```

Данная запись интерпретируется следующим образом:

- последовательно проверяются условия $cond_i$, $i \in \{1, \ldots, n\}$;
  - если условие $cond_i$ истинно, результатом является $expr_i$;
- ситуация, когда все условия ложны, считается ошибкой.

Тип **case**-выражения – минимальный тип, к которому неявно приводимы все выражения $expr_i$. Если такого типа нет, выдается ошибка.

Пример 15.11

Ниже отношение переходов модуля из примера 15.8 задано с помощью **case**:

```
next(state) :=
  case
    (state = ready & request = TRUE) : busy;
    TRUE                             : ready;
  esac;
```

□

15.1.5. Средства спецификации требований

Инструмент NuSMV позволяет специфицировать требования в виде LTL- и CTL- формул, а также на языке PSL (Property Specification Language) [PSL10].

Остановимся подробнее на LTL-спецификациях. Они предваряются ключевым словом **LTLSPEC**, после которого идет формула следующего вида:

- выражение типа **boolean** (элементарное высказывание);
- формула, использующая классические логические связки (см. табл. 15.2);
- φ **U** ψ – «до тех пор, пока не» (*Until*);
- **X**φ – «в следующий момент времени» (*neXt time*);
- **F**φ – «когда-нибудь в будущем» (*in the Future*);
- **G**φ – «всегда» (*Globally*);
- φ **V** ψ – «освобождение» (*Release*).

Отметим, что в NuSMV есть темпоральные операторы прошедшего времени (нами не рассматриваемые), а также варианты **F** и **G** с ограничениями по времени:

- $\pi \models \mathbf{F}[l, u]\varphi$ тогда и только тогда, когда $\exists t \in [l, u]$: $\pi^{(t)} \models \varphi$;
- $\pi \models \mathbf{G}[l, u]\varphi$ тогда и только тогда, когда $\forall t \in [l, u]$: $\pi^{(t)} \models \varphi$.

Свойство *справедливости* (см. лекцию 9) выражается с помощью клауз **JUSTICE** (в ранних версиях NuSMV использовалось ключевое слово **FAIRNESS**):

```
JUSTICE prop1 -- выражение типа boolean
...
JUSTICE propn
```

При проверке свойств учитываются только те вычисления модели, на которых каждая из **JUSTICE**-формул истинна бесконечное число раз.

15.2. Примеры SMV-моделей

Рассмотрим более сложные примеры использования SMV. Сравните их с Promela-моделями, разработанными для аналогичных задач (см. лекцию 10 и упражнения к ней).

15.2.1. Алгоритм Петерсона

Начнем с алгоритма Петерсона взаимного исключения двух процессов (см. лекцию 9). Еще раз приведем описание алгоритма на языке while:

```
while true do
  NCS_i: // некритическая секция
  SET_i: flag_i := 1; turn := i; // запрос
  TST_i: while (flag_1-i = 1) ∧ (turn = i) do skip end; // ожидание
  CRS_i: // критическая секция
  RST_i: flag_i := 0 // выход
end
```

В начальный момент времени все переменные (`turn`, $flag_0$ и $flag_1$) имеют значение 0.

Прежде всего отметим: в основе SMV лежит *синхронный* параллелизм, нам же требуется смоделировать *асинхронное* исполнение процессов. Для этого мы добавим в каждый процесс параметр `exec`:

- `exec = TRUE` – процесс исполняется;
- `exec = FALSE` – процесс приостановлен.

Главный модуль (играющий, помимо прочего, роль планировщика) недетерминированным образом выбирает один из процессов и устанавливает соответствующий флаг *exec* (сбрасывая при этом флаг, установленный ранее).

Рассмотрим модуль *p*, описывающий отдельный процесс.

```
MODULE p(i, flag0, flag1, turn, exec)
  VAR
    state : {NCS, SET, SET1, TST, CRS, RST, RST1};

  ASSIGN
    init(state) := NCS;

    next(state) :=
      case
        !exec   /-- процесс не активен --/  : state;
        (state = NCS)                       : SET;
        (state = SET)                       : SET1;
        (state = SET1)                      : TST;
        (state = TST) &  (flag1 & turn = i) : TST; -- блокировка
        (state = TST) & !(flag1 & turn = i) : CRS;
        (state = CRS)                       : RST;
        (state = RST)                       : RST1;
        (state = RST1)                      : NCS;
      esac;

    init(flag0) := FALSE;

    next(flag0) :=
      case
        !exec         : flag0;
        (state = SET) : TRUE;  -- запрос на вход
        (state = RST) : FALSE; -- выход
      esac;
```

Модуль имеет пять параметров:

- `i` – номер процесса;
- `flag0` – «свой» флаг ($flag_i$);
- `flag1` – «чужой» флаг ($flag_{1-i}$);
- `turn` – общая для обоих процессов переменная;
- `exec` – признак исполнения процесса.

Внутренняя переменная одна: `state` – метка исполняемого оператора. Значения `SET1` и `RST1` обозначают точки после исполнения операторов $flag_i$ = 1 и $flag_i$ = 0 соответственно. Начальное значение переменной `state` – `NCS`.

Описание на языке SMV как бы вывернуто наизнанку: вместо потока управления во главу угла ставятся потоки данных – для переменных задаются способы вычисления следующих значений в зависимости от текущего состояния. Например, запись

```
next(state) :=
  case
    !exec : state;
    ...
    (state = TST) & (flag1 & turn = i) : TST;
    ...
  esac;
```

означает, что если процесс не исполняется (`!exec`) или ожидает разрешения на вход в критическую секцию (`state = TST`), но разрешение не получено (`flag1 & turn = i`), то состояние не меняется.

В модуле `main` объявляются и инициализируются переменные `flag` и `turn`, используемые в процессах, и переменная `exec`, хранящая номер процесса, выбранного для исполнения. Также в модуле `main` создаются два экземпляра модуля `p`:

- процесс `p0` – исполняется, когда `exec = 0b_0`;
- процесс `p1` – исполняется, когда `exec = 0b_1`.

```
MODULE main
  VAR
    flag : array 0..1 of boolean;
    turn : word[1];
    exec : word[1];

    p0 : p(0b_0, flag[0], flag[1], turn, exec0);
    p1 : p(0b_1, flag[1], flag[0], turn, exec1);

  DEFINE
    exec0 := (exec = 0b_0); -- исполняется процесс p0
    exec1 := (exec = 0b_1); -- исполняется процесс p1
```

```
ASSIGN
  init(turn) := 0b_0;

  next(turn) :=
    case
      exec0 & (p0.state = SET1) : 0b_0;
      exec1 & (p1.state = SET1) : 0b_1;
      TRUE                      : turn;
    esac;
```

Начальное значение переменной `turn`, как и логика изменения значения этой переменной, задается в процессе `main`, а не `p0` и `p1`.

Напомним, какие свойства определяют корректность алгоритма Петерсона:

- $\mathbf{G}\{\neg(@CRS_0 \wedge @CRS_1)\}$ – два процесса не могут одновременно пребывать в критической секции;
- $\mathbf{G}(@SET_i \rightarrow \mathbf{F}@CRS_i)$, где $i \in \{0, 1\}$, – если процесс запросил вход в критическую секцию, то рано или поздно этот запрос будет удовлетворен.

В нотации инструмента эти свойства записываются следующим образом:

```
LTLSPEC G !(p0.state = CRS & p1.state = CRS);
LTLSPEC G  (p0.state = SET -> F (p0.state = CRS));
LTLSPEC G  (p1.state = SET -> F (p1.state = CRS));
```

Свойства живости выполняются только в предположении справедливости планировщика, которое в данном случае записывается так:

```
JUSTICE exec0 -- процесс p0 время от времени исполняется
JUSTICE exec1 -- процесс p1 время от времени исполняется
```

15.2.2. Головоломка «Волк–коза–капуста»

Применим инструмент NuSMV для решения головоломки «Волк, коза и капуста». Напомним постановку задачи. Крестьянину нужно перевезти через реку волка, козу и капусту. Лодка небольшая: в ней может поместиться крестьянин, а с ним либо волк, либо коза, либо капуста. Если оставить волка с козой на берегу, волк съест козу; если оставить козу с капустой, коза съест капусту. Как крестьянину перевезти свой груз?

Для обозначения положения лодки (с крестьянином), волка, козы и капусты введем булевы переменные: `boat`, `wolf`, `goat` и `cabb` соответственно. В начальный момент времени все переменные имеют значение `FALSE`. Цель – перевести «систему» в состояние, в котором все эти переменные имеют значение `TRUE`, не нарушив при этом «условия безопасности».

```
MODULE main
  VAR
    boat : boolean;
    wolf : boolean;
    goat : boolean;
    cabb : boolean;

  ASSIGN                -- вначале все находятся на берегу FALSE:
    init(boat) := FALSE; -- крестьянин с лодкой
    init(wolf) := FALSE; -- волк
    init(goat) := FALSE; -- коза
    init(cabb) := FALSE; -- капуста
```

Как меняется состояние системы? Крестьянин плавает через реку туда и обратно:

```
    next(boat) := !boat;
```

Он может перевести с собой любой «объект», находящийся с ним на одном берегу:

```
    next(wolf) := case
                    wolf = boat : {wolf, !wolf};
                    TRUE        :  wolf;
                  esac;

    next(goat) := case
                    goat = boat : {goat, !goat};
                    TRUE        :  goat;
                  esac;

    next(cabb) := case
                    cabb = boat : {cabb, !cabb};
                    TRUE        :  cabb;
                  esac;
```

Следует учесть, что лодка небольшая и вмещает (помимо крестьянина) один «объект»:

```
  TRANS
    count(next(wolf) != wolf,
          next(goat) != goat,
          next(cabb) != cabb) <= 1
```

Введем флаг `safe`, сигнализирующий о безопасности состояния (состояние считается опасным, если волк с козой или коза с капустой остались на одном берегу без присмотра):

```
  DEFINE
    safe := !((wolf = goat & boat != wolf)
             |(goat = cabb & boat != goat));
```

Модель готова. Чтобы заставить инструмент найти решение головоломки, напишем следующую «провокационную» спецификацию:

```
LTLSPEC !(safe U (boat & wolf & goat & cabb))
```

Спецификация утверждает, что не существует вычисления, переводящего «систему» в целевое состояние без нарушения «свойств безопасности». На самом деле такое вычисление есть; инструмент его находит и предъявляет в качестве контрпримера.

15.3. Введение в инструмент NuSMV

NuSMV [NuSMV] – это сравнительно новая реализация SMV (версия 2.6.0 была выпущена 14 октября 2015 года), созданная в Центре Бруно Кесслера (FBK, Fondazione Bruno Kessler). К разработке также причастны Университет Карнеги–Меллона (CMU, Carnegie Mellon University), Университет Генуи (Università degli Studi di Genova) и Трентский университет (Università degli Studi di Trento). Инструмент распространяется с открытым исходным кодом по лицензии LGPL v2.1. В качестве надстройки над NuSMV, позволяющей верифицировать системы с бесконечным числом состояний, в FBK разработан инструмент nuXmv [Cav14] (для этого применяется SMT-решатель MathSAT5 [MatS]).

Для построения и манипулирования с двоичными решающими диаграммами NuSMV использует библиотеку CUDD [Cudd]. В качестве SAT-решателя NuSMV использует MiniSat [MinS] (вариант по умолчанию) или zChaff [Chaff].

15.3.1. Функции инструмента

Инструмент NuSMV может работать в трех режимах:

- режим неограниченной проверки моделей с использованием двоичных решающих диаграмм (BDD-based model checking) – режим по умолчанию;
- режим ограниченной проверки модели с использованием SAT-решателей (SAT-based model checking) – включается опцией `-bmc`;
- интерактивный режим, позволяющий загружать модель из файла, проводить пошаговую симуляцию и формальную проверку – включается опцией `-int`.

15.3.2. Основные опции

Рассмотрим некоторые полезные опции инструмента NuSMV. Более подробная информация доступна в руководстве [Cav15].

Неограниченная проверка модели на основе BDD:

```
NuSMV [-v уровень-вывода] имя-файла
```

С помощью опции `-v` задается детальность отладочного вывода. Значениями опции являются натуральные числа: 1, 2, ... (чем больше число, тем больше информации выводится).

Ограниченная проверка модели на основе SAT:

```
NuSMV [-v уровень-вывода] -bmc [-bmc_length глубина] имя-файла
```

Опция `-bmc` говорит о применении *ограниченной проверки моделей* (*bounded model checking*). Следующая опция `-bmc_length` задает ограничение на глубину проверки – целое неотрицательное число: 0, 1, ... (значение по умолчанию – 10).

Интерактивный режим работы:

```
NuSMV -int
```

Опция `-int` включает интерактивный режим, в котором доступно большое число команд, позволяющих управлять разными аспектами построения моделей и их анализа. Рассмотрим некоторые типовые сценарии.

Неограниченная проверка модели:

```
> read_model -i имя-файла                  чтение модели из файла
> flatten_hierarchy                        «разворачивание» модели
> encode_variables                         двоичная кодировка переменных
> build_model                              построение BDD-представления
> check_ltlspec [-P имя-свойства]          проверка LTL-свойства
> quit                                     выход
```

Ограниченная проверка модели:

```
> read_model -i имя-файла
> flatten_hierarchy
> encode_variables
> build_boolean_model                      построение булевых уравнений
> bmc_setup                                инициализация режима BMC
> check_ltlspec_bmc -k глубина [-P имя-свойства]
> quit
```

Симуляция модели:

```
> read_model -i имя-файла
> flatten_hierarchy
> encode_variables
> build_model
> pick_state [-i|-r]                       выбор начального состояния
> simulate -k число-шагов [-v]             симуляция модели
> quit
```

Команда `pick_state` отвечает за выбор начального состояния:

- `-i` – интерактивный выбор:
  - если состояний много, инструмент предложит задать ограничение;
  - в противном случае выведет список всех доступных состояний;
- `-r` – случайный выбор (вариант по умолчанию).

После симуляции можно посмотреть трассу, запустив команду `show_traces`. Выведется начальное состояние и список идентификаторов состояний. Посмотреть состояние можно с помощью команды

```
> goto_state идентификатор
```

Распечатать идентификатор текущего состояния можно с помощью команды

```
> print_current_state
```

Сброс сессии верификации осуществляется командой

```
> reset
```

После этого можно загружать другую модель и работать с ней.
По соглашению, файлы с SMV-моделями имеют расширение `smv`.

15.4. Вопросы и упражнения

1. Опишите схему моделирования асинхронных параллельных систем на языке SMV.
2. Реализуйте на языке SMV синхронный канал передачи сообщений (слов длины $k = 4$) от одного процесса к другому (см. лекцию 9). Разработайте тестовое окружение – пару *асинхронных* процессов, взаимодействующих через этот канал. Исследуйте работу системы с помощью симулятора NuSMV (или nuXmv).
3. Реализуйте на языке SMV алгоритм Евклида нахождения НОД двух натуральных чисел (беззнаковых слов длины 32). Используя симулятор NuSMV, протестируйте разработанную модель на некоторых парах чисел.
4. Реализуйте на языке SMV *каскадный сумматор* (*ripple-carry adder*) k-битных слов (где $k = 4$): a и b – складываемые слова; c – результат сложения; *overflow* – бит переноса. Для реализации каскадного сумматора используйте *полные 1-битные сумматоры* (*full adders*):

   $z = x \oplus y \oplus c_{in}$,

   $c_{out} = maj(x, y, c_{in}) = (x \wedge y) \vee ((x \oplus y) \wedge c_{in})$.

   Здесь x и y – складываемые биты; c_{in} – входной бит переноса; z – бит результата; c_{out} – выходной бит переноса.

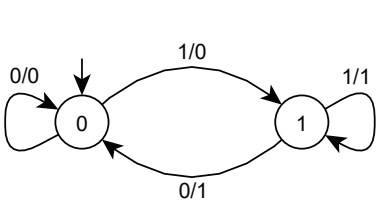
5. Реализуйте на языке SMV очередь (FIFO) заданного размера ($N = 8$), элементами которой являются k-битные слова ($k = 4$). Интерфейс следующий:
   - вход `enq_val` (1 бит) – запуск операции добавления элемента в очередь;
   - вход `enq_dat` (k бит) – добавляемый элемент;
   - вход `deq_val` (1 бит) – запуск операции извлечения элемента из очереди;
   - выход `deq_dat` (k бит) – извлекаемый элемент;
   - выход `enq_rdy` (1 бит) – возможность добавления (в очереди есть место);
   - выход `deq_rdy` (1 бит) – возможность извлечения (очередь не пуста);

   Обратите внимание: операции добавления и извлечения элементов можно запускать параллельно.
6. Используя очередь из предыдущего упражнения, смоделируйте на языке SMV систему «Производитель–потребитель»: производитель время от времени генерирует данные и записывает их в очередь (если очередь не полна); потребитель время от времени считывает данные из очереди (если очередь не пуста).
7. Реализуйте на языке SMV *арбитр со статической системой приоритетов*. На каждом такте арбитр принимает запросы со стороны N клиентов ($N = 4$) – один клиент может выставить только один запрос. Если есть запрос от клиента с номером 0, доступ дается ему; иначе если есть запрос от клиента с номером 1, доступ дается ему и т. д. Интерфейс следующий:
   - вход $\texttt{req_val}_i$ (1 бит) – признак запроса со стороны i-го клиента;
   - выход $\texttt{grant}_i$ (1 бит) – признак предоставления доступа i-му клиенту.

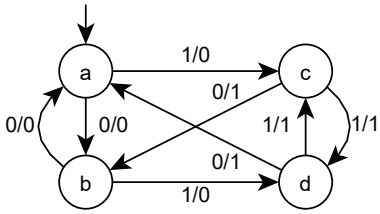
   Формализуйте следующие свойства (используйте конструкцию **LTLSPEC**):
   - *корректность* – доступ может быть предоставлен только тем клиентам, кто выставил запросы;
   - *взаимное исключение* – в каждый момент времени доступ получает не более чем один клиент;
   - *справедливость* – если клиент выставляет запрос, то рано или поздно он получит доступ (предполагается, что клиент не снимает запрос).

   Проверьте с помощью инструмента NuSMV, какие из этих свойств выполняются.
8. Реализуйте на языке SMV арбитр, удовлетворяющий свойствам корректности, взаимного исключения и справедливости (см. предыдущую задачу).
9. Разработайте на языке SMV модель системы управления стиральной машиной (см. упражнение к лекции 1). Специфицируйте требования в виде формул LTL (используйте конструкцию **LTLSPEC**). Проверьте модель с помощью инструмента NuSMV.

10. Представьте на языке SMV конечные автоматы, показанные ниже. С помощью инструмента NuSMV докажите их эквивалентность: $\mathbf{G}\{\lambda_A(s_A, x) = \lambda_B(s_B, x)\}$.



Автомат A



Автомат B

11. Реализуйте на языке SMV алгоритм целочисленного деления двух натуральных чисел (беззнаковых слов длины 4). Докажите корректность реализации с помощью инструмента NuSMV.
12. Решите с помощью инструмента NuSMV головоломку «Ханойская башня». Есть три стержня, на один из которых нанизано $N = 8$ дисков. Диски различаются диаметром и лежат меньший на большем. Задача состоит в том, чтобы перенести пирамиду дисков на другой стержень (за наименьшее число шагов). За один шаг разрешается переносить только один диск, причем нельзя класть больший на меньший.
13. Смоделируйте на языке SMV протокол MSI обеспечения *когерентности кеш-памяти* в многоядерном процессоре. У каждого ядра есть локальная кеш-память, содержащая копии некоторых блоков основной памяти. С каждым блоком связано состояние:
    - *M* = *Modified* – данные были модифицированы («грязная» копия);
    - *S* = *Shared* – данные согласованы с данными в памяти («чистая» копия);
    - *I* = *Invalid* – данных нет.

    Будем считать, что все ядра подключены к общей шине и видят, что делают другие ядра.

    Состояние меняется по следующему принципу (см. рис. ниже[1]):
    - чтение данных ядром (*PrRd*):
      - если состояние *I* – данные загружаются, состояние становится *S*;
        - на шину выставляется запрос *BusRd* (загрузка данных в кеш-память ядра);
      - *S* или *M* – состояние не меняется;
    - запись данных ядром (*PrWr*):
      - если состояние *I* – данные загружаются, состояние становится *M*;
        - на шину выставляется запрос *BusRdX* (эксклюзивная загрузка данных в кеш-память ядра);

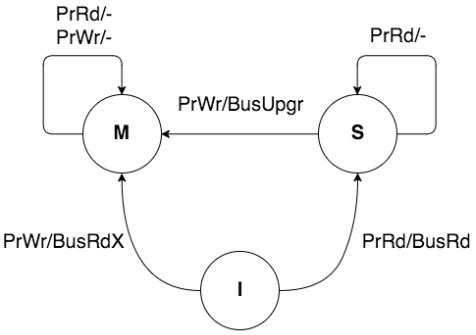
[1] https://en.wikipedia.org/wiki/MSI_protocol.

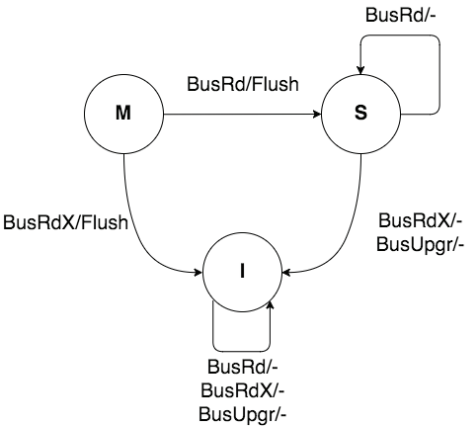
- *S* – состояние меняется на *M*;
  - на шину выставляется запрос *BusUpgr* (сброс данных в кеш-памяти других ядер);
- *M* – состояние не меняется;

❍ появление на шине запроса *BusRd* (от другого ядра):
- если состояние *I* или *S* – состояние не меняется:
  - в случае *S* данные посылаются ядру-запросчику (передача данных не моделируется).
- *M* – состояние меняется на *S*:
  - данные посылаются ядру-запросчику, измененные данные записываются в память (*Flush*);

❍ появление на шине запроса *BusRdX*:
- состояние сбрасывается в *I*;
  - если состояние было *M* – измененные данные записываются в память (*Flush*);

❍ появление на шине запроса *BusUpgr*:
- состояние сбрасывается в *I*.





Специфицируйте и проверьте свойство когерентности кеш-памяти: «грязная» копия блока данных (копия с состоянием *M*) может присутствовать не более чем в одном ядре.

14. Формализуйте операционную семантику языка SMV.

Лекция 16

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 (model-based testing). 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, символьное исполнение и конколическое (конкретно-символьное) тестирование, генерация тестов с помощью обхода графа состояний.

16.1. 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 (точнее, ее модели) 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 – так называемого *тестового набора* (*test suite*). 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 табл. 16.1. Более детальное сравнение методов можно посмотреть в обзоре [Кул08].

Таблица 16.1. Сравнение методов тестирования и формальной верификации

Тестирование	**Формальная верификация**
+ Проверяется реальная система + Проверка осуществляется в реальных условиях	+ Проверяются все возможные сценарии + Проверяется класс возможных реализаций
– Выполняется на поздних этапах разработки – Проверяется ограниченный набор сценариев – Высокая трудоемкость разработки тестов – Необходимость большого штата тестировщиков	– Проверяется модель, а не реальная система – Проверяется ограниченный набор свойств – Высокая вычислительная сложность – Высокие требования к квалификации верификаторов

Методы тестирования и формальной верификации не являются взаимоисключающими; напротив, они дополняют друг друга и могут использоваться совместно. Возможна, например, следующая организация процесса создания ПО, в целом соответствующая концепции *разработки на основе моделей* (*model-driven development / engineering*) [Rai12]:

- создается модель системы;
- проводится формальная верификация модели;
- на основе модели синтезируется или реализуется система;
- осуществляется тестирование системы.

Более того, формальные методы могут применяться для «усиления» тестирования, а именно для решения следующих задач:

- генерация тестовых воздействий (последовательностей стимулов);
- проверка корректности поведения (реакций, выдаваемых в ответ на стимулы);
- оценка полноты тестирования (репрезентативности тестового набора).

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 практики этого типа, сделаем небольшой экскурс в историю.

Тестирование на основе моделей в том или ином виде существует с 1970-х годов, однако первоначально применялось лишь в таких областях, как проектирование аппаратуры и телекоммуникационных протоколов [Pet15]. Постепенно в индустрию ПО пришло осознание, что модели могут использоваться для тестирования более широкого класса систем. В конце 1980-х годов стали появляться работы (например, [Ost88]), предлагающие методы построения тестов по формальным спецификациям; это были полуавтоматические подходы, позволяющие генерировать «шаблоны» для дальнейшей ручной разработки тестов по ним. Полноценные инструменты стали создаваться в середине 1990-х годов. Первые промышленно применимые MBT-технологии были разработаны в ИСП РАН – это KVEST [Bur99] и UniTESK [Бур03b]. Оба подхода базировались на программных контрактах (пред- и постусловиях) и техниках обхода графов состояний. Чуть позже появился близкий по возможностям инструмент SpecExplorer [Bla05] от Microsoft Research, основанный на машинах абстрактных состояний (ASM, Abstract State Machines). Современные методы тестирования немыслимы без применения формальных методов.

16.2. Тестирование программ без состояния

Начнем с *программ преобразования данных*, поведение которых зависит только от значений входных переменных (см. лекцию 2). Такие программы называют *программами без состояния*, поскольку их поведение не зависит от предыстории запусков. Техники, описанные ниже, могут использоваться для автоматизации тестирования отдельных функций (методов, процедур).

16.2.1. 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. Самый простой подход этого типа – *вероятностное* (*стохастическое*) *тестирование*:

- генерируется случайное воздействие (значения входных переменных):
  - если предусловие нарушается, воздействие отбрасывается;
  - в противном случае воздействие подается на программу;
- по завершении программы (если 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(x);                 // 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[1]. Недостатки очевидны:

- многие тестовые воздействия приводят к одинаковому поведению тестируемой программы, т. е. к исполнению одних и тех же путей в графе потока управления;
- вероятность реализации некоторых ситуаций, так называемых *граничных ситуаций* (*corner cases*), может быть крайне низкой (к таким ситуациям относятся проходы по путям, описываемым формулами с большим числом условий: $\varphi_1 \wedge \ldots \wedge \varphi_k$).

[1] Некоторые трудности может привнести генерация сложно структурированных данных, но этот вопрос мы здесь опускаем.

16.2.2. Тестирование черного ящика

Ранее, в лекции 3, мы говорили о таких характеристиках методов верификации, как *значимость* (*soundness*) и *полнота* (*completeness*):

- метод называется *значимым*, если из того, что вынесен отрицательный вердикт, следует, что программа ошибочна;
- метод называемся *полным*, если из того, что вынесен положительный вердикт, следует, что программа корректна.

В этих терминах знаменитое высказывание Эдсгера Дейкстры о том, что тестирование может доказать наличие 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* (*black-box testing*). Так называется класс методов построения тестов, не использующих информацию о структуре программы (исходный или бинарный код) [Bei95]. Ниже описывается еще один подход этого типа, основанный на использовании *функциональных моделей тестового покрытия*.

Пусть заданы программа (реализация) и требования к ней (спецификация). *Тестовым покрытием*, или просто *покрытием*, называется конечное множество *тестовых ситуаций* – классов эквивалентности возможных вычислений программы. *Моделью покрытия* называется способ описания покрытия по реализации и/или спецификации. Модель покрытия называется *функциональной*, если она базируется только на спецификации (не использует реализацию).

На основе программных контрактов можно определить следующие функциональные модели покрытия:

- *покрытие ветвей функциональности*;
- *покрытие значений истинности условий*.

Ветви функциональности – это тестовые ситуации, соответствующие принципиально разным вариантам поведения программы. В языке ACSL (см. лекцию 4) ветви функциональности задаются с помощью конструкции

behavior *b*: **assumes** φ ...,

где *b* – имя ветви, а φ – ее условие. В классическом случае ветви функциональности *разделимы* (*disjoint*), т. е. имеют попарно несовместные условия, а их множество *полно* (*complete*): из истинности предусловия следует истинность условия хотя бы одной ветви. В языке ACSL это требование может быть нарушено: в этом случае целесообразно использовать *кросс-покрытие*, где

тестовая ситуация идентифицируется не отдельной ветвью, а их комбинацией (максимальным непротиворечивым множеством ветвей).

Пример 16.1

В представленном ниже фрагменте спецификации функции дихотомического поиска (см. лекцию 4) выделены две ветви: первая (*succes*) имеет тривиальное условие (*true*) – постусловие ветви проверяется всегда; вторая (*failure*) требует упорядоченности массива – если это не так, постусловие не проверяется.

```
behavior success:
  ...
behavior failure:
  assumes \forall integer i, integer j;
    0 <= i < j <= n-1 ==> x[i] <= x[j];
  ...
```

Соответственно, здесь можно выделить две тестовые ситуации:

- {*succes*};
- {*succes*, *failure*}.

□

Покрытие значений истинности условий – это семейство моделей покрытия, определяемых в терминах комбинаций значений истинности элементарных формул (атомов). Самыми известными моделями этого типа являются:

- *покрытие всех возможных комбинаций* (*product coverage*);
- *покрытие попарных комбинаций* (*pairwise coverage*);
- *покрытие условий/решений* (*C/DC*, *Condition/Decision Coverage*) – покрытие значений истинности каждого условия независимо от других условий (*condition coverage*) плюс покрытие значений истинности формулы в целом (*decision coverage*);
- *модифицированное покрытие условий/решений* (*MC/DC*, *Modified Condition/Decision Coverage*) – демонстрация влияния каждого условия на значение истинности формулы (для каждого условия должна присутствовать пара комбинаций, отличающихся значением только этого условия и соответствующих разным значениям формулы)[1] [Chi94].

Пример 16.2

Пусть предусловие имеет вид $((y = 0) \rightarrow (x = 0)) \wedge ((x > 3) \vee (y < x))$. Формула содержит четыре атома:

$(y = 0)$, $(x = 0)$, $(x > 3)$ и $(y < x)$.

Всего возможны $2^4 = 16$ комбинаций значений истинности, однако нас устраивают только те из них, для которых предусловие истинно. Получаем

[1] Критерий тестового покрытия MC/DC используется, например, для тестирования авионики (стандарт DO-178B [DO178B]).

9 вариантов[1], более половины 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*) – пример: $x = 0$, $y = -1$;
7) (*false*, *false*, *true*, *true*) – пример: $x = 4$, $y = 3$;
8) (*false*, *false*, *true*, *false*) – пример: $x = 4$, $y = 4$;
9) (*false*, *false*, *false*, *true*) – пример: $x = 3$, $y = 2$.

□

Покрытие не только задает критерий полноты тестирования, но и может использоваться для *направленной генерации тестовых воздействий* (*coverage-driven verification*). Для этого используются SMT-решатели (см. лекцию 8).

Например, для ситуации (*false*, *false*, *true*, *false*), соответствующей формуле

$$(y \neq 0) \wedge (x \neq 0) \wedge (x > 3) \wedge (y \geq x),$$

с использованием решателя может быть построено воздействие $\{x \mapsto 4, y \mapsto 4\}$.

В реальной жизни ограничения могут быть гораздо сложнее и вовлекать указатели, результаты работы с файлами, базами данных и т. п. (исследование этих вопросов выходит за рамки книги).

Пример 16.3

Приведем простой пример на критерий MC/DC. Рассмотрим формулу $D \equiv C_1 \vee C_2 \vee C_3$, где C_1, C_2 и C_3 – некоторые элементарные условия. Согласно критерию, тесты должны покрывать ситуации, демонстрирующие влияние отдельных условий (C_i) на значение истинности формулы в целом (D).

Например, набор {(*false*, *false*, *false*), (*true*, *true*, *true*)} этому требованию не соответствует, хотя удовлетворяет критерию C/DC. Минимальным MC/DC-набором для рассматриваемого примера является следующий:

1) (*false*, *false*, *false*) – формула D ложна;
2) (*true*, *false*, *false*) – в паре с 1 показывает влияние C_1: D истинна;
3) (*false*, *true*, *false*) – влияние C_2;
4) (*false*, *false*, *true*) – влияние C_3.

□

Более подробную информацию о критерии MC/DC можно найти в работе [Hay01].

---

[1] Ситуации соответствуют конъюнкциям совершенной ДНФ, построенной по формуле предусловия.

16.2.3. Тестирование белого ящика

Тестирование белого ящика (*white-box testing*) 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 составляется формула, истинность которой в начальном состоянии гарантирует исполнение программы по этому пути. Для этого используется *метод обратных подстановок* (см. лекции 3 и 5) или (что более распространено) SSA-представление (см. лекцию 11).

Пример 16.3

Для программы целочисленного деления ограничим число итераций в цикле числом 2 и построим множество возможных вычислений (путей) – см. рис. 16.1.

Рассмотрим путь, соответствующий исполнению ровно двух итераций цикла (на рисунке он выделен серым цветом). Этот путь можно записать в форме линейной аннотированной программы (см. лекцию 3):

`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;`	$((r - b) \geq b) \wedge ((r - b) < 2b) \equiv (r \geq 2b) \wedge (r < 3b)$
`{r ≥ b}`	$(r \geq b) \wedge (r < 2b)$
`q := q + 1;`	
`r := r - b`	$((r - b) < b) \equiv (r < 2b)$
`{r < b}`	$(r < b)$

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

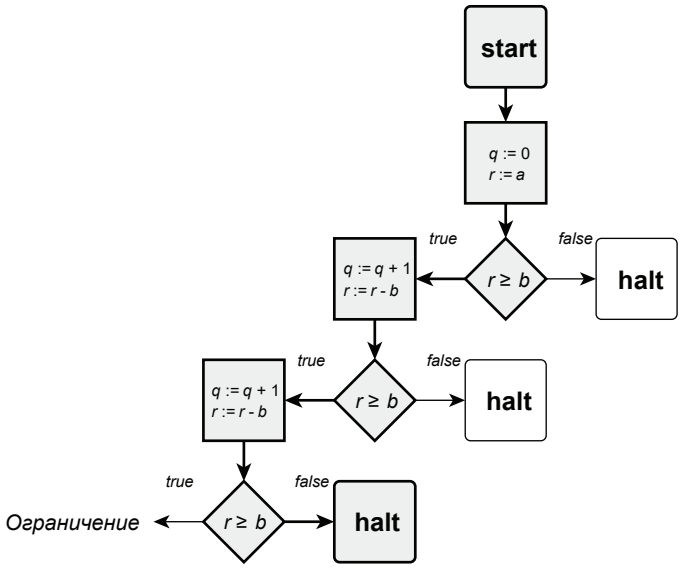


Рис. 16.1 ❖ Ограниченный поиск в глубину в программе целочисленного деления

Применяя метод обратных подстановок, получим формулу

$(a \geq b) \wedge (a \geq 2b) \wedge (a < 3b)$.

Совмещая ее 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 символьном исполнении программ могут возникать сложные ограничения, которые не по силам SMT-решателям. В последнее время получил распространение подход, называемый *конколическим тестированием* (*concolic testing*), совмещающий конкретное и символьное исполнения программы (*concolic* = *conc*rete + symb*olic*). Концепция была предложена в 2005 году Годефруа[1], Кларлундом[2] и Сеном[3] [God05]. Термин «конколическое тестирование» впервые появился в работе [Sen05].

---

[1] Патрис Годефруа (Patrice Godefroid) – бельгийский ученый-информатик; автор метода редукции частичных порядков в проверке моделей параллельных систем; известен работами в области фаззинга (fuzzing) ПО.

[2] Нильс Кларлунд (Nils Klarlund) – датский и американский исследователь-разработчик; специалист по символьному анализу и верификации программ, а также пользовательским интерфейсам.

[3] Каушик Сен (Koushik Sen) – американский ученый-информатик индийского происхождения; известен работами по тестированию, мониторингу и верификации программ.

Для реализации конколического тестирования программа инструментируется так, чтобы при ее исполнении строилась формула пути. В несколько упрощенной форме подход выглядит следующим образом.

1. Генерируется случайное тестовое воздействие (входные данные программы).
2. Тестовое воздействие подается на программу:
   a) при исполнении программы строится формула пути;
   b) если путь новый, он запоминается.
3. Выбирается один из пройденных путей:
   a) одно из условий формулы пути заменяется на его отрицание;
      i) условия, следующие за измененным (описывающие исполнение программы после соответствующего ветвления), удаляются;
   b) если формула уже встречалась:
      i) выбирается другое условие или другой путь;
   c) если не удается составить новую формулу, тестирование завершается.
4. Полученная формула передается решателю:
   a) если решатель не справился:
      i) генерируется случайное тестовое воздействие;
   b) если формула невыполнима:
      i) выбирается другое условие или путь;
   c) если формула выполнима:
      i) построенная модель используется как тестовое воздействие.
5. Переход к шагу 2.

Формулы, описывающие вычисления программы, не меняются – они по-прежнему могут быть сложными, – однако рандомизация данных компенсирует неидеальность решателей и позволяет продвинуться дальше в покрытии путей. Также в конколическом тестировании применяется абстрактная интерпретация (см. лекцию 7) – вместо сложных формул строятся сравнительно простые аппроксимирующие ограничения (их решения с большой вероятностью являются решениями исходных формул). При наличии циклов длина путей ограничивается искусственно. Если известно предусловие программы, его, конечно, следует учитывать.

Пример 16.4

Обратимся к программе целочисленного деления из примера выше. Первым делом в процессе конколического тестирования генерируется случайное воздействие. Предположим, что 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*

При исполнении программы (с учетом предусловия) строится формула:

$(b > 0) \wedge (a \geq b) \wedge (a < 2b)$.

В этой формуле берется последнее условие и заменяется на противоположное:

$(b > 0) \wedge (a \geq b) \wedge (a \geq 2b) \equiv (b > 0) \wedge (a \geq 2b)$.

Полученное ограничение решается: в результате получается новое воздействие, например $\{a \mapsto 123, b \mapsto 45\}$.

Процесс продолжается...

□

16.3. Тестирование программ с состоянием

Перейдем 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 большое число программ и библиотек: реализации абстрактных типов данных (таких как список, стек, отображение и т. п.), реагирующие системы (драйверы, контроллеры, коммутаторы и т. п.), крупные корпоративные системы и др. Техники, описанные ниже, могут использоваться для тестирования любых систем функций над общими переменными (модулей и классов).

16.3.1. Обход графа состояний программы

Естественным способом тестирования программы с состоянием является исследование ее пространства состояний (см. лекцию 11). В отличие от верификации моделей, в тестировании программ имеется принципиальное ограничение – программу нельзя (по крайней мере очень трудно) перевести в произвольное состояние. В связи с этим исследование пространства состояний осуществляется путем (последовательного) обхода графа состояний – построения маршрута, содержащего все дуги графа (переходы программы).

Пусть S – множество всех возможных состояний программы; X – множество тестовых воздействий (стимулов), при этом для каждого $x \in X$ задано предусловие φ_x – предикат на S; $T \subseteq S \times X \times S$ – отношение переходов, конструируемое в процессе обхода. Введем вспомогательные обозначения:

- $T[s, x] \equiv \{s' \in S \mid (s, x, s') \in T\}$;
- $New(s) \equiv \{x \in X \mid s \vDash \varphi_x \wedge T[s, x] = \emptyset\}$.

Работу генератора стимулов можно представить так:

```
T := ∅;                          // множество пройденных переходов
while true do                    // главный цикл генератора:
  s := getState();               //   берется текущее состояние
  if New(s) ≠ ∅ then             //   если есть еще не примененные стимулы,
    x := choose(New(s));         //     выбирается один из них и
    applyStimulus(x);            //     подается на программу;
    s' := getState();            //     множество пройденных переходов
    T := T ∪ {(s, x, s')}        //     пополняется
  else                           //   иначе
    p := getPath(s, T);          //     строится путь в состояние,
                                 //     где есть непримененные стимулы;
    if p = ∅ then                //     если такого состояния нет,
      break                      //     тестирование прекращается
    end;
    for x ∈ p do                 //   подаются тестовые воздействия,
      applyStimulus(x)           //   определенные вдоль пути
    end
  end
end
```

В каждом состоянии генератор ищет новые, еще не примененные в этом состоянии тестовые воздействия и, если находит, подает одно из них (*applyStimulus*). Если таких воздействий нет, запускается вложенный поиск (*getPath*), возвращающий путь из текущего состояния в состояние, в котором есть непройденные переходы (непримененные воздействия); при отсутствии таких состояний процесс завершается. Вложенный поиск может быть реализован разными способами: как в глубину, так и в ширину. Для применимости метода достаточно, чтобы граф состояний был *сильно связным*, т. е. имел одну компоненту сильной связности (см. лекцию 12), и *детерминированным*: для каждого состояния s и стимула x должно существовать не более одного перехода вида (s, x, s') [Бур03а].

16.3.2. Абстракция состояний программы

В этой книге мы сознательно избегали динамических структур данных: их использование усложняет состояния программы и делает пространство состояний необозримым. Для тестирования приходится искусственно ограничивать поиск. Как и в формальной верификации, для этого применяются методы абстракции (см. лекцию 11). Поясним сказанное на простом примере.

Пример 16.5

Рассмотрим структуру данных *очередь* (для определенности будем считать, что элементами очереди являются целые числа, хотя это непринципиально). Над очередью определены две операции:

- *enq*(*q*, *x*) – добавить элемент *x* в хвост очереди *q*;
- *deq*(*q*) – извлечь элемент из головы непустой очереди *q*.

Используем функцию абстракции *f*(*q*) = *size*(*q*), где *size*(*q*) – размер очереди *q*. Другими словами, мы считаем, что очереди эквивалентны тогда и только тогда, когда они содержат одинаковое число элементов. Зададим ограничение: *size*(*q*) ≤ 2 (если содержит два элемента, применение операции *enq* запрещено).

На рис. 16.2 показан фрагмент графа состояний очереди и граф состояний абстрактной модели: дуги, ведущие вниз, соответствуют вызовам *enq*; дуги, ведущие вверх, – *deq*. Тестирование очереди сводится к обходу графа состояний абстрактной модели: для этого достаточно четырех вызовов: *enq*, *enq*, *deq*, *deq*.

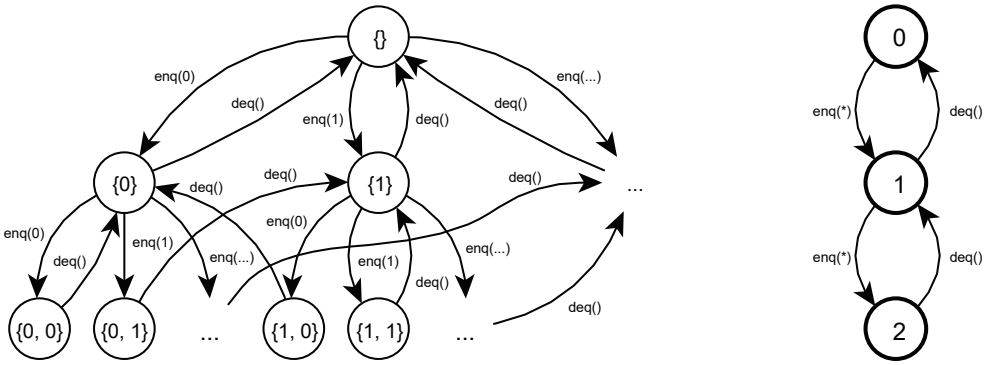


Рис. 16.2 ❖ Фрагмент графа состояний очереди (слева) и его абстракция (справа)

□

16.4. Тестирование реагирующих систем и параллельных программ

Для описания свойств реагирующих систем используется логика LTL (см. лекцию 9). Ранее мы рассмотрели, как проверить истинность формулы LTL на модели программы – структуре Крипке с конечным числом состояний. Можно ли LTL-спецификации использовать для проверки поведения реальных систем в динамике? Исследования по этой теме ведутся в рамках такого направления, как *мониторинг времени исполнения* (*runtime monitoring*, *runtime verification*) [Bau11]. Идея в следующем: по автомату Бюхи для формулы LTL (см. лекцию 13), или непосредственно по формуле LTL, строится контрольный автомат (монитор), который на каждом шаге возвращает статус проверки: *undef*, *true* или *false*. Как только статус становится определенным (*true* или *false*), монитор завершается (в случае *false* с выдачей сообщения об

ошибке). При построении контрольных автоматов по недетерминированным автоматам Бюхи применяют алгоритмы детерминизации, близкие к методам построения ДКА по НКА [Сер12].

Методы проверки моделей могут применяться для тестирования параллельных программ. Для этого используют техники *управляемого исполнения*. Программа инструментируется: в точки, в которых осуществляется взаимодействие процессов с окружением или другими процессами (обращение к разделяемой переменной, прием или посылка сообщения и т. п.), добавляется код, который может приостанавливать исполнение. Решения о приостановке тех или иных процессов принимаются специальным планировщиком, работающим в составе тестовой системы; его целью является обход пространства состояний, т. е. систематический перебор возможных вариантов исполнения. Примером инструмента, работающего по этой схеме, является Java PathFinder (JPF) – средство проверки программ на байт-коде Java [JPF]. Возможен и другой вариант, когда планировщик не отслеживает состояния процессов системы, а воспроизводит трассы, полученные при верификации ее модели, например с помощью инструмента Spin [Spin].

16.5. Заключение

В заключение лекции и книги в целом повторим мысль, высказанную в самом начале. Существует множество подходов к верификации программ и аппаратуры, однако задача верификации настолько трудна (и ответственна), что ни один из них, взятый в отдельности, не является панацеей: удовлетворительные результаты достигаются только при совместном использовании разных техник (проверка моделей, статический анализ, фаззинг, тестирование, дедуктивная верификация и др.).

Формальные методы глубоко укоренились в программной и компьютерной инженерии, стали обязательным инструментом разработки систем ответственного назначения. Нет никаких сомнений в том, что формальные методы будут совершенствоваться. Появляются и новые направления, главное из которых – применение нейронных сетей, прежде всего больших языковых моделей (LLM, Large Language Models). Уверены, что в ближайшее время нас ждут удивительные технологии разработки и анализа программ, сочетающие методы математической логики и искусственного интеллекта.

16.6. 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 символьного 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%-го 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
ПК	персональный компьютер
ПНФ	предваренная нормальная форма
ПО	программное обеспечение
ССФ	скулемовская 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)
C/DC	покрытие условий и решений (condition/decision coverage)
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–Патнэм–Логеман–Лавленд (Davis–Putnam–Logemann–Loveland)[1]
HDL	язык описания аппаратуры (hardware description language)
LLM	большая языковая модель (large language model)
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)

[1] Семейство алгоритмов решения задачи выполнимости КНФ.

NDFS	вложенный поиск в глубину (nested depth-first search)
NP	недетерминированно полиномиальный (non-deterministic polynomial)[2]
ROBDD	сокращенная упорядоченная двоичная решающая диаграмма (reduced ordered binary decision diagram)
S1S	монадическая логика второго порядка с одной функцией следования (second-order logic with 1 successor)
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)
ZDD	двоичная решающая диаграмма с подавлением нулей (zero-suppressed decision diagram)

Обозначения

Множества и отношения

$\emptyset$ или {}	пустое множество
$\{x_1, x_2, \ldots\}$	множество, состоящее из элементов $x_1, x_2, \ldots$
$(x_1, \ldots, x_n)$	кортеж (n-ка) из элементов $x_1, \ldots, x_n$
$\{x_i\}_{i=1}^n$	последовательность из элементов $x_1, \ldots, x_n$
$x \in X$ ($x \notin X$)	принадлежность (непринадлежность) элемента x множеству X
$X \subseteq Y$ ($X \nsubseteq Y$)	включение (невключение) множества X в множество Y
$X \subset Y$	строгое включение множества X в множество Y ($X \subseteq Y$ и $X \neq Y$)
$\lvert X \rvert$	мощность конечного множества
$\bar{X}$	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-я 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 нем отношением частичного порядка $\prec \subseteq X^2$ (частично упорядоченное множество)
R^*	рефлексивно-транзитивное замыкание бинарного отношения R

[2] Класс задач, допускающих решение за полиномиальное время на недетерминированной машине Тьюринга.

$Im(R)$	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: $Dom(R \rhd Y)$
χ_X	характеристическая функция множества (отношения) X

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
$\models \varphi$ ($\not\models \varphi$)	общезначимость (необщезначимость) формулы φ
$I \models \varphi$ ($I \not\models \varphi$)	истинность (ложность) формулы φ в интерпретации I
$\neg\varphi$ или $\bar{\varphi}$	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)
$\varphi \rightarrow \psi$	импликация из формулы φ формулы ψ (если φ, то ψ)
$\varphi \leftrightarrow \psi$ или $\varphi \equiv \bar{\varphi}$	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 ($\bar{p}$ для p, p для $\bar{p}$)
$\varphi \approx \psi$	равносильность формул φ и ψ (одновременная выполнимость/невыполнимость)
$[x_1 := t_1, \ldots, x_n := t_n]$	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(φ)	замыкание формулы φ (множество всех подформул и их отрицаний)
atoms(φ)	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
$v \cdot w$ или vw	конкатенация конечного слова v и слова w
w^n	n-я степень конечного слова w: $w^0 = \varepsilon$; если $n \geq 1$, то $w^n = w \cdot w^{n-1}$
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
$A \otimes B$	прямое произведение 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

$\mathbb{B}$	булево множество: {0, 1}
$\mathbb{N}$	множество натуральных чисел: {1, 2, ...}
$\mathbb{N}_0$	множество натуральных чисел, расширенное 0: {0, 1, ...}
$\mathbb{Z}$	множество целых чисел: {0, ±1, ±2, ...}
$\mathbb{Q}$	множество рациональных чисел: $\left\{\frac{x}{y} \middle\| x \in \mathbb{Z}, y \in \mathbb{N}\right\}$
$\mathbb{Q}_+$	множество положительных рациональных чисел: $\left\{\frac{x}{y} \middle\| x \in \mathbb{N}, y \in \mathbb{N}\right\}$
$[x, y]$	отрезок («интервал») 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 («пол»)
$\lceil x \rceil$	целая часть числа x сверху («потолок»)
$\min(x_1, \ldots, x_n)$	минимальное из чисел $x_1, \ldots, x_n$
$\max(x_1, \ldots, x_n)$	максимальное из чисел $x_1, \ldots, x_n$
НОД(x, y)	наибольший общий делитель чисел x и y
$\sim x$	побитовое НЕ 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 $\boxed{1}$
$L(n)$	переменная, сопоставленная (нелистовой) вершине n (пометка вершины n)
$high(n)$	конец дуги, исходящей из (нелистовой) вершины n и соответствующей условию $L(n) = 1$
$low(n)$	конец дуги, исходящей из (нелистовой) вершины n и соответствующей условию $L(n) = 0$
$root(F)$	корень диаграммы F
$L(F)$	переменная, сопоставленная корню нетривиальной диаграммы F: $L(root(F))$
$then(F)$	поддиаграмма нетривиальной диаграммы F с корнем в $high(root(F))$
$else(F)$	поддиаграмма нетривиальной диаграммы F с корнем в $low(root(F))$
$Reduce(F)$	результат применения правил сокращения к диаграмме F
$Compose(x, T, E)$	результат композиции диаграмм T и E по переменной x: $root(C) = x$, $then(C) = T$, $else(C) = E$, где $C = Compose(x, T, E)$
$Apply(F \circ G)$	$Reduce(Compose(x, Apply(F_{\|x=1} \circ G_{\|x=1}), Apply(F_{\|x=0} \circ G_{\|x=0})))$
$[\![F]\!]$	функция, реализуемая диаграммой F

Программирование

ε	пустая программа
skip	пустой оператор (не изменяет состояния программы)
$x := t$	оператор присваивания: x – переменная, t – выражение (терм)
$P; Q$	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
$sp(P, \varphi)$	сильнейшее постусловие программы P относительно предусловия φ
$ssa(P)$	SSA-представление программы P
$\{\varphi\}$	утверждение φ истинно в соответствующей точке программы
{**inv**: φ}	φ – инвариант соответствующего цикла
{**bd**: u, $\langle M, \prec\rangle$}	u – ограничивающая функция цикла ($\langle M, \prec\rangle$ – фундированное множество)
$M[\![P]\!]$	семантика программы P: $M[\![P]\!]: S \to 2^S$, 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
start: $x := t$	начальный оператор блок-схемы: x – переменная, t – выражение
assign: $x := t$	оператор присваивания блок-схемы: x – переменная, t – выражение
test: B	условный оператор блок-схемы: B – логическая формула
join	оператор соединения блок-схемы
halt: $x := t$	завершающий оператор блок-схемы: x – переменная, t – выражение
ϵ	пустая пометка дуги блок-схемы
$P \mid\mid Q$	параллельная композиция процессов P и Q
$P@l$	пребывание процесса P в точке l (l – состояние управления P)
$\mathcal{R}(P)$	символьное представление программы P

Список источников

Книги и статьи

[Адж98] *Аджиев В.* Мифы о безопасном ПО: уроки знаменитых катастроф // Открытые системы. 1998. № 6 (http://www.osp.ru/os/1998/06/179592/).

[Але20] *Алексеев В. Б.* Лекции по дискретной математике. М.: Инфра-М; Высшее образование: Бакалавриат, 2020. 90 с.

[Буз14] *Буздалов Д. В., Корныхин Е. В., Панферов А. А., Петренко А. К., Хорошилов А. В.* Практикум по дедуктивной верификации программ. М.: Изд. отдел фак-та ВМК МГУ им. М. В. Ломоносова; МАКС Пресс, 2014. 100 с.

[Бур03a] *Бурдонов И. Б., Косачев А. С., Кулямин В. В.* Неизбыточные алгоритмы обхода ориентированных графов. Детерминированный случай // Программирование. 2003. № 5. С. 11–30. (Английский перевод: I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin. *Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case* // Programming and Computer Software, 29 (5), 2003. P. 245–258. DOI: 10.1023/A:1025733107700.)

[Бур03b] *Бурдонов И. Б., Косачев А. С., Кулямин В. В., Петренко А. К.* Подход UniTesK к разработке тестов // Программирование. 2003. № 6. С. 25–43. (Английский перевод: V. V. Kuliamin, A. K. Petrenko, A. S. Kossatchev, I. B. Burdonov. *The UniTesK Approach to Designing Test Suites* // Programming and Computer Software, 29 (6), 2003. P. 310–322. DOI: 10.1023/B:PACS.0000004131.54590.fb.)

[Бур11] *Бурдонов И. Б., Грошев С. Г., Демаков А. В., Камкин А. С., Косачев А. С., Сортов А. А.* Параллельное тестирование больших автоматных моделей // Вестник ННГУ. 2011. № 3. С. 187–193.

[Вел11] *Вельдер С. Э., Лукин М. А., Шалыто А. А., Яминов Б. Р.* Верификация автоматных программ. СПб: Наука, 2011. 244 с.

[Евк50] Начала Евклида / перев. с греч. и комм. Д. Д. Мордухая-Болтовского; под ред. М. Я. Выгодского и И. Н. Веселовского. М.; Л.: ГИТТЛ; Т. 1. 1948; Т. 2. 1949; Т. 3. 1950.

[Ерш87] *Ершов Ю. Л., Палютин Е. А.* Математическая логика. М.: Наука, 1987. 336 с.

[Зах12] *Захаров В. А.* Проблема эквивалентности программ: модели, алгоритмы, сложность: дис. … докт. физ.-мат. наук. ВМК МГУ, 2012. 438 с.

[Кам15] *Камкин А. С.* Проецирование систем переходов: преодоление комбинаторного взрыва при верификации параллельных систем // Программирование. 2015. № 6. С. 53–71. (Английский перевод: A. S. Kamkin. *Projecting Transition Systems: Overcoming State Explosion in Concurrent System Verification* // Programming and Computer Software, 41 (6), 2015. P. 311–324. DOI: 10.1134/S0361768815060043.)

[Кар03] *Карпов Ю. Г.* Теория автоматов. СПб.: Питер, 2003. 208 с.

[Кар10] *Карпов Ю. Г.* Model Checking. Верификация параллельных и распределенных программных систем. СПб.: БХВ-Петербург, 2010. 560 с.

[Кас03] *Касьянов В. Н., Евстигнеев В. А.* Графы в программировании: обработка, визуализация и применение. СПб.: БХВ-Петербург, 2003. 1104 с.

[Кот91] *Котов В. Е., Сабельфельд В. К.* Теория схем программ. М.: Наука, 1991. 248 с.

[Кро01] *Кронрод А. С.* Беседы о программировании. М.: Едиториал УРСС, 2001. 248 с.

[Кул08] *Кулямин В. В.* Методы верификации программного обеспечения / Всероссийский конкурс обзорно-аналитических статей по приоритетному направлению «Информационно-телекоммуникационные системы». 2008. 117 с. (http://www.ict.edu.ru/ft/005645/62322e1-st09.pdf).

[Лав09] *Лавров И. А., Максимова Л. Л.* Задачи по теории множеств, математической логике и теории алгоритмов. М.: ФИЗМАТЛИТ, 2009. 256 с.

[Ман13] *Мандрыкин М. У., Мутилин В. С., Хорошилов А. В.* Введение в метод CEGAR – уточнение абстракции по контрпримерам // Труды ИСП РАН, 24. 2013. С. 219–292. DOI: 10.15514/ISPRAS-2013-24-12.

[Мас64] *Маслов С. Ю.* Обратный метод установления выводимости в классическом исчислении // ДАН СССР, 159 (1). 1964. С. 17–20.

[Мир23] *Миронов А. М.* Методы верификации программ. М.: ДМК Пресс, 2023. 336 с.

[Неп88] *Непомнящий В. А., Рякин О. М.* Прикладные методы верификации программ. М.: Радио и связь, 1988. 256 с.

[Сер12] *Серебряков В. А.* Теория и реализация языков программирования. М.: ФИЗМАТЛИТ, 2012. 236 с.

[Тра70] *Трахтенброт Б. А., Барздинь Я. М.* Конечные автоматы (поведение и синтез). М.: Наука, 1970. 400 с.

[Уба76] *Убар Р.* Генерация тестов цифровых схем с использованием альтернативных графов // Труды Таллинского технического университета. № 409. 1976. С. 75–81.

[Усп87] *Успенский В. А., Семенов А. Л.* Теория алгоритмов: основные открытия и приложения. М.: Наука, 1987. 288 с.

[Цей68] *Цейтин Г. С.* О сложности вывода в исчислении высказываний // Записки научного семинара ЛОМИ, 8. 1968. С. 234–259.

[Шок81] *Шокин Ю. И.* Интервальный анализ. М.: Наука, 1981. 112 с.

[Ябл86] *Яблонский С. В.* Эквивалентные преобразования управляющих систем. М.: МГУ, 1986. 40 с.

[Bla05] A. Blass, Yu. Gurevich, L. Nachmanson, M. Veanes. *Play to Test*. Technical Report MSR-TR-2005-04, 2005 (http://research.microsoft.com/apps/pubs/default.aspx?id=70129).

[Aho74] A. Aho, J. Hopcroft, J. Ullman. *Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. 470 p. (Русский перевод: *Ахо А. В., Хопкрофт Д. Э., Ульман Дж. Д.* Разработка и анализ компьютерных алгоритмов. М.: Вильямс, 2021. 544 с.)

[Aho06] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006. 1000 p. (Русский перевод: *Ахо А. В., Лам М. С., Сети Р., Ульман Дж. Д.* Компиляторы: принципы, технологии и инструментарий. М.: Вильямс, 2014. 1184 с.)

[Ake78] S. B. Akers. *Binary Decision Diagrams*. IEEE Transactions on Computers, C-27 (6), 1978. P. 509–516. DOI: 10.1109/TC.1978.1675141.

[And79] R. B. Anderson. *Proving Programs Correct*. John Wiley & Sons, 1979. (Русский перевод: *Андерсон Р.* Доказательство правильности программ. М.: Мир, 1982. 287 с.)

[Apt09] K. R. Apt, F. S. de Boer, E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 2009. 502 p.

[Bai08] C. Baier, J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008. 984 p.

[Bau10] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto. *ACSL: ANSI/ISO C Specification Language*. Version 1.4. 2010.

[Bau11] A. Bauer, M. Leucker, C. Schallhart. *Runtime Verification for LTL and TLTL*. ACM Transactions on Software Engineering and Methodology, 20 (4), 2011. P. 14:1–14:64. DOI: 10.1145/2000799.2000800.

[Bei90] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990. 550 p.

[Bei95] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley, 1995. 320 p. (Русский перевод: *Бейзер Б.* Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем. СПб.: Питер, 2004. 320 с.)

[Ben10] M. Ben-Ari. *Principles of the Spin Model Checker. Supplementary Material on Spin Version 6*, 2010. 7 p.

[Ben12] M. Ben-Ari. *Mathematical Logic for Computer Science*. Springer, 2012. 354 p.

[Bie03] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu. *Bounded Model Checking*. Advances in Computers, 58 (3), 2003. P. 117–148. DOI: 10.1016/S0065-2458(03)58003-2.

[Bir67] G. Birkhoff. *Lattice Theory*. American Mathematical Society, 1967. 418 p. (Русский перевод: *Биркгоф Г.* Теория решеток. М.: Наука. Главная редакция физико-математической литературы, 1984. 568 с.)

[Boe01] B. Boehm, V. R. Basili. Software Defect Reduction Top 10 List // *Computer*, 34 (1). 2001. P. 135–137. DOI: 10.1109/2.962984.

[Boz19] M. Bozzano, R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta. nuXmv 2.0.0 User Manual. 2019. 190 p.

[Bro95] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, Anniversary edition, 1995. 336 p. (Русский перевод: *Брукс Ф.* Мифический человеко-месяц, или Как создаются программные системы. СПб.: Символ-Плюс, 2013. 304 с.)

[Bry86] R. E. Bryant. *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers, C-35 (8). 1986. P. 677–691. DOI: 10.1109/TC.1986.1676819.

[Buc60] J. R. Büchi. *On a Decision Method in Restricted Second Order Arithmetic*. International Congress on Logic, Method, and Philosophy of Science. 1960. P. 1–12. DOI: 10.1007/978-1-4613-8928-6_23.

[Bur90] J. R. Burch, E. M. Clarke, K. L. McMillan. *Symbolic Model Checking: 10^{20} States and Beyond*. Proceedings of the 5th IEEE Symposium on Logic in Computer Science (LICS). 1990. P. 428–439. DOI: 10.1109/LICS.1990.113767.

[Bur99] I. Burdonov, A. Kossatchev, A. Petrenko, D. Galter. *KVEST: Automated Generation of Test Suites from Formal Specifications*. Proceedings of Formal Methods: World Congress on Formal Methods in the Development of Computing Systems (FM), LNCS 1708. 1999. P. 608–621. DOI: 10.1007/3-540-48119-2_34.

[Bur16] J. Burghardt, J. Gerlach, T. Lapawczyk. *ACSL By Example. Towards a Verified C Standard Library*. Version 12.1.0 for Frama-C (Magnesium), 2016. 155 p. (https://www.cs.umd.edu/class/spring2016/cmsc838G/frama-c/ACSL-by-Example-12.1.0.pdf).

[Cav14] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta. *The nuXmv Symbolic Model Checker*. Proceedings of the 16th International Conference on Computer Aided Verification (CAV), LNCS 8559, 2014. P. 334-342. DOI: 10.1007/978-3-319-08867-9_22.

[Cav15] R. Cavada, A. Cimatti, C.A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, A. Tchaltsev. *NuSMV 2.6 User Manual*. 2015. 144 p. (https://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf).

[CavTu] R. Cavada, A. Cimatti, G. Keighren, E. Olivetti, M. Pistore, M. Roveri. NuSMV 2.6 Tutorial. 32 p. (https://nusmv.fbk.eu/NuSMV/tutorial/v26/tutorial.pdf).

[Cha73] C.-L. Chang, R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973. 331 p. (Русский перевод: *Чень Ч., Ли Р.* Математическая логика и автоматическое доказательство теорем. М.: Наука, 1983. 358 с.)

[Chi94] J. J. Chilenski, S. P. Miller. *Applicability of Modified Condition/Decision Coverage to Software Testing*. Software Engineering Journal, 7 (5), 1994. P. 193–200. DOI: 10.1049/sej.1994.0025.

[Cla03] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. *Counterexample-Guided Abstraction Refinement*. Proceedings of the International Computer Aided Verification (CAV), LNCS 1855, 2000. P. 154–169. DOI: 10.1007/10722167_15.

[Cla86] E. M. Clarke, E. A. Emerson, A. P. Sistla. *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*. ACM Transactions on Programming Languages and Systems, 8 (2), 1986. P. 244–263. DOI: 10.1145/5397.5399.

[Cla99] E. M. Clarke, O. Grumberg, D. A. Peled. *Model Checking*. The MIT Press, 1999. 314 p. (Русский перевод: *Кларк Э. М., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО, 2002. 416 с.)

[Cou77] P. Cousot, R. Cousot: *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL), 1977. P. 238–252. DOI: 10.1145/512950.512973.

[Cou78] P. Cousot, N. Halbwachs. *Automatic Discovery of Linear Restraints among Variables of a Program*. Conference Record of the 5th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 1978. P. 84-97. DOI: 10.1145/512760.512770.

[Cra57] W. Craig. *Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem*. Journal of Symbolic Logic, 22, 1957. P. 250–268. DOI: 10.2307/2963593.

[Dah72] O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare. *Structured Programming*. Academic Press, 1972. 220 p. (Русский перевод: *Дал У., Дейкстра Э., Хоор К.*[1] Структурное программирование. М.: Мир, 1975. 248 с.)

[1] Это не опечатка: это – странный перевод имени Чарльза Энтони Ричарда (Тони) Хоара на русский.

[Dav60] M. Davis, H. Putnam. *A Computing Procedure for Quantification Theory*. Journal of the ACM 7 (3), 1960. P. 201–215. DOI: 10.1145/321033.321034.

[Dav62] M. Davis, G. Logemann, D. Loveland. *A Machine Program for Theorem Proving*. Communications of the ACM 5 (7), 1962. P. 394–397. DOI: 10.1145/368273.368557.

[Dij75] E. W. Dijkstra. *Guarded Commands, Non-determinacy and Formal Derivation of Programs*. Communications of ACM, 18 (8), 1975. P. 453–457. DOI: 10.1145/360933.360975.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976. 232 p. (Русский перевод: *Дейкстра Э.* Дисциплина программирования. М.: Мир, 1978. 275 с.)

[DO178B] RTCA/DO-178B. *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Inc., 1992.

[Flo67] R. W. Floyd. *Assigning Meaning to Programs*. Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics, 19, 1967. P. 19–32. DOI: 10.1090/psapm/019/0235771.

[Gla02] R. L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, 2002. – 224 p. (Русский перевод: *Р. Гласс.* Факты и заблуждения профессионального программирования. СПб.: Символ-Плюс, 2007. 240 с.)

[God94] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. PhD Thesis, University of Liege, Computer Science Department, 1994. 136 p.

[God05] P. Godefroid, N. Klarlund, K. Sen. *DART*: *Directed Automated Random Testing*. Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 40 (6), 2005. P. 213–223. DOI: 10.1145/1064978.1065036.

[Gri81] D. Gries. *The Science of Programming*. Springer Verlag, 1981. 388 p. (Русский перевод: *Грис Д.* Наука программирования. М.: Мир, 1984. 416 с.)

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, 1992. 441 p.

[Har09] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009. 702 p.

[Har12] D. Harris, S. Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann, 2012. – 720 p. (Русский перевод: *Харрис Д., Харрис С.* Цифровая схемотехника и архитектура компьютера. М.: ДМК Пресс, 2018. 792 с.)

[Hay01] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, L. K. Rierson. *A Practical Tutorial on Modified Condition/Decision Coverage*. NASA/TM-2001-210876, 2001. 85 p. (https://shemesh.larc.nasa.gov/fm/papers/Hayhurst-2001-tm210876-MCDC.pdf).

[Hil99] D. Hilbert. *Grundlagen der Geometrie*. Festschrift zur Feier der Enthüllung des Gauss-Weber-Denkmals in Göttingen. Teubner, 1899. (Русский перевод: *Гильберт Д.* Основания геометрии. Л.: Сеятель, 1923. 152 с.)

[Hoa69] C. A. R. Hoare. *An Axiomatic Basis for Computer Programming*. Communications of the ACM, 12(10), 1969. P. 576–585. DOI: 10.1145/363235.363259.

[Hol03] G. J. Holzmann. *The SPIN Model Checker. Primer and Reference Manual*. Addison-Wesley, 2003. 608 p.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991. 512 p.

[Hol97] G. J. Holzmann. *State Compression in Spin: Recursive Indexing and Compression Training Runs*. Proceedings of the 3rd International Spin Workshop, 1997. 10 p. (http://spinroot.com/spin/Workshops/ws97/gerard.pdf).

[Hol98] G. J. Holzmann. *An Analysis of Bitstate Hashing*. Formal Methods in System Design, 13 (3), 1998. P. 287–307. DOI: 10.1023/A:1008696026254.

[Ker88] B. W. Kernighan, D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. 272 p. (Русский перевод: *Керниган Б., Ритчи Д.* Язык программирования С. М.: Изд. дом «Вильямс», 2006. 304 с.)

[Kes98] Y. Kesten, A. Pnueli, L. Raviv. *Algorithmic Verification of Linear Temporal Logic Specifications*. Proceedings of International Colloquium on Automata, Languages, and Programming (ICALP), LNCS 1443, 1998. P. 1–16. DOI: 10.1007/BFb0055036.

[Kri63] S. A. Kripke. *Semantical Considerations on Modal Logic*. Acta Philosophica Fennica, 16, 1963. P. 83–94.

[Kro08] D. Kroening, O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008. 306 p.

[Lee59] C. Y. Lee. *Representation of Switching Circuits by Binary-Decision Programs*. The Bell System Technical Journal, 38 (4), 1959. P. 985–999. DOI: 10.1002/j.1538-7305.1959.tb01585.x. (Русский перевод: *Ли*. Представление переключательных схем с помощью программ двоичного решения // Вопросы теории математических машин: сборник статей: пер. с англ. / под ред. И. Я. Акушского и Э. А. Глузберга. М.: Машиностроение, 1964. С. 219–232.)

[Lei23] K. R. M. Leino. *Program Proofs*. The MIT Press, 2023. 496 p. (Русский перевод: *Лейно К.* Доказательство корректности программ. М.: ДМК Пресс, 2024. 530 с.)

[Llo07] S. Lloyd. *Programming the Universe: A Quantum Computer Scientist Takes on the Cosmos*. Vintage, 2007. – 256 p. (Русский перевод: *Лойд С.* Программируя Вселенную: Квантовый компьютер и будущее науки. М.: Альпина Нон-фикшн, 2014. 256 с.)

[Mah04] Y. S. Mahajan, Z. Fu, S. Malik. *Zchaff2004: An Efficient SAT Solver*. Theory and Applications of Satisfiability Testing (SAT), LNCS 3542, 2004. P. 360-375. DOI: 10.1007/11527695_2.

[Mar18] C. Marché, Y. Moy. *The Jessie Plugin for Deductive Verification in Frama-C*. Tutorial and Reference Manual. Version 2.40. 2018. 27 p. (http://krakatoa.lri.fr/jessie.pdf).

[Mar82] A. Martelli, U. Montanari. *An Efficient Unification Algorithm*. ACM Transactions on Programming Languages and Systems, 4 (2), 1982. P. 258-282. DOI: 10.1145/357162.357169.

[Mar96] J. P. Marques-Silva, K. A. Sakallah. *GRASP – A New Search Algorithm for Satisfiability*. Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (CAD), 1996. P. 220–227. DOI: 10.1109/ICCAD.1996.569607.

[Mcc04] S. McConnell. *Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2004. 960 p. (Русский перевод: *Макконнелл С.* Совершенный код: практическое руководство по разработке программного обеспечения. М.: Русская Редакция, 2016. 896 с.)

[Mcc62] J. McCarthy. *Towards a Mathematical Science of Computation*. Proceedings of IFIP Congress, 1962. P. 21–28.

[Mcm01] K. McMillan. *Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking*. Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME), 2001. P. 179–195. DOI: 10.1007/3-540-44798-9_17.

[Mcm92] K. L. McMillan. *Symbolic Model Checking. An Approach to the State Explosion Problem*. Doctoral Dissertation, Carnegie Mellon University, 1992. 214 p.

[Men64] E. Mendelson. *Introduction to Mathematical Logic*. D. Van Nostrand Company, Inc., 1964. 300 p. (Русский перевод: *Мендельсон Э.* Введение в математическую логику. М.: Наука, 1976. 320 с.)

[Min13] Y. Minsky, A. Madhavapeddy, J. Hickey. *Real World OCaml: Functional Programming for the Masses*. O'Reilly Media, 2013. 510 p. (Русский перевод: *Мински Я., Мадхавапедди А., Хикки Дж.* Программирование на языке OCaml. М.: ДМК Пресс, 2014. 536 с.)

[Mit96] J. C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1995. 845 p. (Русский перевод: *Митчелл Дж.* Основания языков программирования. М.; Ижевск: НИЦ «Регулярная и хаотическая динамика», 2010. 720 с.)

[Mon09] D. Monniaux. *A Minimalistic Look at Widening Operators*. Higher-Order and Symbolic Computation, 22 (2), 2009. P. 145–154. DOI: 10.1007/s10990-009-9046-8.

[Muk12] M. Mukund. *Finite-State Automata on Infinite Inputs*. Modern Applications of Automata Theory, 2012. P. 45–78. DOI: 10.1142/9789814271059_0002.

[Mul63] D. E. Muller. *Infinite Sequences and Finite Machines*. Proceedings of the 4th IEEE Symposium on Switching Circuit Theory and Logical Design, 1963. P. 3–16. DOI: 10.1109/SWCT.1963.8.

[Nel79] G. Nelson, D.C. Oppen. *Simplification by Cooperating Decision Procedures*. ACM Transactions on Programming Languages and Systems, 2 (1), 1979. P. 245–257. DOI: 10.1145/357073.357079.

[Nel80] G. Nelson, D. C. Oppen. *Fast Decision Procedures Based on Congruence Closure*. Journal of the ACM, 27 (2), 1980. P. 356–364. DOI: 10.1145/322186.322198.

[Nie99] F. Nielson, H. R. Nielson, C. Hankin. *Principles of Program Analysis*. Springer, 1999. 473 p.

[Nie07] R. Nieuwenhuis, A. Olivas. *Fast Congruence Closure and Extensions*. Information and Computation, 205 (4), 2007. P. 557–580. DOI: 10.1016/j.ic.2006.08.009.

[Ost88] T. J. Ostrand, M. J. Balcer. *The Category-Partition Method for Specifying and Generating Functional Tests*. Communications of the ACM, 31 (6), 1988. P. 676–686. DOI: 10.1145/62959.62964.

[Pap82] C. H. Papadimitriou, K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Courier Corporation, 1982. 496 p. (Русский перевод: *Пападимитриу Х., Стайглиц К.* Комбинаторная оптимизация. Алгоритмы и сложность. М.: Мир, 1985. 512 с.)

[Pea89] G. Peano. *Arithmetices principia, nova methodo exposita*. Fratres Bocca, 1889. 20 p.

[Pet81] G. L. Peterson. *Myths About the Mutual Exclusion Problem*. Information Processing Letters, 12 (3), 1981. P. 115–116. DOI: 10.1016/0020-0190(81)90106-X.

[Pet15] A. K. Petrenko, B.-H. Schlingloff, N. Pakulin. *Preface*. Proceedings of the 10th Model-Based Testing Workshop (MBT), EPTCS 180, 2015. P. 1–6. DOI: 10.4204/EPTCS.180.0

[Plo81] G. Plotkin. *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[Pnu77] A. Pnueli. *The Temporal Logic of Programs*. Proceedings of the 18th IEEE Symposium on Foundation of Computer Science (FOCS), 1977. P. 46–57. DOI: 10.1109/SFCS.1977.32.

[PSL10] 1850-2010 – *IEEE Standard for Property Specification Language* (*PSL*), 2010. DOI: 10.1109/IEEESTD.2010.5446004.

[Rab69] M. O. Rabin. *Decidability of Second Order Theories and Automata on Infinite Trees*. Transaction of the AMS 141, 1969. P. 1–35. DOI: 10.2307/1995086. JSTOR 1995086.

[Rai12] C. Raistrick, P. Francis, J. Wright, C. Carter, I. Wilkie. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004. 412 p.

[Rob65] J. A. Robinson. *A Machine-Oriented Logic Based on the Resolution Principle*. Journal of the ACM, 12 (1), 1965. P. 23–41. DOI: 10.1145/321250.321253.

[Roz10] K. Y. Rozier. *Linear Temporal Logic Symbolic Model Checking*. Computer Science Review, 2010. 41 p. DOI: 10.1016/j.cosrev.2010.06.002.

[Sen05] K. Sen, D. Marinov, G. Agha. *CUTE: A Concolic Unit Testing Engine for C*. Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), 2005. P. 263–272. DOI: 10.1145/1081706.1081750.

[Tar72] R. Tarjan. *Depth-First Search and Linear Graph Algorithms*. SIAM Journal on Computing, 1 (2), 1972 P. 146–160. DOI: 10.1137/0201010.

[Tho96] W. Thomas. *Languages, Automata, and Logic*. Bericht 9607, Institut für Informatik und Praktische Mathematik der Christian-Albrechts-Universität zu Kiel, D-24098 Kiel, 1996. 75 p.

[Tre08] J. Tretmans. *Model Based Testing with Labelled Transition Systems*. Formal Methods and Testing, LNCS 4949, 2008. P. 1–38. DOI: 10.1007/978-3-540-78917-8_1.

[Tur36] A. M. Turing. *On Computable Numbers, with an Application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, Series 2, 42 (1), 1936. P. 230–265. (Errata appeared in Series 2, 43, 1937. P. 544–546.) DOI: 10.1112/plms/s2-42.1.230.

[Utt12] M. Utting, A. Pretschner, B. Legeard. *A Taxonomy of Model-Based Testing Approaches*. Software Testing, Verification & Reliability, 22 (5), 2012. P. 297–312. DOI: 10.1002/stvr.456.

[Var86] M. Y. Vardi, P. Wolper. *An Automata-Theoretic Approach to Automatic Program Verification*. Proceedings of the 1st Symposium on Logic in Computer Science (LICS), 1986. P. 332–344 (http://spinroot.com/spin/Doc/lics86.pdf).

[Wol02] P. Wolper. *Constructing Automata from Temporal Logic Formulas: A Tutorial*. Lectures on Formal Methods and Performance Analysis, 2002. P. 261–277.

[Yua06] J. Yuan, C. Pixley, A. Aziz. *Constraint-Based Verification*. Springer, 2006. 254 p.

[Zha03] L. Zhang. *Searching For Truth: Techniques For Satisfiability of Boolean Formulas*. PhD Thesis, Princeton University, 2003. 211 p.

Материалы из интернета

[AV] Модуль AstraVer для платформы Frama-C: https://forge.ispras.ru/projects/astraver-toolset.

[Chaff] SAT-решатель zChaff: https://www.princeton.edu/~chaff/zchaff.html.

[CPA] Инструмент верификации CPAchecker: https://cpachecker.sosy-lab.org.

[Cudd] Библиотека CUDD: http://vlsi.colorado.edu/~fabio/CUDD/.

[Dam] M. Dam. *Temporal Logic. Lecture 2*: https://www.csc.kth.se/~mfd/Courses/Temporal_logic/lecture2.pdf.

[FraC] Платформа статического анализа Frama-C: http://frama-c.com.

[Jess] Модуль Jessie для платформы Frama-C: http://frama-c.com/jessie.html.

[JPF] Инструмент верификации Java PathFinder: http://javapathfinder.sourceforge.net.

[Krak] Модули Krakatoa и Jessie: http://krakatoa.lri.fr.

[MatS] SMT-решатель MathSAT5: https://mathsat.fbk.eu.

[MinS] SAT-решатель MiniSat: http://minisat.se.

[NuSMV] Инструмент проверки моделей NuSMV: http://nusmv.fbk.eu.

[Prom] Документация по языку Promela: http://spinroot.com/spin/Man/promela.html.

[Rüm] P. Rümmer. *Craig Interpolation in SAT and SMT*. SAT/SMT Summer School, 2014: https://satsmt2014.forsyte.at/files/2014/01/interpolation_philipp.pdf.

[SMTi] Интерполирующий SMT-решатель SMTInterpol: https://ultimate.informatik.uni-freiburg.de/smtinterpol/.

[Spin] Инструмент проверки моделей Spin: http://spinroot.com.

[Why3] Платформа дедуктивной верификации Why3: http://why3.lri.fr.

Именной указатель

A

Ackermann, Wilhelm Friedrich, 121, 133
Akers, Sheldon B. Jr., 236

B

Biere, Armin, 230
Birkhoff, George David, 49
Bryant, Randal, 237
Büchi, Julius Richard, 200

C

Clarke, Edmund Melson Jr., 153
Cousot, Patrick, 109
Cousot, Radhia, 109
Craig, William, 189

D

Davis, Martin David, 127
Dijkstra, Edsger Wybe, 42

E

Ehrenfeucht, Andrzej, 121
Emerson, Ernest Allen II, 153

F

Floyd, Robert W, 79
Fourier, Jean-Baptiste Joseph, 113

G

Gentzen, Gerhard Karl Erich, 49
Godefroid, Patrice, 277
Gödel, Kurt Friedrich, 51
Gries, David, 106

H

Herbrand, Jacques, 120
Hilbert, David, 49
Hintikka, Kaarlo Jaakko Juhani, 214
Hoare, Charles Antony Richard, 39
Holzmann, Gerard J., 157

K

Klarlund, Nils, 277
Kripke, Saul Aaron, 149

L

Lee C. Y., 236
Lindenbaum, Adolf, 214
Lintao Zhang, 130
Logemann, George Wahl, 127
Loveland, Donald W., 127
Löwenheim, Leopold, 121

M

Martelli, Alberto, 132
McCarthy, John, 20, 121
McMillan, Kenneth, 225
Mealy, George H., 22
Minato, Shin-ichi, 239
Montanari, Ugo, 132
Moore, Edward Forrest, 23
Mostowski, Andrzej, 121
Motzkin, Theodore Samuel, 113
Muller, David Eugene, 200

N

Nelson, Charles Gregory, 133
Nicely, Thomas Ray, 18

O

Oppen, Derek Charles, 133

P

Peano, Giuseppe, 36
Peterson, Gary L., 151
Plotkin, Gordon David, 37
Pnueli, Amir, 146
Presburger, Mojżesz, 121
Putnam, Hilary Whitehall, 127

R

Rabin, Michael Oser, 200
Robinson, Abraham, 120
Robinson, George A, 120
Robinson, John Alan, 120
Robinson, Julia Hall, 120, 121
Robinson, Raphael Mitchel, 120, 122

Rosser, John Barkley, 121

S

Schwabhäuser, Wolfram, 121
Scott, Dana Stewart, 45
Sen, Koushik, 277
Shannon, Claude Elwood, 236
Sifakis, Joseph (Σηφάκης, Ιωσήφ), 153
Skolem, Thoralf Albert, 130
Stålmarck, Gunnar, 130
Strachey, Christopher S., 45

T

Tarjan, Robert Endre, 204
Tarski, Alfred, 49, 121

U

Ubar, Raimund, 236

V

Vardi, Moshe, 207

W

Weyl, Hermann Klaus Hugo, 49
Wolper, Pierre, 207

А

Акерс, Шелдон. См. *Akers, Sheldon B. Jr.*
Аккерман, Вильгельм Фридрих. См. *Ackermann, Wilhelm Friedrich*

Б

Бир, Армин. См. *Biere, Armin*
Биркгоф, Джордж Дэвид. См. *Birkhoff, George David*
Брайант, Рэндел. См. *Bryant, Randal*
Бюхи, Юлиус Рихард. См. *Büchi, Julius Richard*

В

Варди, Мойше. См. *Vardi, Moshe*
Вейль (Вайль), Герман Клаус Гуго. См. *Weyl, Hermann Klaus Hugo*
Вольпер, Пьер. См. *Wolper, Pierre*

Г

Гёдель, Курт Фридрих. См. *Gödel, Kurt Friedrich*
Генцен, Герхард Карл Эрих. См. *Gentzen, Gerhard Karl Erich*
Гильберт, Давид. См. *Hilbert, David*
Годефруа, Патрис. См. *Godefroid, Patrice*
Грис, Дэвид. См. *Gries, David*

Д

Дейкстра, Эдсгер Вибе. См. *Dijkstra, Edsger Wybe*
Дэвис, Мартин Дэвид. См. *Davis, Martin David*

К

Кларк, Эдмунд Мельсон. См. *Clarke, Edmund Melson Jr.*
Кларлунд, Нильс. См. *Klarlund, Nils*
Крейг, Вильям. См. *Craig, William*
Крипке, Сол Аарон. См. *Kripke, Saul Aaron*
Кюзо, Патрик. См. *Cousot, Patrick*
Кюзо, Радхия. См. *Cousot, Radhia*

Л

Лавленд, Дональд. См. *Loveland, Donald W.*
Лёвенгейм, Леопольд. См. *Löwenheim, Leopold*
Линденбаум, Адольф. См. *Lindenbaum, Adolf*
Линьтао Чжан. См. *Lintao Zhang*
Ли Ч. Ю.. См. *Lee C. Y.*
Логеман, Джорж Уолл. См. *Logemann, George Wahl*

М

Маккарти, Джон. См. *McCarthy, John*
Макмиллан, Кеннет. См. *McMillan, Kenneth*
Мартелли, Альберто. См. *Martelli, Alberto*
Маслов, Сергей Юрьевич, 131
Матиясевич, Юрий Владимирович, 127
Мили, Джордж. См. *Mealy, George H.*
Минато, Шиничи. См. *Minato, Shin-ichi*
Монтанари, Уго. См. *Montanari, Ugo*
Мостовский, Анджей. См. *Mostowski, Andrzej*
Моцкин, Самуэль Теодор. См. *Motzkin, Theodore Samuel*
Мур, Эдвард Форест. См. *Moore, Edward Forrest*
Мюллер, Дэвид Юджин. См. *Muller, David Eugene*

Н

Найсли, Томас Рей. См. *Nicely, Thomas Ray*
Нельсон, Чарльз Грегори (Грег). См. *Nelson, Charles Gregory*

Непейвода, Николай Николаевич, 35

О

Оппен, Дерек Чарльз. См. *Oppen, Derek Charles*

П

Патнэм, Хилари Уайтхолл. См. *Putnam, Hilary Whitehall*
Пеано, Джузеппе. См. *Peano, Giuseppe*
Петерсон, Гари. См. *Peterson, Gary L.*
Плоткин, Гордон Дэвид. См. *Plotkin, Gordon David*
Пнуэли, Амир. См. *Pnueli, Amir*
Пресбургер, Мойжеш. См. *Presburger, Mojżesz*

Р

Рабин, Михаэль Ошер. См. *Rabin, Michael Oser*
Робинсон, Абрахам (Авраам). См. *Robinson, Abraham*
Робинсон, Джон Алан. См. *Robinson, John Alan*
Робинсон, Джордж. См. *Robinson, George A.*
Робинсон, Джулия Холл. См. *Robinson, Julia Hall*
Робинсон, Рафаэль Митчел. См. *Robinson, Raphael Mitchel*
Россер, Джон Баркли. См. *Rosser, John Barkley*

С

Сен, Каушик. См. *Sen, Koushik*
Сифакис, Иосиф. См. *Sifakis, Joseph* (Σηφάκης, Ιωσήφ)
Скотт, Дана Стюарт. См. *Scott, Dana Stewart*
Скулем, Туральф Альберт. См. *Skolem, Thoralf Albert*
Стольмарк, Гуннар. См. *Stålmarck, Gunnar*
Стрейчи, Кристофер. См. *Strachey, Christopher S.*

Т

Тарский, Альфред. См. *Tarski, Alfred*
Тарьян (Тарджан), Роберт Эндре. См. *Tarjan, Robert Endre*
Трахтенброт, Борис Абрамович, 121

У

Убар, Раймунд. См. *Ubar, Raimund*

Ф

Флойд, Роберт. См. *Floyd, Robert W.*
Фурье, Жан-Батист Жозеф. См. *Fourier, Jean-Baptiste Joseph*

Х

Хинтикка, Каарло Яакко Юхани. См. *Hintikka, Kaarlo Jaakko Juhani*
Хоар, Чарльз Энтони Ричард (Тони). См. *Hoare, Charles Antony Richard*
Хольцман, Джерард. См. *Holzmann, Gerard J.*

Ц

Цейтин, Григорий Самуилович, 124

Ш

Швабхаузер, Вольфрам. См. *Schwabhäuser, Wolfram*
Шеннон, Клод Элвуд. См. *Shannon, Claude Elwood*
Шестаков, Виктор Иванович, 236
Шура-Бура, Михаил Романович, 17

Э

Эмерсон, Эрнест Аллен. См. *Emerson, Ernest Allen II*
Эрбран, Жак. См. *Herbrand, Jacques*
Эренфойхт, Анджей. См. *Ehrenfeucht, Andrzej*

Александр Сергеевич Камкин

Введение в формальные методы верификации программ

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 24,7. Тираж 100 экз.

Веб-сайт издательства: **www.dmkpress.com**