Автоматное программирование

Хлебников Андрей Александрович

7 августа 2018 г.

Оглавление

Введение **1**

Машина Тьюринга **4**

Регулярные выражения **6**

Лексические анализаторы **7**

Описание объекта управления **8**
Диаграма состояний 8
Таблица состояний 8
Блок-схема 8
Функциональная схема 8

Верификация программ методом `Model Checking` **9**
Системы переходов 10
Понятие системы переходов 10
Темпоральная логика 10
LTL 11
CTL 11
Язык Promela 11
Типы данных 12
Процессы 12
Атомарные конструкции 13
Каналы сообщений 13
Ветвления и кнструкции управления 14
Циклы 15
Безусловные переходы 15
Проверки 15
Составные типы данных 16
Исполняемость 16
Ключевые слова 17
SPIN 19

Автоматное программирование **20**

Приложение **21**
Парадигмы программирования 21

Императивное программирование 21
Декларативное программирование 22
Структурное программирование 23
Функциональное программирование 23
Логическое программирование 25
Объектно-ориентированное программирование 25
Процедурное программирование 26
Б.П. Кузнецов об автоматном программировании 27

Введение

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

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

Определяющими для автоматного программирования являются следующие особенности:

временной период выполнения программы разбивается на шаги автомата, каждый из которых представляет собой выполнение определённой (одной и той же для каждого шага) секции кода с единственной точкой входа; такая секция может быть оформлена, например, в виде отдельной функции и может быть разделена на подсекции, соответствующие отдельным состояниям или категориям состояний передача информации между шагами автомата осуществляется только через явно обозначенное множество переменных, называемых состоянием автомата; между шагами автомата программа (или её часть, оформленная в автоматном стиле) не может содержать неявных элементов состояния, таких как значения локальных переменных в стеке, адреса возврата из функций, значение текущего счётчика команд и т. п.; иначе говоря, состояние программы на любые два момента входа в шаг автомата могут различаться между собой только значениями переменных, составляющих состояние автомата (причём такие переменные должны быть явно обозначены в качестве таковых). Полностью выполнение кода в автоматном стиле представляет собой цикл (возможно, неявный) шагов автомата.

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

Сфера применения: Автоматное программирование широко применяется при построении лексических анализаторов (классические конечные автоматы) и синтаксических анализаторов (автоматы с магазинной памятью)[1].

[1] А. Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компиляции = The

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

Часто понятие состояний и машин состояний используется для спецификации программ. Так, при проектировании программного обеспечения с помощью `UML` для описания поведения объектов используются диаграммы состояний (state machine diagrams). Кроме того, явное выделение состояний используется в описании сетевых протоколов (см., например, RFC 793[2]).

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

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

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

В качестве одного из примеров применения автоматной парадигмы можно назвать систему `STAT`[3]; эта система, в частности, включает встроенный язык `STATL`, имеющий чисто автоматную семантику.

Существуют также предложения по использованию автоматного программирования в качестве универсального подхода к созданию компьютерных программ вне зависимости от предметной области. Так, авторы статьи[4] утверждают, что автоматное программирование способно сыграть роль легендарной серебряной пули

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

---

theory of parsing, translation and compiling. - М.: МИР, 1978. - Т. 1. - 612 с

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

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

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

которые будут освещены далле в этом курсе.

В свою очередь есть следующее определение автоматного программирования [1] – это некая общая парадигма программирования, суть которой заключается в том, что создаваемая программа рассматривается как реализация некоторого управляющего автомата.

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

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

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

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

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

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

Машина Тьюринга

Машина Тьюринга[5] – абстрактный исполнитель (абстрактная вычислительная машина). Была предложена Аланом Тьюрингом в 1936 году для формализации понятия алгоритма. Машина Тьюринга является расширением конечного автомата и, согласно тезису Черча – Тьюринга, способна имитировать всех исполнителей (с помощью задания правил перехода), каким-либо образом реализующих процесс пошагового вычисления, в котором каждый шаг вычисления достаточно элементарен. То есть, всякий интуитивный алгоритм может быть реализован с помощью некоторой машины Тьюринга.

В состав машины Тьюринга входит неограниченная в обе стороны лента (возможны машины Тьюринга, которые имеют несколько бесконечных лент), разделённая на ячейки, и управляющее устройство (также называется головкой записи-чтения (ГЗЧ)), способное находиться в одном из множества состояний. Число возможных состояний управляющего устройства конечно и точно задано.

Управляющее устройство может перемещаться влево и вправо по ленте, читать и записывать в ячейки символы некоторого конечного алфавита. Выделяется особый пустой символ, заполняющий все клетки ленты, кроме тех из них (конечного числа), на которых записаны входные данные.

Управляющее устройство работает согласно правилам перехода, которые представляют алгоритм, реализуемый данной машиной Тьюринга. Каждое правило перехода предписывает машине, в зависимости от текущего состояния и наблюдаемого в текущей клетке символа, записать в эту клетку новый символ, перейти в новое состояние и переместиться на одну клетку влево или вправо. Некоторые состояния машины Тьюринга могут быть помечены как терминальные, и переход в любое из них означает конец работы, остановку алгоритма.

Машина Тьюринга называется детерминированной, если каждой комбинации состояния и ленточного символа в таблице соответствует не более одного правила. Если существует пара «ленточный символ — состояние», для которой существует 2 и более команд, такая машина Тьюринга называется недетерминированной.

Конкретная машина Тьюринга задаётся перечислением элементов множества букв алфавита A, множества состояний Q и набором правил, по которым работает машина. Они имеют вид: $q_i a_j \to q_{i1} a_{j1} d_k$ (если головка находится в состоянии q_i, а в обозреваемой ячейке записана буква a_j, то головка переходит в состояние q_{i1}, в ячейку вместо a_j записывается a_{j1}, головка делает движение d_k, которое имеет три варианта: на ячейку влево (L), на ячейку вправо (R),

[5]Информация позаимствована с WiKi

остаться на месте (N)). Для каждой возможной конфигурации $< q_i, a_j >$ имеется ровно одно правило (для недетерминированной машины Тьюринга может быть большее количество правил). Правил нет только для заключительного состояния, попав в которое, машина останавливается. Кроме того, необходимо указать конечное и начальное состояния, начальную конфигурацию на ленте и расположение головки машины.

Регулярные выражения

Лексические анализаторы

Описание объекта управления

Диаграма состояний

Таблица состояний

Блок-схема

Функциональная схема

Верификация программ методом Model Checking

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

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

Системы переходов [4]

Понятие системы переходов

Системой переходов(СП) называется пятерка S вида

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

компоненты которой имеют следующий смысл.

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

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

Темпоральная логика

Одним из языков, на котором можно специфицировать свойства систем, является темпоральная логика [4]. Свойства систем описываются в темпоральной логике при помощи темпоральных формул (которые мы будем называть также просто формулами). Примеры свойств, которые могут описываться в темпоральной логике:

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

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

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

Наиболее известны темпоральные логики

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

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

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

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

тоже принадлежат логике Ф.

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

LTL

CTL

Язык Promela

`PROMELA (Process or Protocol Meta Language)` – это язык описания описания моделей верификации, созданный `Gerard J. Holzmann` [5]. Язык поддерживает создание процессов для проверки распределенных моделей. Модели в языке могут взаимодействовать между собой при помощи каналов сообщений как в синхронном 2режиме, так и в асинхронном. Модели описанные при помощи языка могут быть обработаны и проанализированы SPIN о чем будет рассказано в последующих главах. Существуют иные реализации и утилиты использующие язык `Promela`, но пока они рассматриваться не будут.

В основном, язык предназначен для проверки логики работы парраллельных систем. Модели описанные `Promela` и обработанные утилитой `SPAN` проверяют модель на корректность в режиме случайной или последовательной симуляции или генерируют код на `C` для быстрой и полной проверки в системном окружении. В процессе симуляции и проверки `SPIN` проверяет отсутствия `deadlocks`[6],

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

неопределенных состояний и неиспользуемых частей кода. Также данный подход может проверять правильность системных инвариантов[7], а также поиска зацикливаний и неправильных ветвлений. Также он поддерживает проверку LTL ограничений.

Список спецификаций различных систем, модель которых описана на языке Promela приведена в статье Alberto Lluch[8]

Типы данных

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

Типы bit и bool это синонимы.

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

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

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

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

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

Процессы

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

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

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

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

[7] https://en.wikipedia.org/wiki/Invariant_(computer_science)

[8] http://www.albertolluch.com/research/promelamodels

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

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

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

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

Атомарные конструкции

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

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

Каналы сообщений

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

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

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

```
qname ! expression;
```

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

```
qname ? msg;
```

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

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

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

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

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

Ветвления и конструкции управления

Простейшее сравнение двух переменных:

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

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

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

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

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

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

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

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

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

Циклы

Для повторения группы выражений применяются циклы. Пример

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

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

Безусловные переходы

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

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

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

Проверки

Выжной частью модели описанной языком `Promela` является утверждение

```
assert(any_boolean_condition)
```

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

Составные типы данных

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

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

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

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

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

Исполоняемость

Исполняемость модели обеспечивает базовые средства языка для моделирования синхронизации процессов.

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

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

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

В примере два процесса `P1` и `P2` имеют недетерминированный выбор 1 входа во 2 выход. Возможны два варианта выбора из которых только один будет выбран. Повторение будет бесконечным. При этом модель не получит `deadlock`[9]

Когда SPIN анализирует модель он проверяет ее при помощи недетерминированного алгоритма и проверит все возможные ее состояния. Когда симулятор SPIN будет визуализировать возможные не проверенные связи, он будет использовать генератор случайных чисел, для проверки недетерминированный состояний. Следовательно симулятор может не показать плохие пути выполнения (хотя таких путей в примере нет). Это иллюстрирует разницу между проверкой и симуляцией. Также можно генерировать исполняемый код из моделей `Promela` с использованием `Refinement`[10]

Ключевые слова

Список ключевых слов используемых в языке

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

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

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

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

Полное описание языка в форме Бэкуса-Наура представленя в приложении ??

SPIN

SPIN (англ. Simple Promela Interpreter)[11] - утилита для верификации корректности распределенных программных моделей. Служит для автоматизированной проверки моделей. Развивается Gerard J. Holzmann и его коллегами из Unix group центра Computing Sciences Research Center в Bell Labs начиная с 1980 года. С 1991 года программа распространяется бесплатно вместе с исходными кодами.

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

Описание опций можно посмотреть в приложении **??**

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

Автоматное программирование

Приложение

Парадигмы программирования

Императивное программирование

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

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

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

При императивном подходе к составлению кода (в отличие от функционального подхода, относящегося к декларативной парадигме) широко используется присваивание. Наличие операторов присваивания увеличивает сложность модели вычислений и делает императивные программы подверженными специфическим ошибкам, не встречающимся при функциональном подходе[12].

Основные черты императивных языков:

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

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

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

Декларативное программирование

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

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

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

На повышение уровня декларативности нацелено языково-ориентированное программирование.

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

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

Структурное программирование

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

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

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

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

По мнению Бертрана Мейера, «Революция во взглядах на программирование, начатая Дейкстрой, привела к движению, известному как структурное программирование, которое предложило систематический, рациональный подход к конструированию программ. Структурное программирование стало основой всего, что сделано в методологии программирования, включая и объектное программирование»[13]

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

Функциональное программирование

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

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

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

состояний вычислительного процесса представляется явным образом, например, как список.

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

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

Лямбда-исчисление являются основой для функционального программирования, многие функциональные языки можно рассматривать как «надстройку» над ними[14].

Пример(Erlang):

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

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

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

Логическое программирование

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

Самым известным языком логического программирования является `Prolog`.

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) – методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования[15].

Идеологически ООП - подход к программированию как к моделированию информационных объектов, решающий на новом уровне основную задачу структурного программирования: структурирование информации с точки зрения управляемости[16], что существенно улучшает управляемость самим процессом моделирования, что в свою очередь особенно важно при реализации крупных проектов.

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

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

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

---

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

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

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

Обычный человеческий язык в целом отражает идеологию ООП, начиная с инкапсуляции представления о предмете в виде его имени и заканчивая полиморфизмом использования слова в переносном смысле, что в итоге развивает[17] выражение представления через имя предмета до полноценного понятия-класса.

Процедурное программирование

Процедурное программирование – программирование на императивном языке, при котором последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка[18].

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

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

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

Б.П. Кузнецов об автоматном программировании

Б. П. Кузнецов[19]

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

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

1. Не учтенные состояния автомата, вызванные незнанием предметной области;
2. Дублирование (избыточность) состояний, приводящее к непредсказуемому поведению программы;
3. Не учтенные переходы;
4. Лишние переходы;
5. Неверно ориентированные переходы;
6. Неортогональность входного алфавита;
7. Неверное назначение приоритетов переходов при неортогональном алфавите;
8. Не учтенные входные воздействия и неполнота входного алфавита;
9. Не полный учет букв входного алфавита;
10. Путаница, связанная с неверным отождествлением входных воздействий и буквами входного алфавита (наиболее распространенная ошибка);
11. Неверные булевы формулы, отождествляемые с буквами входного алфавита и помечающие переходы (см. п.6);
12. Не учтенные выходные воздействия, в особенности как реакция на ошибочное поведение управляемого объекта и самого управляющего автомата и связи управляющего и операционного автоматов с объектом управления;
13. Неверная пометка состояний и переходов буквами выходного алфавита;
14. Неверное отождествление букв выходного алфавита с выходными воздействиями;
15. Забытое обнуление или продление выходного воздействия;

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

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

17. Не прослеживаются полные пути в диаграмме состояний;

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

19. Другие ошибки.

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

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

Литература

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

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

[3] А. А. Шалыто and С. Э. Вельдер. О ВЕРИФИКАЦИИ ПРОСТЫХ АВТОМАТНЫХ ПРОГРАММ НА ОСНОВЕ МЕТОДА model checking. УДК 681.3.06.

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

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

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

[7] А. А. Шалыто. АВТОМАТНОЕ ПРОГРАММИРОВАНИЕ, 2006. При поддержке Министерства образования и науки Российской Федерации в рамках научно-исследовательской работы по теме «Разработка технологии создания программного обеспечения систем управления на основе автоматного подхода».

Список иллюстраций