Верификация программ
Часть 1: нерекурсивные программы.

А.М. Миронов

Оглавление

Аннотация

В книге излагаются вопросы доказательства правильности нерекурсивных последовательных и параллельных программ. Основные концепции и основанные на них подходы к верификации иллюстрированы многочисленными примерами верификации программ. Для закрепления усвоения изложенного материала в книге приведено большое количество задач.

Введение

Проблемы верификации (т.е. доказательства правильности) программ занимают центральное положение в теории и практике разработки программного обеспечения. Под правильностью программ понимается их соответствие различным условиям корректности, безопасности, устойчивости в случае непредусмотренного поведения окружения, эффективности использования ресурсов времени и памяти, оптимальности реализованных в программе алгоритмов, и т.п.

Как правило, для обоснования правильности программы её тестируют, т.е анализируют её поведение на некоторых входных данных. Однако тестирование обладает очевидным недостатком: если его возможно провести не для всех допустимых входных данных, а только лишь для их небольшой части (что имеет место почти всегда), то оно не может служить гарантированным обоснованием того, что тестируемая программа обладает проверяемыми свойствами. Как отметил Дейкстра ([1], стр. 41), тестирование может лишь помочь выявить некоторые ошибки, но отнюдь не доказать их отсутствие. Ошибки в программах могут быть весьма тонкими, и чем тоньше ошибка, тем сложнее обнаружить её тестированием. Но во многих программах наличие даже незначительных ошибок категорически недопустимо. Например, наличие даже небольших ошибок в таких программах, как

- программы управления атомными электростанциями,
- программы, управляющие работой медицинских устройств,
- программы в бортовых системах управления самолетов и космических аппаратов,
- программы в системах управления секретными базами данных, системах электронной коммерции, и т.п.

может привести к существенному ущербу для экономики и жизни людей.

Приведем один пример, иллюстрирующий наличие ошибок даже в очень простых программах, правильность которых на первый взгляд не

вызывает никакого сомнения. Рассмотрим программу P, задача которой заключается в зачислении денег на счет клиента банка. Количество денег на счету этого клиента хранится в базе данных банка в переменной x. Когда P выполняет действия по зачислению суммы s на этот счет, она выполняет следующие действия:

- копирует в свою внутреннюю память значение переменной x
- вычисляет новое значение, которое должна иметь переменная x, оно равно сумме текущего значения x и зачисляемой суммы s, и
- заносит в переменную x это новое значение.

Даже если программа P выполняет все свои действия правильно, это не гарантирует корректности обслуживания клиента в том случае, когда состояние его счета может изменяться несколькими такими программами. Рассмотрим ситуацию, когда состояние счета клиента изменяют две программы P_1 и P_2 описанного выше типа. Возможен следующий вариант совместного выполнения этих программ:

- сначала программа P_1 выполняет свое первое действие, оно начинается в момент времени t_1, а заключительное действие P_1 (обновление значения x) происходит в момент времени t_2, и
- в момент времени, лежащий в интервале между t_1 и t_2, программа P_2 начинает свою операцию зачисления денег на счет клиента, причем выполнение первого действия программы P_2 (копирование значения переменной x) производится до выполнения заключительного действия программы P_1.

После завершения работы обоих программ те деньги, которые зачислила на счет клиента одна из программ P_1, P_2, просто пропадут.

Ошибку описанного выше типа можно не обнаружить путем тестирования, т.к. операция зачисления денег на счет клиента выполняется практически мгновенно, и поэтому среди тестов, которыми можно анализировать программы подобного типа, с большой вероятностью могут отсутствовать такие тесты, в которых две различные программы, обслуживающие счета клиентов, почти одновременно обращаются к одному и тому же ресурсу памяти.

Если же в результате какого-либо тестирования указанная выше ошибка обнаруживается, и для её исправления конструируются специальные

программные механизмы (семафоры, локи, и т.п.) с целью задания правильной дисциплины обращения программ к одному и тому же ресурсу памяти, то встает вопрос о том, насколько эти механизмы соответствуют своему предназначению (в частности, защищены ли семафоры от непредусмотренного и неавторизованного изменения их значений). Это тоже может анализироваться путем тестирования, и опять может получиться так, что среди тестов, которыми анализируется поведение указанных выше специальных программных механизмов, с большой вероятностью будут отсутствовать такие тесты, в которых проявляется некорректное поведение этих механизмов.

Гарантированное обоснование правильности программ может быть получено только при помощи альтернативного подхода, принципиально отличного от тестирования. Данный подход называется **верификацией**. В самом общем виде верификация программы может пониматься как построение математического доказательства утверждения о том, что верифицируемая программа соответствует своему предназначению. Предназначение программы может быть выражено, например, путем описания функции, которую должна вычислять эта программа, или правил взаимодействия этой программы с другими программами, т.е. реакции, которую эта программа должна обеспечивать в ответ на получение сигналов или сообщений от других программ.

Формальное описание предназначения программы (или некоторых свойств, которыми она должна обладать) в виде математического утверждения называется **спецификацией** этой программы. Спецификация может представлять собой формальное описание самых разнообразных свойств программы, например:

- её входные и выходные данные находятся в заданном соотношении,
- программа всегда завершает свою работу,
- во время работы программы не происходит сбоев и ненормальных ситуаций (например, деления на 0, извлечения квадратного корня из отрицательного числа, выхода индекса за границы массива, неавторизованных утечек информации, и т.п.),
- программа решает свою задачу за установленное время,
- программа использует не более установленного объема памяти.

Для верификации программы P необходимо определить

- математический смысл всех конструкций, используемых в P, называемый **формальной семантикой** (или просто **семантикой**) этих конструкций, и

- спецификацию *Spec* этой программы, выражающую то свойство программы P, которое необходимо верифицировать,

после чего можно ставить вопрос о верификации P относительно *Spec*, т.е. о построении математического доказательства утверждения о том, что P удовлетворяет *Spec*.

В настоящем тексте излагается один из наиболее широко распространённых методов верификации программ, известный под названием **метод инвариантов**. Одними из первых работ, в которых изложен этот метод, являются статьи [3] и [4]. Метод инвариантов основан на понятиях математической логики, с которыми можно познакомиться по книге [5]. Различные изложения метода инвариантов содержатся в [6]–[11].

Глава 1

Программы, представленные в виде блок-схем

Одним из языков описания программ является язык блок-схем. На данном языке программа представляется в графовой форме. Отметим, что графовая форма представления программ в последнее время завоевывает все большую популярность, по причине того, что такая форма представления программ облегчает их понимание и упрощает их анализ.

1.1 Вспомогательные понятия

Мы будем предполагать, что заданы следующие множества.

- Множество $\mathcal{T}$, элементы которого называются **типами**. Каждому типу $t \in \mathcal{T}$ сопоставлено множество D_t **значений** типа t.
- Множество $\mathcal{X}$, элементы которого называются **переменными**. Каждой переменной $x \in \mathcal{X}$ сопоставлен тип $t(x) \in \mathcal{T}$. Каждая переменная $x \in \mathcal{X}$ может принимать **значения** в множестве $D_{t(x)}$, т.е. в различные моменты времени переменная x может быть связана с различными элементами множества $D_{t(x)}$.
- Множество $\mathcal{C}$, элементы которого называются **константами**. Каждой константе $c \in \mathcal{C}$ сопоставлены тип $t(c) \in \mathcal{T}$ и значение из $D_{t(c)}$, обозначаемое тем же символом c, и называемое **интерпретацией** константы c.
- Множество $\mathcal{F}$, элементы которого называются **функциональными символами (ФС)**. Каждому ФС $f \in \mathcal{F}$ сопоставлены

– **функциональный тип** $t(f)$, который представляет собой запись вида $(t_1, \ldots, t_n) \to t$, где $t_1, \ldots, t_n, t \in \mathcal{T}$, и

– функция вида $D_{t_1} \times \ldots \times D_{t_n} \to D_t$, где $(t_1, \ldots, t_n) \to t \ = t(f)$, данная функция обозначается тем же символом f и называется **интерпретацией** ФС f.

Ниже мы будем использовать ФС $+, -, \cdot, div, mod$, где

– ФС $+, -, \cdot$ имеют функциональный тип $(\texttt{int}, \texttt{int}) \to \texttt{int}$, и $\texttt{int}$ – тип, значениями которого являются целые числа,

– ФС div, mod имеют функциональный тип $(\texttt{int}, \texttt{int}_0) \to \texttt{int}$, и $\texttt{int}_0$ – тип, значениями которого являются целые числа, отличные от 0,

и функции $+$, $-$, и $\cdot$ представляют собой соответствующие арифметические операции, div и mod вычисляют частное и остаток соответственно от деления первого аргумента на второй.

Выражения строятся из переменных, констант и ФС. Множество всех выражений обозначается символом $\mathcal{E}$. Каждое выражение e имеет тип $t(e) \in \mathcal{T}$, определяемый структурой выражения e. Правила построения выражений имеют следующий вид:

- каждая переменная и константа является выражением того типа, который сопоставлен этой переменной или константе, и

- если $e_1, \ldots, e_n \in \mathcal{E}$, $f \in \mathcal{F}$, и $t(f)$ имеет вид $(t(e_1), \ldots, t(e_n)) \to t$, то знакосочетание $f(e_1, \ldots, e_n)$ является выражением типа t.

Выражения $+(e_1, e_2)$, $-(e_1, e_2)$, $\cdot(e_1, e_2)$, $div(e_1, e_2)$, $mod(e_1, e_2)$ будут записываться в более привычном виде $e_1 + e_2$, $e_1 - e_2$, $e_1 e_2$, e_1 / e_2 и $e_1 \% e_2$ соответственно.

Среди типов, входящих в $\mathcal{T}$, имеется тип $\texttt{bool}$, множество значений которого имеет вид $\{0, 1\}$. Выражения типа $\texttt{bool}$ называются **формулами**. Множество всех формул обозначается символом $\mathcal{B}$. При построении формул могут использоваться обычные булевские ФС ($\wedge, \vee, \to$ и т.д.), которым соответствуют функции коньюнкции, дизъюнкции, и т.д. Символ 1 обозначает тождественно истинную формулу, а символ 0 – тождественно ложную формулу. Формулы вида $\wedge(e_1, e_2)$, $\vee(e_1, e_2)$, и т.п. мы будем записывать в более привычном виде $e_1 \wedge e_2$, $e_1 \vee e_2$, и т.д. В некоторых случаях формулы вида $e_1 \wedge \ldots \wedge e_n$ будут записываться в виде $\left[\!\left[\begin{array}{c} e_1 \\ \ldots \\ e_n \end{array} \right]\!\right]$.

Среди ФС, входящих в $\mathcal{F}$, присутствуют ФС, обозначаемые одной и той же записью if_then_else, функциональный тип этих ФС имеет вид $(\texttt{bool}, t, t) \to t$, где t – произвольный тип из $\mathcal{T}$. Функция, соответствующая каждому такому ФС, отображает тройку вида $(1, d_1, d_2)$ в d_1, и тройку вида $(0, d_1, d_2)$ в d_2. Выражения вида $if\_then\_else(e, e_1, e_2)$ будут записываться более коротко в виде $e?e_1 : e_2$.

$\forall\, e \in \mathcal{E}$ запись X_e обозначает множество переменных, входящих в e.

$\forall\, X \subseteq \mathcal{X}$ записи $\mathcal{E}(X)$ и $\mathcal{B}(X)$ обозначают множества $\{e \in \mathcal{E} \mid X_e \subseteq X\}$ и $\mathcal{E}(X) \cap \mathcal{B}$ соответственно.

Пусть $X \subseteq \mathcal{X}$. **Означиванием** переменных из множества X называется произвольное соответствие ξ, которое сопоставляет каждой переменной $x \in X$ некоторое значение $x^\xi \in D_{t(x)}$. Множество всех означиваний переменных из X обозначается записью $X^\bullet$.

Если $X \subseteq \mathcal{X}$, $e \in \mathcal{E}(X)$, $\xi \in X^\bullet$, то e^ξ обозначает значение из $D_{t(e)}$, называемое **значением** e **на** ξ, и определяемое следующим образом:

- если $e \in \mathcal{X}$, то e^ξ предполагается заданным,
- если $e \in \mathcal{C}$, то e^ξ является интерпретацией константы e, и
- если $e = f(e_1, \ldots, e_n)$, то $e^\xi \stackrel{\text{def}}{=} f(e_1^\xi, \ldots, e_n^\xi)$.

Выражения $e_1, e_2 \in \mathcal{E}$ считаются равными, если

$$\forall\, \xi \in (X_{e_1} \cup X_{e_2})^\bullet \;\; e_1^\xi = e_2^\xi.$$

Пусть $X \subseteq \mathcal{X}$, $\xi \in X^\bullet$, $x \in X$ $e \in \mathcal{E}(X)$, $t(x) = t(e)$. Запись $\xi(x := e)$ обозначает означивание из $X^\bullet$, определяемое следующим образом:

$$x^{\xi(x:=e)} \stackrel{\text{def}}{=} e^\xi, \;\; \forall\, x \in X \setminus \{x\} \;\; x^{\xi(x:=e)} \stackrel{\text{def}}{=} x^\xi. \tag{1.1}$$

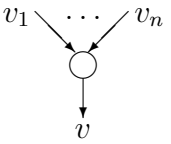
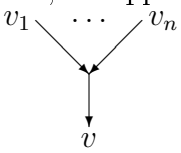
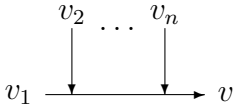
Если $x \in \mathcal{X}$ $e, e' \in \mathcal{E}$, и $t(x) = t(e)$, то запись $(x := e)e'$ обозначает выражение, получаемое из e' заменой всех вхождений x в e' на e.

Для любого списка выражений $e_1, \ldots, e_n$ запись $(e_1, \ldots, e_n)$ обозначает выражение, значение которого на каждом означивании $\xi \in X^\bullet$, где $X \supseteq \bigcup\limits_{i=1}^{n} X_{e_i}$, равно кортежу $(e_1^\xi, \ldots, e_n^\xi)$. Будем обозначать тип выражения $(e_1, \ldots, e_n)$ записью $(t(e_1), \ldots, t(e_n))$.

1.2 Понятие блок-схемы

Блок-схема (БС) представляет собой конечный ориентированный граф, каждая вершина которого имеет один из следующих типов:

- **начальная** вершина, она обозначается символом $\odot$, в каждой БС имеется только одна начальная вершина, из неё выходит одно ребро, и в неё не входит ни одного ребра,
- **присваивание**, вершина такого типа обозначается записью вида $\boxed{x := e}$, где $x \in \mathcal{X}$, $e \in \mathcal{E}$, $t(x) = t(e)$, из каждого присваивания выходит только одно ребро,
- **условный переход**, вершина такого типа обозначается записью вида $\textcircled{b}$, где $b \in \mathcal{B}$, из каждого условного перехода выходит два ребра, одно из которых имеет метку 1, а другое - метку 0,
- **пустой оператор**, такая вершина обозначается символом $\bigcirc$, из неё выходит только одно ребро,
- **терминальная** вершина, она обозначается символом $\otimes$, из каждой терминальной вершины не выходит ни одного ребра.

Пустой оператор в БС как правило не изображают, т.е. фрагмент БС вида  может изображаться в виде  , или  , или другим аналогичным образом.

В БС могут использоваться следующие обозначения:

- последовательность вершин вида $\boxed{x_1 := e_1} \to \cdots \to \boxed{x_n := e_n}$, где в каждую вершину (кроме, возможно, первой) входит только одно ребро, может обозначаться прямоугольником $\boxed{\begin{array}{c} x_1 := e_1 \\ \cdots \\ x_n := e_n \end{array}}$,
- запись

$$\boxed{(x_1, \ldots, x_n) := (e_1, \ldots, e_n)} \tag{1.2}$$

где $x_1, \ldots, x_n \in \mathcal{X}$, $e_1, \ldots, e_n \in \mathcal{E}$, $\forall\, i = 1, \ldots, n \;\; t(x_i) = t(e_i)$, означает последовательность присваиваний вида

$$\boxed{y_1 := e_1} \to \cdots \to \boxed{y_n := e_n} \to \boxed{x_1 := y_1} \to \cdots \to \boxed{x_n := y_n} \tag{1.3}$$

где переменные $y_1, \ldots, y_n$ – новые, т.е. если какая-либо БС содержит фрагмент вида (1.2), то он рассматривается как последовательность (1.3), где переменные $y_1, \ldots, y_n$ присутствуют только в присваиваниях, входящих в (1.3), и не присутствуют в этой БС вне (1.3),

- запись вида $\boxed{x \rightleftharpoons y}$ где $x, y \in \mathcal{X},\ t(x) = t(y)$, имеет тот же смысл, что и запись $\boxed{(x, y) := (y, x)}$.

Пусть P – БС. Будем обозначать записями V_P и X_P совокупность всех вершин P и переменных, входящих в P, соответственно.

Выполнение БС P – это обход вершин P, начиная с начальной вершины (т.е. последовательность переходов по рёбрам от одной вершины P к другой), с выполнением действий, сопоставленных проходимым вершинам. С каждым шагом $i \geq 0$ выполнения P связаны вершина $v_i \in V_P$ и означивание $\xi_i \in X_P^\bullet$ (называемые **текущей вершиной** и **текущим означиванием** на шаге i). Если v_i – нетерминальная вершина, то определен $i+1$-й шаг выполнения P, в P есть ребро из v_i в v_{i+1}, и

- если v_i – начальная вершина или пустой оператор, то $\xi_{i+1} = \xi_i$,
- если $v_i = \boxed{x := e}$, то $\xi_{i+1} \stackrel{\text{def}}{=} \xi_i(x := e)$,
  (можно интерпретировать действие, соответствующее этой вершине, как обновление значения переменной x: после исполнения этого действия значение x становится равным значению выражения e на текущих значениях переменных БС P)
- если $v_i = \textcircled{b}$, то $\xi_{i+1} = \xi_i$, и ребро из v_i в v_{i+1} имеет метку b^{ξ_i},

а если v_i – терминальная вершина, то выполнение БС завершается.

Отметим, что выполнение последовательности действий, соответствующих фрагменту $\boxed{x \rightleftharpoons y}$, можно рассматривать как перемену местами содержимого переменных x и y.

1.3 Массивы и матрицы

При записи программ могут использоваться **массивы**, обозначаемые записями вида $a_{m..n}$, где a – имя массива, m и n – выражения типа `int`, обозначающие нижнюю и верхнюю границы массива соответственно. Массив $a_{m..n}$ может обозначаться более коротко путем указания лишь его

имени, без указания нижней и верхней границ. Если значения m, n нижней и верхней границ массива a таковы, что $m \leq n$, то массив a непуст, в противном случае он является пустым. Будем обозначать символом $\mathbf{N}$ множество натуральных чисел (0, 1, ...), и $\forall\, m, n \in \mathbf{N}$ будем обозначать записью $\overline{m,n}$ множество $\{i \in \mathbf{N} \mid m \leq i \leq n\}$. Для каждого массива $a_{m..n}$ и каждого $i \in \overline{m,n}$ определен объект, обозначаемый записью a_i, и называемый **компонентой** массива a с индексом i. Компоненты массива a рассматриваются как переменные одного и того же типа. Для каждого массива a его тип $t(a) \in \mathcal{T}$ определяется как тип его компонентов.

Если a и b – массивы одинакового типа и с одинаковыми нижними и верхними индексами (т.е. данные массивы имеют вид $a_{m..n}$ и $b_{m..n}$), то

- запись $a := b$ обозначает операцию копирования компонентов массива b в соответствующие компоненты массива a, т.е. последовательность присваиваний $a_m := b_m$, $a_{m+1} := b_{m+1}$, ..., $a_n := b_n$,
- запись $b = perm(a)$ означает, что b получается перестановкой a, т.е. существует биекция f на $\overline{m,n}$, такая, что $\forall\, i \in \overline{m,n}\ \ b_i = a_{f(i)}$.

Пусть задан массив $a_{m..n}$, $\mathcal{F}$ имеет ФС $\leq$ типа $(t(a), t(a)) \to \texttt{bool}$, и $i, j, i', j' \in \overline{m,n}$. Будем использовать следующие обозначения:

- запись $ord(a_{i..j})$ обозначает формулу $\bigwedge\limits_{i \leq k < j} (a_k \leq a_{k+1})$,
- запись $a_{i..j} \leq a_{i'..j'}$ обозначает формулу $\bigwedge\limits_{\substack{i \,\leq\, k \,\leq\, j \\ i' \,\leq\, k' \,\leq\, j'}} (a_k \leq a_{k'})$,
- запись $a_{i..j} \leq a_{i'}$ обозначает формулу $\bigwedge\limits_{i \leq k \leq j} (a_k \leq a_{i'})$,

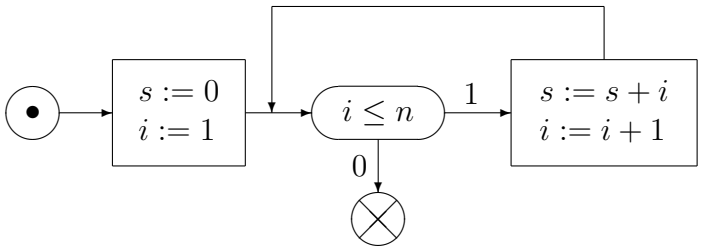
где запись вида $e_1 \leq e_2$ обозначает формулу $\leq (e_1, e_2)$.

При записи программ также могут использоваться **матрицы**, которые обозначаются записями вида $a_{m..n,m'..n'}$, где a – имя матрицы, m, n, m', n' – выражения типа `int`, обозначающие нижнюю и верхнюю границы по строкам и по столбцам матрицы a, соответственно. Матрицы иногда называют двумерными массивами. Матрица $a_{m..n,m'..n'}$ может обозначаться более коротко путем указания лишь её имени, без указания нижней и верхней границ. Для каждой матрицы $a_{m..n,m'..n'}$ и каждых $i \in \overline{m,n}, j \in \overline{m',n'}$ определен объект, обозначаемый записью a_{ij}, и называемый **компонентой** матрицы a с индексами i и j. Компоненты матрицы a рассматриваются как переменные одного и того же типа. Для каждой матрицы a её тип $t(a) \in \mathcal{T}$ определяется как тип ее компонентов.

Как и для массивов, для матриц определяется операция $a := b$ копирования компонентов матрицы b в соответствующие компоненты матрицы a.

1.4 Примеры блок-схем

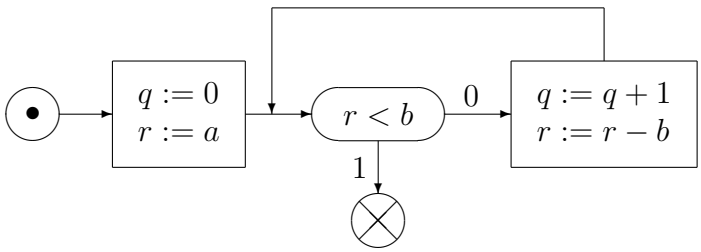
1. БС, вычисляющая сумму $1 + \ldots + n$, где $n \geq 1$ – входное значение, результат должен быть занесён в переменную s.



(1.4)

   Все переменные в данной БС имеют тип `int`.

2. БС, вычисляющая частное и остаток от целочисленного деления a на b, где a на b – положительные целые числа. В результате выполнения данной программы в переменные q и r должны быть занесены частное и остаток соответственно от деления a на b, т.е. должны быть выполнены соотношения $a = bq + r$ и $0 \leq r < b$.



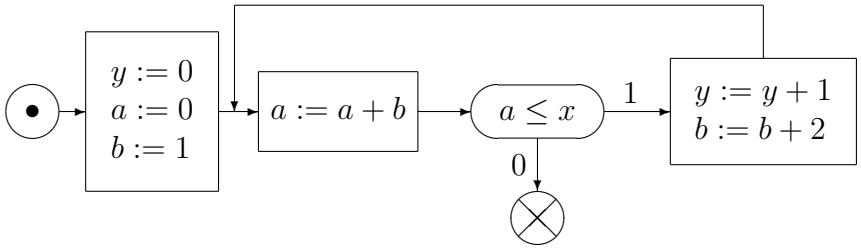
(1.5)

   Все переменные в данной БС имеют тип `int`.

   Алгоритм, реализованный в данной БС, заключается в вычитании b из a до тех пор, пока результат не станет меньше b. Число этих вычитаний равно частному, а результат вычитаний – остатку от целочисленного деления a на b.

3. БС, вычисляющая целую часть числа $\sqrt{x}$, где x – неотрицательное

целое число. Результат должен быть занесён в переменную y.

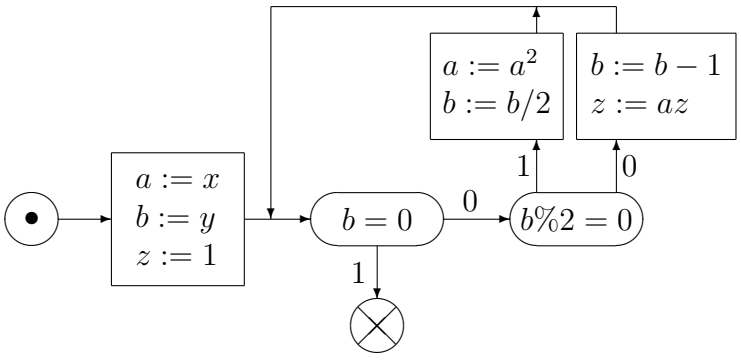


(1.6)

Все переменные в данной БС имеют тип `int`.

Алгоритм, реализованный в данной БС, заключается в использовании тождества $1+3+5+\ldots+(2k-1) = k^2$. Переменная b принимает последовательно значения $1, 3, 5, \ldots$, эти значения суммируются и результат заносится в переменную a, а число слагаемых в этой сумме заносится в переменную y. Когда значение a станет превосходить x, из приведенного выше тождества следует, что y содержит требуемое значение $\lfloor \sqrt{x} \rfloor$.

4. БС, реализующая быстрый алгоритм возведения в степень. В результате работы данной БС вычисляется значение x^y, где $y \geq 0$. Результат должен быть занесен в переменную z.



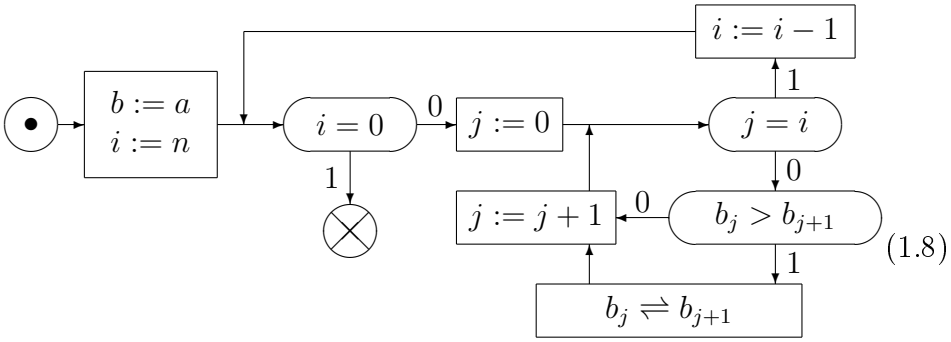
(1.7)

Все переменные в данной БС имеют тип `int`. Исходное значение y предполагается неотрицательным. Вычисление x^y производится по следующему принципу:

- если $y = 0$, то $x^y = 1$,
- если y – положительное четное, то $x^y = (x^2)^{y/2}$, и
- если y – нечетное, то $x^y = x^{y-1}x$.

Данный принцип позволяет понизить число умножений при вычислении x^y с $y-1$ до $O(\log_2 y)$.

5. БС, реализующая сортировку массива методом пузырька. Сортируемый массив имеет вид $a_{0..n}$, где $n \geq 0$. Результат должен быть записан в массив $b_{0..n}$. Требуется, чтобы после завершения выполнения этой БС были верны утверждения $b = perm(a)$ и $ord(b)$



(1.8)

Алгоритм, реализованный в этой БС, заключается в выполнении n прогонок по массиву b, каждая из которых заключается в просмотре слева направо участка от 0-й до i-й позиции сортируемого массива, и перемене местами его соседних элементов, если они нарушают порядок. Сначала производится прогонка по всему массиву, в результате этой прогонки в последнюю позицию массива помещается его максимальный элемент, в результате следующей прогонки в предпоследнюю позицию помещается следующий по величине элемент, и т.д.

1.5 Задача верификации блок-схем

Пусть задана БС P, и её спецификация представляет собой пару формул $(Pre, Post)$ с переменными из X_P, имеющих следующий смысл:

- формула Pre называется **предусловием**, и выражает условие, которому должны удовлетворять значения переменных БС P в момент начала её выполнения, и
- формула $Post$ называется **постусловием**, и выражает условие, которому должны удовлетворять значения переменных БС P после завершения её выполнения.

Пусть P – БС, и $\xi \in X_P^\bullet$. Мы будем говорить, что P **выполняется с начальным означиванием** ξ, если в момент начала выполнения P значение каждой переменной $x \in X_P$ было равно x^ξ.

Запись $\xi \xrightarrow{P} \otimes$ обозначает утверждение: если P выполняется с начальным означиванием ξ, то выполнение P когда-либо завершится.

Если верно $\xi \xrightarrow{P} \otimes$, то запись ξP обозначает означивание из $X_P^\bullet$, определяемое следующим образом: пусть P выполняется с начальным означиванием ξ, тогда $\forall\, x \in X_P$ значение $x^{\xi P}$ равно тому значению, которое будет иметь переменная x после завершения выполнения P.

Задача **верификации** БС P относительно спецификации $(Pre, Post)$ заключается в доказательстве следующего утверждения:

$$\forall\, \xi \in X_P^\bullet \quad \text{если} \;\; Pre^\xi = 1, \;\; \text{то} \;\; \xi \xrightarrow{P} \otimes \;\; \text{и} \;\; Post^{\xi P} = 1. \tag{1.9}$$

Данное утверждение обозначается записью $Pre \xrightarrow{P} Post$.

Глава 2

Метод инвариантов для верификации блок-схем

В этом параграфе излагается метод доказательства утверждений вида $Pre \overset{P}{\to} Post$, называемый **методом инвариантов**. Для его формулировки введём понятия базового множества и базового пути.

2.1 Базовые множества и базовые пути

Пусть задана БС P. **Путь** в P – это последовательность $\pi = \alpha_1 \dots \alpha_k$ рёбер P, такая, что $\forall\, i = 1, \dots, k-1$ конец ребра α_i совпадает с началом ребра α_{i+1}. Путь $\pi = \alpha_1 \dots \alpha_k$ называется **циклом** в P, если $\alpha_1 = \alpha_k$.

Множество M точек на некоторых ребрах БС P называется **базовым** для P, если каждый цикл в P содержит ребро, на котором имеется точка из M. Если M – базовое множество для P, то путь π в P называется **базовым** относительно M, если он непуст, на первом и последнем ребрах этого пути имеются точки из M, и на других рёбрах этого пути точек из M нет. Множество всех базовых относительно M путей в P обозначается записью $\Pi_{P,M}$.

Докажем, что множество $\Pi_{P,M}$ конечно. Если $\Pi_{P,M}$ бесконечно, то оно содержит пути сколь угодно большой длины. Тогда в $\Pi_{P,M}$ есть путь π вида $\alpha_1 \dots \alpha_i \dots \alpha_j \dots \alpha_k$, где $\alpha_i = \alpha_j$, $1 < i < j < k$. Последовательность $\alpha_i \dots \alpha_j$ является циклом, и, согласно определению базового множества, одно из рёбер этого цикла содержит точку из M, что противоречит предположению о том, что путь π – базовый относительно M.

$\forall\, \pi \in \Pi_{P,M}$ будем обозначать записями $start(\pi)$ и $end(\pi)$ точки из M, которые лежат на первом и последнем ребре π, соответственно.

Базовое множество M для БС P называется **полным**, если

- на ребре, выходящем из начальной вершины P, есть точка из M (такая точка называется **начальной**), и
- на каждом ребре, входящем в какую-либо терминальную вершину P, есть точка из M (такие точки называются **терминальными**).

Пусть M – полное базовое множество для БС P. Каждому выполнению $Exec$ БС P соответствует последовательность $\pi = \alpha_1\alpha_2\ldots$ рёбер P, в которой каждое ребро α_i является тем ребром, по которому происходит перемещение на шаге i этого выполнения от текущей вершины v_i к вершине v_{i+1}. Обозначим записью $i_1 i_2 \ldots$ последовательность номеров тех рёбер из π, на которых есть базовые точки. Согласно определению понятий полного базового множества и выполнения БС, $i_1 = 1$, и для каждой пары (i_j, i_{j+1}) соседних индексов в $i_1 i_2 \ldots$ последовательность $\pi_j = \alpha_{i_j} \ldots \alpha_{i_{j+1}}$ является базовым относительно M путём. Если путь π конечен, то его последнее ребро входит в терминальную вершину, и, следовательно, содержит точку из M. Таким образом, можно представить π в виде последовательности $\pi_1\pi_2\ldots$, где π_1, π_2, $\ldots$ – базовые относительно M пути. Каждый член π_j этой последовательности мы будем называть **компонентой** выполнения $Exec$.

$\forall\, \pi \in \Pi_{P,M}, \forall \xi, \xi' \in X_P^\bullet$ запись $\xi \xrightarrow{\pi} \xi'$ означает, что π – компонента некоторого выполнения БС P, и ξ, ξ' – текущие означивания в моменты прохода через точки $start(\pi)$ и $end(\pi)$ соответственно при движении по пути π во время этого выполнения.

2.2 Описание метода инвариантов для верификации блок-схем

Пусть заданы БС P и её спецификация, выражаемая предусловием Pre и постусловием $Post$. Доказательство утверждения $Pre \xrightarrow{P} Post$ **методом инвариантов** имеет следующий вид.

1. Выбирается полное базовое множество точек M для P, и с каждой точкой $m \in M$ связывается формула $\varphi_m \in \mathcal{B}$, называемая **инвариантом** в точке m, причем выполнены следующие условия:
   - если m – начальная точка, то $\varphi_m = Pre$,
   - если m – терминальная точка, то $\varphi_m = Post$,
   - для любого пути $\pi \in \Pi_{P,M}$, и любых означиваний $\xi, \xi' \in X_P^\bullet$, таких, что $\xi \xrightarrow{\pi} \xi'$, верна импликация
   $$\varphi^{\xi}_{start(\pi)} = 1 \quad \Rightarrow \quad \varphi^{\xi'}_{end(\pi)} = 1. \tag{2.1}$$

Ниже, при анализе импликаций вида (2.1), $\forall x \in X_P$ будем обозначать значения x^ξ и $x^{\xi'}$ записями x и x' соответственно.

2. Свойство завершаемости P ($\forall \xi \in X_P^\bullet \;\; Pre^\xi = 1 \Rightarrow \xi \xrightarrow{P} \otimes$) обосновывается путем указания

   - базового множества N для P,
   - частично упорядоченного множества L, являющегося **фундированным**, т.е. такого, что не существует бесконечной строго убывающей последовательности $l_0, l_1, \ldots$ элементов L, и
   - множества выражений $\{u_n \in \mathcal{E}(X_P) \mid n \in N\}$,

   таких, что

   - при каждом выполнении P, $\forall n \in N$ при каждом проходе через n текущее означивание ξ удовлетворяет условию $u_n^\xi \in L$, и
   - для любого пути $\pi \in \Pi_{P,N}$, и любых означиваний $\xi, \xi' \in X_P^\bullet$, таких, что $\xi \xrightarrow{\pi} \xi'$, верно неравенство $u^\xi_{start(\pi)} > u^{\xi'}_{end(\pi)}$.

Изложенный в этом пункте метод инвариантов был открыт Р. Флойдом [2] и впервые был изложен в [3].

2.3 Обоснование метода инвариантов

Докажем, что если выполнены условия в пунктах 1 и 2 параграфа 2.2, то $Pre \xrightarrow{P} Post$, т.е. $\forall \xi \in X_P^\bullet$ из $Pre^\xi = 1$ следует $\xi \xrightarrow{P} \otimes$ и $Post^{\xi P} = 1$.

Пусть $Exec$ – произвольное выполнение БС P с начальным означиванием ξ, удовлетворящим условию $Pre^\xi = 1$. Этому выполнению соответствует последовательность $\pi = \alpha_1\alpha_2\ldots$ рёбер P, в которой каждое ребро α_i является тем ребром, по которому происходит перемещение на шаге i этого выполнения от текущей вершины v_i к вершине v_{i+1}.

Если бы выполнение $Exec$ было бесконечным, то π тоже была бы бесконечной, и некоторое ребро встречалось бы в ней бесконечно много раз. Пусть $i_1 i_2 \ldots$ – бесконечная последовательность номеров рёбер из π, таких, что $\alpha_{i_1} = \alpha_{i_2} = \ldots$. Подпоследовательности $\pi_{i_j} = \alpha_{i_j} \ldots \alpha_{i_{j+1}}$ $(j \geq 1)$ последовательности π являются циклами, и т.к. N – базовое множество, то $\forall j \geq 1 \quad \pi_{i_j}$ содержит ребро, на котором есть точка из N. Таким образом, точки из N присутствуют на бесконечном числе членов последовательности π. Обозначим номера таких членов записями $j_1, j_2, \ldots$, а

точки из N на них – записями n_1, n_2, ... Пути $\alpha_{j_k} \dots \alpha_{j_{k+1}}$ $(k \geq 1)$ являются базовыми относительно N, поэтому, согласно пункту 2 параграфа 2.2, текущие означивания $\xi_{j_1}, \xi_{j_2}, \dots$ удовлетворяют неравенствам

$$u_{n_1}^{\xi_{j_1}} > u_{n_2}^{\xi_{j_2}} > \dots \tag{2.2}$$

т.е. в L есть бесконечная строго убывающая последовательность (2.2), что противоречит предположению о фундированности L.

Таким образом, выполнение $Exec$ является конечным. В соответствии со сказанным в конце пункта 2.1, можно представить π в виде конечной последовательности $\pi_1 \dots \pi_k$ путей из $\Pi_{P,M}$.

Согласно определениям выполнения и полного базового множества, а также условиям в пункте 1 параграфа 2.2

- $m_0 \stackrel{\text{def}}{=} start(\pi_1)$ – начальная точка, поэтому $\varphi_{m_0} = Pre$,
- $m_k \stackrel{\text{def}}{=} end(\pi_k)$ – терминальная точка, поэтому $\varphi_{m_k} = Post$ и
- $\forall\, i = 1, \dots, k-1 \quad m_i \stackrel{\text{def}}{=} end(\pi_{i-1}) = start(\pi_i) \in M$.

Кроме того, $\forall\, i = 1, \dots, k$ текущие означивания ξ_{i-1} и ξ_i вычисления $Exec$ до и после прохождения компоненты π_i последовательности $\pi_1 \dots \pi_k$ соответственно удовлетворяют условию $\xi_{i-1} \stackrel{\pi_i}{\to} \xi_i$, поэтому, согласно (2.1),

$$\forall\, i = 1, \dots, k \quad \varphi_{m_{i-1}}^{\xi_{i-1}} = 1 \;\Rightarrow\; \varphi_{m_i}^{\xi_i} = 1.$$

Суммируя все вышесказанное, получаем цепочку импликаций:

$$(Pre^{\xi} = 1) \Rightarrow (\varphi_{m_0}^{\xi} = 1) \Rightarrow (\varphi_{m_1}^{\xi_1} = 1) \Rightarrow \dots$$
$$\dots \Rightarrow (\varphi_{m_k}^{\xi_k} = 1) \Rightarrow (Post^{\xi P} = 1). \blacksquare$$

2.4 Примеры фундированных множеств

В качестве фундированных множеств, требуемых для обоснования завершаемости, можно брать, например, следующие множества:

- множество $\mathbf{N}$ натуральных чисел $(0, 1, \dots)$,
- множество L^k кортежей длины k элементов произвольного фундированного множества L, с лексикографическим порядком, который определяется следующим образом: для любой пары кортежей $a = (a_1, \dots, a_k)$, $b = (b_1, \dots, b_k)$ из L^k

  $a \leq b \Leftrightarrow a = b$ или $\exists\, i \in \{1, \dots, k\} : a_i < b_i, \forall\, j \in \{1, \dots, i-1\}\ a_j = b_j$.

Обоснуем фундированность этого множества. Пусть существует бесконечная строго убывающая последовательность

$$(a_1^1, \ldots, a_k^1) > (a_1^2, \ldots, a_k^2) > \ldots \tag{2.3}$$

элементов множества L^k. Из (2.3) и из определения лексикографического порядка следует, что $a_1^1 \geq a_1^2 \geq \ldots$. Поскольку L фундировано, то в этой последовательности неравенств не может быть бесконечного количества строгих неравенств, т.е.

$$\exists\, i_1 :\ a_1^{i_1} = a_1^{i_1+1} = \ldots. \tag{2.4}$$

Из последнего соотношения и из (2.3) следует, что

$$(a_2^{i_1}, \ldots, a_k^{i_1}) > (a_2^{i_1+1}, \ldots, a_k^{i_1+1}) > \ldots \tag{2.5}$$

Применяя изложенные выше рассуждения к (2.5), получаем, что

$$\exists\, i_2 \geq i_1 :\ a_2^{i_2} = a_2^{i_2+1} = \ldots, \tag{2.6}$$

откуда на основании (2.5) следует, что

$$(a_3^{i_2}, \ldots, a_k^{i_1}) > (a_3^{i_2+1}, \ldots, a_k^{i_1+1}) > \ldots.$$

Продолжая так и дальше, в конце концов получим, что

$$\exists\, i_k \geq i_{k-1} :\ a_k^{i_k} = a_k^{i_k+1} = \ldots. \tag{2.7}$$

Из (2.4), (2.6), ..., (2.7) следует, что кортежи в (2.3), начиная с кортежа с номером i_k, совпадают, что противоречит предположению. ■

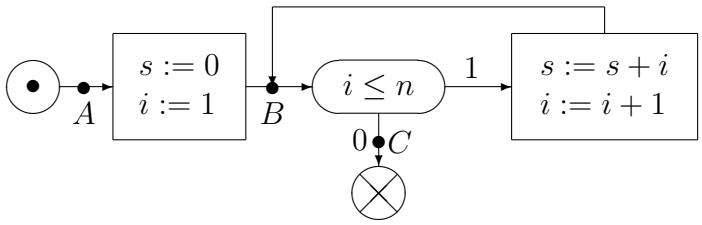
2.5 Примеры использования метода инвариантов

В этом пункте мы рассмотрим применение метода инвариантов для верификации БС, изложенных в пункте 1.4.

2.5.1 Верификация блок-схемы вычисления суммы

Верифицируем БС (1.4) относительно предусловия $n \geq 1$ и постусловия $s = n(n+1)/2$. В качестве множества M точек, необходимого для верификации этой БС методом инвариантов, возьмем множество $\{A, B, C\}$

точек, изображенных ниже:



Инварианты в точках A, B, C имеют следующий вид:

$$\varphi_A = (n \ge 1) \quad \text{(предусловие)},$$
$$\varphi_B = (s = (i-1)i/2) \wedge (i \le n+1),$$
$$\varphi_C = (s = n(n+1)/2) \quad \text{(постусловие)}.$$

Для доказательства того, что инвариант φ_B определен правильно, необходимо исследовать все этапы выполнения этой БС, которые начинаются и заканчиваются проходом через точки из M.

1. На первом этапе выполнения этой БС происходит переход от A к B, с выполнением присваиваний в блоке между A и B (т.е $s := 0$ и $i := 1$), и (2.1) имеет вид

$$n \ge 1 \;\Rightarrow\; \left[\!\!\left[\begin{array}{l} s' = (i'-1)i'/2 \\ i' \le n'+1 \end{array} \right]\!\!\right]. \tag{2.8}$$

   Т.к. $s' = 0$, $i' = 1$ и $n' = n$, то (2.8) переписывается в виде

$$n \ge 1 \;\Rightarrow\; \left[\!\!\left[\begin{array}{l} 0 = 0 \cdot 1/2 \\ 1 \le n+1 \end{array} \right]\!\!\right], \tag{2.9}$$

   что, очевидно, верно.

2. Рассмотрим произвольный этап выполнения этой БС, на котором происходит переход от B к B по циклу, здесь

   - выполняются присваивания $s := s + i$ и $i := i + 1$, и
   - верно условие, на основании которого возможно движение по этому циклу ($i \le n$).

   Импликация (2.1) с учетом этого условия имеет вид

$$\left[\!\!\left[\begin{array}{l} s = (i-1)i/2 \\ i \le n+1 \\ i \le n \end{array} \right]\!\!\right] \;\Rightarrow\; \left[\!\!\left[\begin{array}{l} s' = (i'-1)i'/2 \\ i' \le n'+1 \end{array} \right]\!\!\right] \tag{2.10}$$

   Учитывая соотношения $s' = s + i$, $i' = i + 1$, $n' = n$, заключаем, что импликация (2.10) верна.

3. Рассмотрим этап выполнения БС, на котором происходит переход от B к C. Он возможен только при условии $i > n$. Импликация (2.1) с учетом этого условия имеет вид

$$\left[\!\!\left[\begin{array}{l} s = (i-1)i/2 \\ i \le n+1 \\ i > n \end{array} \right]\!\!\right] \Rightarrow\ s' = n'(n'+1)/2. \qquad (2.11)$$

Учитывая равенства $s' = s$, $n' = n$, а также то, что i имеет тип `int`, и, следовательно, из $i \le n+1$ и $i > n$ следует, что $i = n+1$, заключаем, что импликация (2.11) верна.

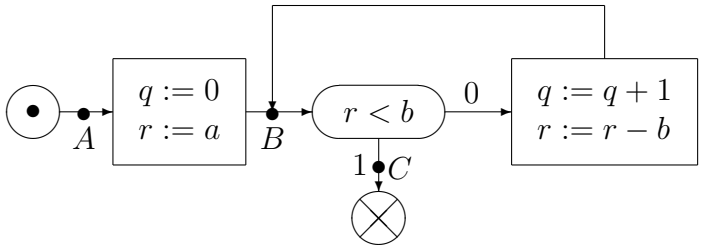
Для доказательства завершаемости положим $N \stackrel{\text{def}}{=} \{B\}$, $L \stackrel{\text{def}}{=} \mathbf{N}$, $u_B \stackrel{\text{def}}{=} n+1-i$. Из доказанного выше соотношения φ_B следует, что при каждом проходе через B текущее означивание ξ удовлетворяет условию $u_B^{\xi} \in \mathbf{N}$, и при произвольном переходе от B к B по циклу

$$u_B^{\xi} = (n+1-i) > (n+1-(i+1)) = (n'+1-i') = u_B^{\xi'},$$

где ξ и ξ' – текущие означивания до и после прохода по циклу.

2.5.2 Верификация блок-схемы деления с остатком

Верифицируем БС (1.5) относительно предусловия $(a \ge 0) \wedge (b \ge 1)$ и постусловия $(a = bq + r) \wedge (0 \le r < b)$. Выберем точки A, B, C, как показано на рисунке:



Инвариантами φ_A и φ_C являются предусловие и постусловие соответственно, а инвариант φ_B имеет вид $(a = bq + r) \wedge (r \ge 0)$.

Докажем, что φ_B определен правильно.

1. При первом входе в B φ_B имеет вид $(a = b \cdot 0 + a) \wedge (a \ge 0)$, истинность данного соотношения следует из φ_A.

2. При переходе от B к B по циклу верно дополнительное условие $r \geq b$, с учетом которого импликация (2.1) имеет вид

$$\left[\!\!\left[\begin{array}{l} a = bq + r \\ r \geq 0 \\ r \geq b \end{array} \right]\!\!\right] \Rightarrow \left[\begin{array}{l} a' = b'q' + r' \\ r' \geq 0 \end{array} \right]. \qquad (2.12)$$

Учитывая соотношения $a' = a$, $b' = b$, $q' = q + 1$, $r' = r - b$, заключаем, что импликация (2.13) верна.

3. При переходе от B к C верно дополнительное условие $r < b$, что в сочетании с φ_B влечёт φ_C.

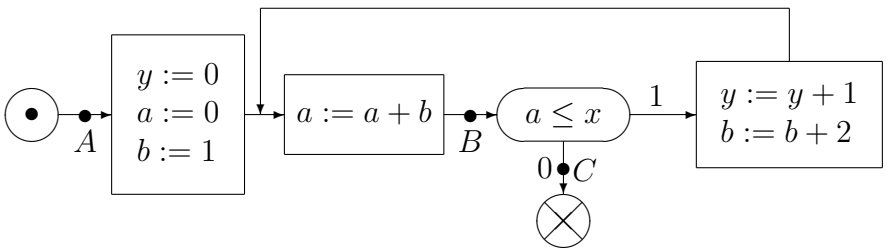
Для доказательства завершаемости положим $N \stackrel{\text{def}}{=} \{B\}$, $L \stackrel{\text{def}}{=} \mathbf{N}$, $u_B \stackrel{\text{def}}{=} r$. Из доказанного выше соотношения φ_B следует, что при каждом проходе через B текущее означивание ξ удовлетворяет условию $u_B^{\xi} \in \mathbf{N}$, и, с учетом дополнительного соотношения $b \geq 1$ (истинность которого в точке B следует из φ_A и из того, что значение переменной b не изменяется в процессе выполнения БС), заключаем, что при произвольном переходе от B к B по циклу

$$u_B^{\xi} = r > r - b = r' = u_B^{\xi'},$$

где ξ и ξ' – текущие означивания до и после прохода по циклу.

2.5.3 Верификация блок-схемы извлечения корня

Верифицируем БС (1.6) относительно предусловия $x \geq 0$ и постусловия $y = \lfloor \sqrt{x} \rfloor$. Выберем точки A, B, C, как показано на рисунке:



Определим инварианты в выбранных точках следующим образом:

$$\varphi_A \stackrel{\text{def}}{=} (x \geq 0),\ \varphi_B \stackrel{\text{def}}{=} \left[\!\!\left[\begin{array}{l} y \geq 0 \\ y^2 \leq x \\ a = (y+1)^2 \\ b = 2y + 1 \end{array} \right]\!\!\right],\ \varphi_C \stackrel{\text{def}}{=} \left[\begin{array}{l} y \geq 0 \\ y^2 \leq x < (y+1)^2 \end{array} \right]$$

(нетрудно видеть, что φ_C эквивалентна постусловию).

Докажем, что φ_B определен правильно.

1. При первом входе в B φ_B имеет вид $\left[\!\!\left[\begin{array}{l} 0 \geq 0 \\ 0^2 \leq x \\ 1 = (0+1)^2 \\ 1 = 2 \cdot 0 + 1 \end{array} \right]\!\!\right]$, истинность данного соотношения следует из φ_A.

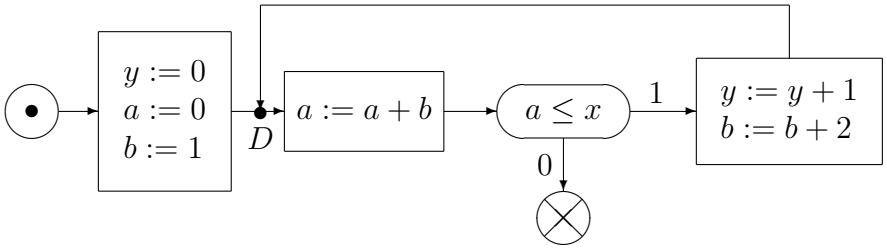
2. При переходе от B к B по циклу верно дополнительное условие $a \leq x$, с учетом которого импликация (2.1) имеет вид

$$\left[\!\!\left[\begin{array}{l} y \geq 0 \\ y^2 \leq x \\ a = (y+1)^2 \\ b = 2y + 1 \\ a \leq x \end{array} \right]\!\!\right] \Rightarrow \left[\!\!\left[\begin{array}{l} y' \geq 0 \\ (y')^2 \leq x' \\ a' = (y'+1)^2 \\ b' = 2y' + 1 \end{array} \right]\!\!\right]. \qquad (2.13)$$

   Учитывая соотношения $y' = y + 1$, $b' = b + 2$, $a' = a + b + 2$, $x' = x$, нетрудно установить, что импликация (2.13) верна.

3. При переходе от B к C верно дополнительное условие $x < a$, что в сочетании с φ_B влечёт φ_C.

Докажем завершаемость данной БС. Положим $N \stackrel{\text{def}}{=} \{D\}$, где точка D выбрана так, как показано на рисунке:



и, кроме того, $L \stackrel{\text{def}}{=} \mathbf{N}$, $u_D \stackrel{\text{def}}{=} x - a$. Для обоснования того, что

1. при каждом проходе через D текущее означивание ξ удовлетворяет условию $u_D^{\xi} \in \mathbf{N}$, и

2. при переходе от D к D по циклу $u_D^{\xi} = x - a > x' - a' = u_D^{\xi'}$, где ξ и ξ' – текущие означивания до и после прохода по циклу,

определим вспомогательный инвариант в D: $\varphi_D \stackrel{\text{def}}{=} (x \geq a) \wedge (b \geq 1)$.

Истинность φ_D при каждом проходе через D следует из того, что

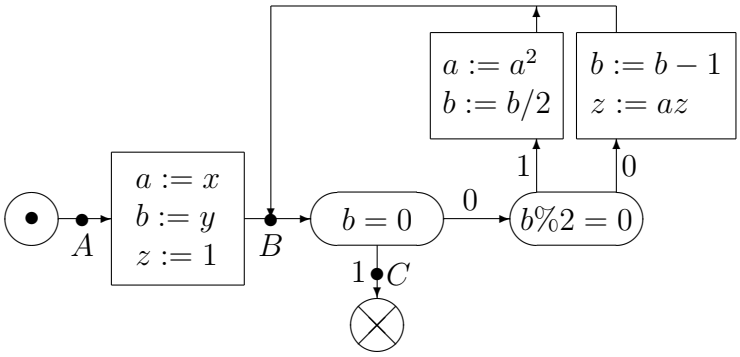
- φ_D верно при первом проходе через D (это следует из φ_A), и

- при произвольном переходе от D к D по циклу верно соотношение, написанное в условном операторе БС ($a \le x$).

Из истинности φ_D при каждом проходе через D следуют вышеупомянутые свойства 1 и 2, связанные с точкой D.

2.5.4 Верификация блок-схемы возведения в степень

Верифицируем БС (1.7) относительно предусловия $y \ge 0$ и постусловия $z = x^y$. Выберем точки A, B, C, как показано на рисунке:



Определим инварианты в выбранных точках следующим образом:

$$\varphi_A \stackrel{\text{def}}{=} (y \ge 0),\ \varphi_B \stackrel{\text{def}}{=} \left[\!\!\left[\begin{array}{l} b \ge 0 \\ y \ge 0 \\ za^b = x^y \end{array} \right]\!\!\right],\ \varphi_C \stackrel{\text{def}}{=} (z = x^y).$$

Нетрудно доказать, что φ_B определен правильно.

Для доказательства завершаемости данной БС можно взять $N \stackrel{\text{def}}{=} \{B\}$, $L \stackrel{\text{def}}{=} \mathbf{N}$, $u_B \stackrel{\text{def}}{=} b$.

2.5.5 Верификация блок-схемы сортировки

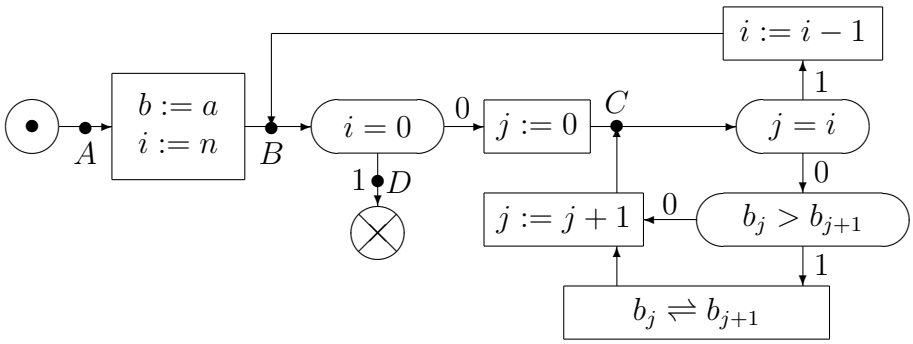
Верифицируем БС (1.8) относительно предусловия $n \ge 0$ и постусловия $(b = perm(a)) \wedge ord(b)$.

Конъюнктивные члены в постусловии обосновываются раздельно.

Обоснование утверждения $b = perm(a)$ заключается в том, что

- оно является верным после выполнения первого присваивания, и
- все остальные действия в этой БС сохраняют его истинность, т.к. единственное действие в БС, которое изменяет массив b – перестановка его соседних компонентов ($b_j \rightleftharpoons b_{j+1}$), однако эта перестановка не изменяет содержимое массива b.

Для обоснования утверждения $ord(b)$ выберем точки A, B, C, D, как показано на рисунке:



Инвариантами φ_A и φ_D являются формулы $(n \geq 0)$ и $ord(b_{0..n})$ соответственно. Инварианты φ_B и φ_C имеют следующий вид:

$$\varphi_B \stackrel{\text{def}}{=} \left[\!\!\left[\begin{array}{l} 0 \leq i \leq n \\ ord(b_{i..n}) \\ b_{0..i} \leq b_{i+1..n} \end{array} \right]\!\!\right], \quad \varphi_C \stackrel{\text{def}}{=} \left[\!\!\left[\begin{array}{l} 1 \leq i \leq n \\ 0 \leq j \leq i \\ ord(b_{i..n}) \\ b_{0..i} \leq b_{i+1..n} \\ b_{0..j-1} \leq b_j \end{array} \right]\!\!\right]$$

где используются обозначения, введенные в пункте 1.3.

Докажем, что инварианты φ_B и φ_C определены правильно.

1. При первом входе в B $\;i' = n$, и $\varphi_B^{\xi'} = \left[\!\!\left[\begin{array}{l} 0 \leq n \leq n \\ ord(b_{n..n}) \\ b_{0..n} \leq b_{n+1..n} \end{array} \right]\!\!\right]$. Истинность данного соотношения следует из φ_A.

2. При переходе от B к C верно дополнительное условие $i \neq 0$. Кроме того меняет свое значение только переменная j $(j' = 0)$. С учетом этого импликация (2.1) имеет вид

$$\left[\!\!\left[\begin{array}{l} 0 \leq i \leq n \\ ord(b_{i..n}) \\ b_{0..i} \leq b_{i+1..n} \\ i \neq 0 \end{array} \right]\!\!\right] \Rightarrow \left[\!\!\left[\begin{array}{l} 1 \leq i \leq n \\ 0 \leq 0 \leq i \\ ord(b_{i..n}) \\ b_{0..i} \leq b_{i+1..n} \\ b_{0..0-1} \leq b_0 \end{array} \right]\!\!\right] \qquad (2.14)$$

   Нетрудно видеть, что импликация (2.14) верна.

3. При переходе от C к B верно дополнительное условие $j = i$. Кроме

того, меняет свое значение только i. С учетом этого (2.1) имеет вид

$$\left[\!\!\left[\begin{array}{l} 1 \le i \le n \\ 0 \le j \le i \\ ord(b_{i..n}) \\ b_{0..i} \le b_{i+1..n} \\ b_{0..j-1} \le b_j \\ j = i \end{array} \right]\!\!\right] \Rightarrow \left[\!\!\left[\begin{array}{l} 0 \le i' \le n \\ ord(b_{i'..n}) \\ b_{0..i'} \le b_{i'+1..n} \end{array} \right]\!\!\right]$$

Т.к. $i' = i - 1$, то последнюю импликацию можно переписать в виде

$$\left[\!\!\left[\begin{array}{l} 1 \le i \le n \\ ord(b_{i..n}) \\ b_{0..i} \le b_{i+1..n} \\ b_{0..i-1} \le b_i \end{array} \right]\!\!\right] \Rightarrow \left[\!\!\left[\begin{array}{l} 0 \le i - 1 \le n \\ ord(b_{i-1..n}) \\ b_{0..i-1} \le b_{i..n} \end{array} \right]\!\!\right] \tag{2.15}$$

Нетрудно видеть, что импликация (2.15) верна.

4. При переходе от C к C по короткому циклу верны дополнительные условия $j \ne i$ и $b_j \le b_{j+1}$. Кроме того, меняет свое значение только переменная j ($j' = j + 1$). С учетом этого (2.1) имеет вид

$$\left[\!\!\left[\begin{array}{l} 1 \le i \le n \\ 0 \le j < i \\ ord(b_{i..n}) \\ b_{0..i} \le b_{i+1..n} \\ b_{0..j-1} \le b_j \\ b_j \le b_{j+1} \end{array} \right]\!\!\right] \Rightarrow \left[\!\!\left[\begin{array}{l} 1 \le i \le n \\ 0 \le j + 1 \le i \\ ord(b_{i..n}) \\ b_{0..i} \le b_{i+1..n} \\ b_{0..j} \le b_{j+1} \end{array} \right]\!\!\right] \tag{2.16}$$

Нетрудно видеть, что импликация (2.16) верна.

5. При переходе от C к C по длинному циклу верны дополнительные условия $j \ne i$ и $b_j > b_{j+1}$. Кроме того, меняют свое значение переменная j и массив b ($j' = j + 1$, $b'_j = b_{j+1}$, $b'_{j+1} = b_j$, $\forall\, k \in \{0, \ldots, n\} \setminus \{j, j+1\} \quad b'_k = b_k$). С учетом этого (2.1) имеет вид

$$\left[\!\!\left[\begin{array}{l} 1 \le i \le n \\ 0 \le j < i \\ ord(b_{i..n}) \\ b_{0..i} \le b_{i+1..n} \\ b_{0..j-1} \le b_j \\ b_{j+1} < b_j \end{array} \right]\!\!\right] \Rightarrow \left[\!\!\left[\begin{array}{l} 1 \le i \le n \\ 0 \le j + 1 \le i \\ ord(b'_{i..n}) \\ b'_{0..i} \le b'_{i+1..n} \\ b'_{0..j} \le b'_{j+1} \end{array} \right]\!\!\right]. \tag{2.17}$$

Отдельно рассмотрим случаи $j + 1 < i$ и $j + 1 = i$.

- Если $j+1<i$, то

$$\left\{\begin{array}{l} b'_0=b_0,\ldots,b'_{j-1}=b_{j-1},\\ b'_j=b_{j+1},b'_{j+1}=b_j,\\ b'_{j+2}=b_{j+2},\ldots,b'_i=b_i,\ldots,b'_n=b_n \end{array}\right.$$

откуда нетрудно обосновать (2.17).

- Если $j+1=i$, то (2.17) следует из импликации

$$\left[\!\!\left[\begin{array}{l} 1\le i\le n\\ ord(b_{i..n})\\ b_{0..i}\le b_{i+1..n}\\ b_{0..i-2}\le b_{i-1}\\ b_i<b_{i-1} \end{array}\right]\!\!\right] \Rightarrow \left[\!\!\left[\begin{array}{l} ord(b'_{i..n})\\ b'_{0..i}\le b'_{i+1..n}\\ b'_{0..i-1}\le b'_i \end{array}\right]\!\!\right] \qquad (2.18)$$

при условии $\left\{\begin{array}{l} b'_0=b_0,\ldots,b'_{i-2}=b_{i-2},\\ b'_{i-1}=b_i,b'_i=b_{i-1},\\ b'_{i+1}=b_{i+1},\ldots,b'_n=b_n. \end{array}\right.$

6. При переходе от B к D верно дополнительное условие $i=0$, что в сочетании с φ_B влечёт φ_D.

Докажем завершаемость БС (1.8). Положим $N\stackrel{\text{def}}{=}\{C\}$, $L\stackrel{\text{def}}{=}\mathbf{N}^2$ (лексикографический порядок), $u_C\stackrel{\text{def}}{=}(i,i-j)$. Из φ_C следует, что при проходе через C текущее означивание ξ удовлетворяет условию $u_C^\xi\in\mathbf{N}^2$, и при переходе от C к C по верхнему циклу $i'=i-1$, поэтому

$$u_C^\xi=(i,i-j)>(i-1,\ldots)=u_C^{\xi'},$$

а при переходе от C к C по нижним циклам $i'=i$, $j'=j+1$, поэтому

$$u_C^\xi=(i,i-j)>(i,i-(j+1))=(i',i'-j')=u_C^{\xi'},$$

где ξ и ξ' – текущие означивания до и после прохода по циклам.

Глава 3

Процессные представления блок-схем

Для автоматизации верификации БС P методом инвариантов целесообразно сначала преобразовать эту БС в определяемый ниже граф G_P, который называется **процессным представлением** БС P. Для определения процессного представления БС необходимо ввести излагаемые в следующих пунктах вспомогательные понятия.

3.1 Действия

Будем понимать под **элементарным действием (ЭД)** запись $x := e$ или $[\![b]\!]$, где $x \in \mathcal{X}$, $e \in \mathcal{E}$, $t(x) = t(e)$, $b \in \mathcal{B}$. ЭД вида $x := e$ называется **присваиванием** переменной x значения выражения e, а ЭД вида $[\![b]\!]$ называется **проверкой условия**, выражаемого формулой b.

ЭД вида $[\![b]\!]$, где формула b сама имеет вид $[\![b']\!]$ (т.е. является конъюнкцией), будет обозначаться без дублирующих квадратных скобок (т.е. записью $[\![b']\!]$).

Множество всех ЭД обозначается символом $\mathcal{A}$, множество всех конечных последовательностей действий обозначается записью $\mathcal{A}^*$. Элементы $\mathcal{A}^*$ называются **действиями**. Если $a \in \mathcal{A}$ и $A, A' \in \mathcal{A}^*$, то записи aA, Aa и AA' обозначают конкатенации ЭД a и последовательности A, или A и A', соответственно.

$\forall\, a \in \mathcal{A}$ и $\forall\, A \in \mathcal{A}^*$ записи X_a и X_A обозначают множества переменных, содержащихся в a и A соответственно.

$\forall\, a \in \mathcal{A}$, $\forall\, X : X_a \subseteq X \subseteq \mathcal{X}$, $\forall\, \xi \in X^\bullet$ ξa обозначает объект, который

- равен означиванию $\xi(x := e)$ (см. (1.1)), если $a = (x := e)$,

- равен означиванию ξ, если $a = [\![b]\!]$ и $b^\xi = 1$
- не определен, если $a = [\![b]\!]$ и $b^\xi = 0$.

$\forall A \in \mathcal{A}^*$, $\forall X : X_A \subseteq X \subseteq \mathcal{X}$, $\forall \xi \in X^\bullet$ ξA обозначает объект, который

- совпадает с $(\ldots((\xi a_1)a_2)\ldots)a_k$, если $A = a_1 \ldots a_k$, и все объекты ξa_1, $(\xi a_1)a_2$, …, $(\ldots((\xi a_1)a_2)\ldots)a_k$ определены,
- не определен, в противном случае.

Действие называется **редуцируемым**, если оно имеет вид $Aa[\![b]\!]A'$, где $A, A' \in \mathcal{A}^*$, $a \in \mathcal{A}$. **Редукцией** действия $Aa[\![b]\!]A'$ называется действие

- $A[\![b' \wedge b]\!]A'$, если $a = [\![b']\!]$,
- $A[\![(x := e)b]\!](x := e)A'$, если $a = (x := e)$
  (определение выражений вида $(x := e)e'$ см. в конце пункта 1.1).

Нетрудно доказать, что $\forall\, A \in \mathcal{A}^*$

- если A' – редукция A, то $\forall X : X_A \subseteq X \subseteq \mathcal{X}$, $\forall \xi \in X^\bullet$ $\xi A = \xi A'$ (т.е. ξA и $\xi A'$ либо оба не определены, либо определены и совпадают),
- $\exists\, A_1, \ldots, A_k$ $(k \geq 1) : A_1 = A$, $\forall\, i = 1, \ldots, k-1$ A_{i+1} – редукция A_i, и A_k нередуцируемо.

Действие A_k из предыдущего абзаца называется **нормальной формой** действия A, и обозначается записью $NF(A)$. Множество всех нередуцируемых действий обозначается записью $\mathcal{A}^*_n$. Из определения понятия редукции следует, что каждое действие из $\mathcal{A}^*_n$ содержит не более одной проверки условия.

$\forall\, A \in \mathcal{A}^*_n$ запись $[\![A]\!]$ обозначает ЭД $[\![b]\!]$, если оно является первым в A, и ЭД $[\![1]\!]$, если A не содержит проверки условия.

$\forall A \in \mathcal{A}^*_n$, $\forall e \in \mathcal{E}$ запись Ae обозначает выражение

$$(x_1 := e_1) \ldots (x_k := e_k)e,$$

где $(x_1 := e_1) \ldots (x_k := e_k)$ – список всех присваиваний, входящих в A.

$\forall A \in \mathcal{A}^*_n, \forall\, \varphi, \psi \in \mathcal{B}$ запись $\varphi \overset{A}{\to} \psi$ обозначает формулу

$$\varphi \wedge [\![A]\!] \to A\psi,$$

которая выражает утверждение: $\forall\, \xi \in X^\bullet$, где $X_\varphi \cup X_A \cup X_\psi \subseteq X \subseteq \mathcal{X}$, если $\varphi^\xi = 1$ и ξA определено, то $\psi^{\xi A} = 1$.

3.2 Понятие процессного графа

Процессным графом (ПГ) называется конечный граф G со следующими свойствами:

- граф G имеет выделенные вершины G^0 и $G^\otimes$, называемые **начальной** и **терминальной** вершинами соответственно, при изображении ПГ в виде рисунка данные вершины обозначаются символами $\odot$ и $\otimes$ соответственно,
- каждому ребру α графа G сопоставлена метка $A_\alpha \in \mathcal{A}_n^*$,
- $G^\otimes$ – единственная вершина G, из которой не выходят рёбра.

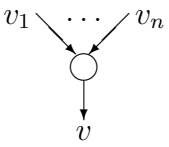
Пусть G – ПГ. Будем обозначать записями V_G и X_G совокупность всех вершин G и переменных, входящих в G, соответственно. Будем предполагать, что в каждом ПГ G среди его переменных присутствует переменная at_G, множеством значений которой является V_G, и метка каждого ребра α содержит присваивание вида $at_G := v$, где v – конец ребра α. В записи меток рёбер ПГ данное присваивание будет опускаться.

Выполнение ПГ G – это обход вершин G, начиная с G^0, с выполнением действий, сопоставленных проходимым рёбрам. С каждым шагом $i \geq 0$ выполнения G связаны вершина v_i этого графа и означивание $\xi_i \in X_G^\bullet$ (называемые **текущей вершиной** и **текущим означиванием** на шаге i). Если $v_i = G^\otimes$, то выполнение G на шаге i завершается, иначе выполнение на шаге i заключается в

- выборе произвольного ребра α, которое выходит из v_i и удовлетворяет условию $[\![A_\alpha]\!]^{\xi_i} = 1$,
- замене текущего означивания ξ_i на означивание $\xi_{i+1} \stackrel{\text{def}}{=} \xi_i A_\alpha$, и
- переходе в конец выбранного ребра, который будет текущей вершиной v_{i+1} на $i+1$–м шаге выполнения.

3.3 Построение процессного представления блок-схем

Пусть задана БС P. Алгоритм построения ПГ G_P, называемого **процессным представлением** БС P, состоит из следующих шагов.

1. Удаляются пустые операторы: каждый фрагмент вида 

заменяется на 

.

2. Удаляется начальная вершина P и выходящее из неё ребро, конец этого ребра – начальная вершина G^0_P графа G_P.

3. Удаляются терминальные вершины P, кроме одной, которая является терминальной вершиной $G^\otimes_P$ графа G_P, рёбра с концами в удаляемых вершинах перенаправляются в $G^\otimes_P$.

4. Каждая оставшаяся вершина v БС P является вершиной ПГ G_P, и

   - если v имеет вид $\boxed{x := e}$, и P содержит ребро $v \to v'$, то G_P содержит ребро $v \xrightarrow{x:=e} v'$,
   - если v имеет вид $\textcircled{b}$, и P содержит рёбра $v \xrightarrow{1} v'$ и $v \xrightarrow{0} v''$, то G_P содержит рёбра $v \xrightarrow{[\![b]\!]} v'$ и $v \xrightarrow{[\![\neg b]\!]} v''$.

Получившийся ПГ G_P может быть редуцирован путем применения операции **удаления вершины**: если v – вершина ПГ G_P, отличная от G^0_P и $G^\otimes_P$, в G_P нет ребер вида $v \xrightarrow{A} v$, и списки ребер, входящих в v и выходящих из v, имеют вид

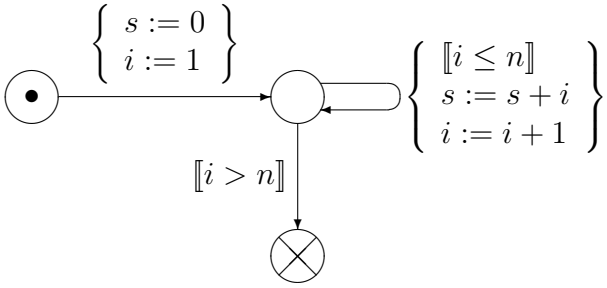
$$v_1 \xrightarrow{A_1} v, \ldots, v_k \xrightarrow{A_k} v \quad \text{и} \quad v \xrightarrow{A'_1} v'_1, \ldots, v \xrightarrow{A'_{k'}} v'_{k'} \tag{3.1}$$

соответственно, то операция удаления v из G_P заключается в удалении этой вершины и замене всех ребер из (3.1) на ребра вида $v_i \xrightarrow{A_{ij}} v'_j$, где $i \in \{1, \ldots, k\}$, $j \in \{1, \ldots, k'\}$, и $A_{ij} \stackrel{\text{def}}{=} NF(A_i A'_j)$.

Результат применения этой операции обозначается той же записью G_P и рассматривается как ПГ, эквивалентный в некотором смысле исходному ПГ. Данная операция может быть применена несколько раз.

Приведем пример процессного представления БС: ПГ, полученный удалением нескольких вершин из ПГ, построенного по БС (1.4), имеет

вид



3.4 Верификация блок-схем с использованием процессного представления

Пусть задан ПГ G. Для каждого пути $v_0 \stackrel{A_1}{\to} \ldots \stackrel{A_k}{\to} v_k$ в G записи $start(\pi)$ и $end(\pi)$ обозначают начало v_0 и конец v_k пути π соответственно, и запись A_π обозначает действие $NF(A_1 \ldots A_k)$. Путь π называется **циклом**, если $start(\pi) = end(\pi)$.

Множество вершин M ПГ G называется **базовым**, если для каждого цикла в G хотя бы одна из его вершин лежит в M. Если, кроме того, M содержит G^0 и $G^\otimes$, то M называется **полным** базовым множеством. Путь π в G называется **базовым** относительно M, если он непуст, $start(\pi)$ и $end(\pi)$ лежат в M, а остальные вершины π не лежат в M. Нетрудно доказать, что множество $\Pi_{G,M}$ всех базовых относительно M путей в G конечно.

Теорема.

Пусть заданы БС P и её спецификация, выражаемая предусловием Pre и постусловием $Post$. Тогда утверждение $Pre \stackrel{P}{\to} Post$ верно, если

- существует полное базовое множество M вершин G_P, причем с каждой вершиной $m \in M$ связана формула $\varphi_m \in \mathcal{B}$, и
  - $\varphi_{G_P^0} = Pre$, $\varphi_{G_P^\otimes} = Post$,
  - $\forall\, \pi \in \Pi_{G_P,M}\ \ (\varphi_{start(\pi)} \stackrel{A_\pi}{\to} \varphi_{end(\pi)}) = 1$,
- существуют базовое множество N вершин G_P и фундированное множество L, такие, что с каждой вершиной $n \in N$ связано выражение $u_n \in \mathcal{E}(X_P)$, причем выполнены условия:

– если при каком-либо выполнении G_P текущей вершиной в какой-либо момент является вершина $n \in N$, то текущее означивание ξ в этот момент удовлетворяет условию $u_n^\xi \in L$,

– $\forall\, \pi \in \Pi_{G_P,N} \quad \Big([\![A_\pi]\!] \to (u_{start(\pi)} > A_\pi u_{end(\pi)})\Big) = 1.$

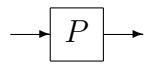
Для доказательства теоремы заметим, что если существуют множества M, N и L, упоминаемые в формулировке этой теоремы, то по ним нетрудно построить аналогичные множества M_P, N_P, L_P, необходимые для доказательства $Pre \overset{P}{\to} Post$ на основе метода инвариантов. Действительно, если вершина m графа G_P принадлежит множеству M, то этой вершине соответствует некоторая вершина в исходной БС P. Мы зачисляем в M_P точки на всех рёбрах P, входящих в эту вершину, и сопоставляем каждой такой точке формулу φ_m. Аналогично определяется N_P. В качестве L_P можно взять L. Проверка всех условий, упоминаемых в формулировке метода инвариантов, является несложной. ■

Глава 4

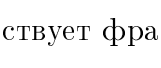
Верификация программ, представленных в операторной форме

4.1 Понятие программы в операторной форме

Еще один способ описания программ заключается в представлении их в операторной форме. В настоящий момент данная форма описания программ является наиболее широко распространенной.

Программа в операторной форме представляет собой оператор из определяемого ниже множества $\mathcal{P}$ операторов. В нижеследующем определении мы поясняем смысл каждого оператора $P \in \mathcal{P}$ путем указания соответствующего ему фрагмента БС, который будем обозначать записью вида 

.

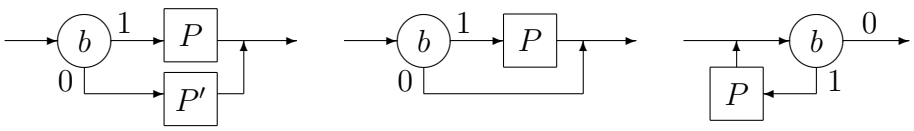
1. $\forall x \in \mathcal{X}, \forall e \in \mathcal{E} : t(x) = t(e)$, запись $x := e$ является оператором из $\mathcal{P}$, ему соответствует фрагмент БС 

.

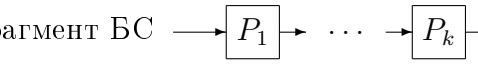
2. $\forall b \in \mathcal{B}, \forall P, P' \in \mathcal{P}$ записи

$$\textbf{if } b \textbf{ then } P \textbf{ else } P', \quad \textbf{if } b \textbf{ then } P, \quad \textbf{while } b \textbf{ do } P \tag{4.1}$$

являются операторами из $\mathcal{P}$, им соответствуют фрагменты БС

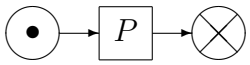


(последняя из программ (4.1) называется **циклом**),

3. $\forall\, P_1, \ldots, P_k \in \mathcal{P}$ $(k \geq 1)$ запись $\{P_1; \ldots; P_k\}$ является оператором, ему соответствует фрагмент БС  .

   Данный оператор может обозначаться также другой записью, в которой входящие в него операторы $P_1, \ldots, P_k$ расположены в столбик (при этом фигурные скобки могут отсутствовать), в этом случае точка с запятой для разделения $P_1, \ldots, P_k$ не используется.

$\forall\, P \in \mathcal{P}$ БС, соответствующая этому оператору, имеет вид



Мы будем обозначать эту БС тем же символом P.

Можно доказать, что определенная выше система операторов является алгоритмически полной, т.е. любую функцию, вычислимую по Тьюрингу, можно представить в виде программы в операторной форме.

Наряду с определенными выше видами операторов используются также и другие виды операторов, например:

- **do** P **while** b, где $P \in \mathcal{P}$ и $b \in \mathcal{B}$, этот оператор эквивалентен оператору $\{P;\ \textbf{while}\ b\ \textbf{do}\ P\}$,
- **for** $i := e_1$ **step** e_2 **until** e_3 **do** P, где i – переменная типа `int`, e_1, e_2, e_3 – выражения типа `int`, и $P \in \mathcal{P}$, этот оператор эквивалентен оператору

$$\{i := e_1;\ \textbf{while}\ i \neq e_3\ \textbf{do}\ \{P;\ i := i + e_2\};\ P\}.$$

- **go to** L, где L – метка какого-либо оператора, для возможности использования этого оператора необходимо ввести новое синтаксическое правило: если P – оператор, и L – символ, то запись $L : P$ является оператором (в котором L рассматривается как метка),
- **stop**, выполнение этого оператора приводит к завершению работы.

4.2 Примеры программ в операторной форме

Все переменные в излагаемых ниже примерах имеют тип `int`. Для каждой из представленных программ мы указываем её предусловие Pre и постусловие $Post$.

1. Программа P_1, вычисляющая $y = \lfloor \sqrt{x} \rfloor$:

$$\{y := 0;\ a := 1;\ b := 1\}$$
$$\textbf{while } (a \le x) \textbf{ do } \left\{ \begin{array}{l} y := y + 1 \\ a := a + b + 2 \\ b := b + 2 \end{array} \right\}$$

   $Pre : x \ge 0$, $Post : (y \ge 0) \wedge (y^2 \le x < (y+1)^2)$.

2. Программа P_2, вычисляющая наибольший общий делитель (НОД) $c = gcd(a, b)$ положительных чисел a и b:

$$\{c := a;\ d := b\}$$
$$\textbf{while } (c \ne d) \textbf{ do } \{\textbf{if } c > d \textbf{ then } c := c - d \textbf{ else } d := d - c\}$$

   $Pre : a, b > 0$, $Post : c = \max\{x \mid \exists\, a', b' : a = xa', b = xb'\}$.

3. Программа P_3, вычисляющая НОД $c = gcd(a, b)$ положительных чисел a и b:

$$\{x := a;\ y := b;\ z := 1\}$$
$$\textbf{while } (x\%2 = 0) \textbf{ do} \left\{ \begin{array}{l} \textbf{if } y\%2 = 0 \textbf{ then } (y, z) := (y/2, 2z) \\ x := x/2 \end{array} \right\}$$
$$\textbf{while } (y\%2 = 0) \vee (x \ne y) \textbf{ do}$$
$$\left\{ \begin{array}{l} \textbf{if } (y\%2 = 1) \textbf{ then } (x, y) := (y, |x - y|) \\ y := y/2 \end{array} \right\}$$
$$c := xz$$

   Предусловие и постусловие для P_3 – те же, что и для P_2.

4.3 Метод инвариантов для верификации программ в операторной форме

Как и для БС, спецификация программ из $\mathcal{P}$ м.б. задана в виде предусловия и постусловия. Задача верификации программы $P \in \mathcal{P}$ относительно предусловия $Pre \in \mathcal{B}(X_P)$ и постусловия $Post \in \mathcal{B}(X_P)$ заключается в обосновании утверждения $Pre \xrightarrow{P} Post$, где $\forall\, P \in \mathcal{P},\ \forall\, \varphi, \psi \in \mathcal{B}(X_P)$ запись $\varphi \xrightarrow{P} \psi$ обозначает утверждение

$$\forall\, \xi \in X_P^\bullet \quad \text{если } \varphi^\xi = 1, \text{ то } \xi \xrightarrow{P} \otimes \text{ и } \psi^{\xi P} = 1, \tag{4.2}$$

и P в (4.2) обозначает соответствующую БС.

В описании излагаемого ниже метода инвариантов предполагается, что анализируемый оператор P состоит только из операторов присваивания, операторов вида (4.1), и операторов вида $\{P_1;\ \ldots;\ P_k\}$.

Метод инвариантов обоснования утверждения вида $\varphi \overset{P}{\to} \psi$, где $P \in \mathcal{P}$ и $\varphi, \psi \in \mathcal{B}(X_P)$, имеет следующий вид.

1. Если $P = (x := e)$ то обоснование утверждения $\varphi \overset{P}{\to} \psi$ сводится к доказательству импликации $\varphi \to (x := e)\psi$.

2. Если $P =$ **if** b **then** P_1 **else** P_2, то обоснование утверждения $\varphi \overset{P}{\to} \psi$ сводится к обоснованию утверждений $\varphi \wedge b \overset{P_1}{\to} \psi$ и $(\varphi \wedge \neg b) \overset{P_2}{\to} \psi$.

3. Если $P =$ **if** b **then** P_1, то обоснование утверждения $\varphi \overset{P}{\to} \psi$ сводится к обоснованию утверждений $\varphi \wedge b \overset{P_1}{\to} \psi$ и $(\varphi \wedge \neg b) \to \psi$.

4. Если $P =$ **while** b **do** P, то обоснование утверждения $\varphi \overset{P}{\to} \psi$ сводится к построению формулы $I \in \mathcal{B}$ (называемой **инвариантом цикла**), фундированного множества L, и выражения u, таких, что

   - доказуемы импликации $\varphi \to I$, $I \to (u \in L)$, $(I \wedge \neg b) \to \psi$, и
   - можно обосновать утверждение

   $$I \wedge b \wedge (u = z) \overset{P}{\to} I \wedge (u < z), \tag{4.3}$$

   где z – новая переменная (т.е. не входит в I, b, u, P).

5. Если $P = \{P_1; \ldots; P_k\}$, где $k \geq 2$, то обоснование утверждения $\varphi \overset{P}{\to} \psi$ сводится к построению формулы $\eta \in \mathcal{B}$, такой, что можно обосновать утверждения $\varphi \overset{P'}{\to} \eta$ и $\eta \overset{P_k}{\to} \psi$, где $P' = \{P_1; \ldots; P_{k-1}\}$. Если P_k имеет вид $x := e$, то $\eta = (x := e)\psi$.

4.4 Пример верификации программы в операторной форме

Приведем пример верификации программы P_1 из пункта 4.2. Данная программа имеет вид $\{B_1; B_2\}$, где B_2 является циклом. Определим

$$\eta \overset{\text{def}}{=} (y^2 \leq x) \wedge (a = (y+1)^2) \wedge (b = 2y+1) \wedge (b > 0) \wedge (x - a + b \geq 0).$$

Нетрудно доказать, что

- $Pre \overset{B_1}{\to} \eta$, т.е. $(x \geq 0) \to (y := 0)(a := 1)(b := 1)\eta$,
- $\eta \overset{B_2}{\to} Post$, в качестве инварианта цикла B_2 можно взять η, в качестве фундированного множества L – множество $\mathbf{N}$ натуральных чисел, и в качестве выражения u – выражение $x - a + b$.

Отметим, что конъюнктивный член $(u < z)$ в заключении импликации (4.3) для данного случая обосновывается импликацией

$$(b > 0) \wedge (x - a + b = z) \to (x - a < z).$$

Глава 5

Верификация параллельных программ

5.1 Понятие параллельной программы

В настоящем тексте под **параллельной программой** будем понимать конечную совокупность **последовательных программ**, каждая из которых представлена в виде ПГ. Если параллельная программа состоит из последовательных программ $P_1, \ldots, P_k$, то будем обозначать её записью $(P_1, \ldots, P_k)$. Запись $X_{(P_1,\ldots,P_k)}$ обозначает совокупность всех переменных, входящих в какую-либо из программ $P_1, \ldots, P_k$.

Выполнение параллельной программы $(P_1, \ldots, P_k)$ происходит путем совместного выполнения входящих в неё последовательных программ $P_1, \ldots, P_k$ в соответствии с описанием в пункте 3.2. В каждый момент этого выполнения определено текущее означивание $\xi \in X^{\bullet}_{(P_1,\ldots,P_k)}$. Будем предполагать, что $\forall\, i = 1, \ldots, k$ каждый шаг выполнения P_i состоит из

- ожидания в текущей вершине (длительность этого ожидания может быть произвольной), и
- мгновенного перехода в вершину, которая будет текущей на следующем шаге, и изменения текущего означивания в соответствии с меткой того ребра, по которому был совершен переход,

причем в каждый момент времени возможно выполнение перехода не более чем в одной из программ $P_1, \ldots, P_k$. Если в некоторый момент выполнения $(P_1, \ldots, P_k)$ одна из программ P_i изменила текущее означивание ξ на новое означивание ξ', то ξ' станет новым текущим означиванием для всех программ $P_1, \ldots, P_k$.

Приведенное выше описание выполнения параллельной программы $P = (P_1, \ldots, P_k)$ не является точным описанием реального выполнения этой программы. Ниже мы рассматриваем различные несоответствия между реальным выполнением параллельных программ и тем понятием выполнения, которое определено в нашей модели, и обосновываем адекватность понятия выполнения в нашей модели.

1. При реальном выполнении программ переход из одного состояния каждой из компонент P_i в другое состояние этой компоненты происходит не мгновенно, а длится столько времени, сколько длится исполнение действия, связанного с этим переходом. Мы считаем, что каждый переход можно рассматривать как мгновенный, т.к.

   - до тех пор, пока выполнение действия, связанного с этим переходом, не завершилось, можно считать что компонента P_i остается в том состоянии, в котором она находилась в момент начала выполнения этого действия, и
   - как только выполнение этого действия завершилось, состояние компоненты P_i мгновенно изменилось.

2. Во время реального выполнения P допускается одновременное выполнение некоторых действий, относящихся к разным компонентам P_i, P_j этой программы. Будем предполагать, что при реальном выполнении P могут выполняться одновременно лишь такие пары действий, которые удовлетворяют следующему условию: если в одном из этих действий производится изменение значения некоторой переменной, то эта переменная не входит в другое из этих действий. Нетрудно видеть, что если пара действий, удовлетворяющая сформулированному выше условию, выполняется одновременно, то значения переменных программы изменятся так же, как если бы данные действия исполнились не одновременно, а по очереди. Таким образом, для анализа логических свойств какой-либо параллельной программы можно без ограничения общности предполагать, что выполнение этой программы происходит в соответствии с нашей моделью, т.е. действия в компонентах, из которых состоит эта параллельная программа, никогда не выполняются одновременно.

Отметим также следующий момент. В нашей модели действия компонентов параллельной программы являются последовательностями элементарных действий. Согласно определению понятия выполнения ПГ, данные последовательности выполняются неразрывно, т.е. если в некоторый момент компонента P_i параллельной программы $(P_1, \ldots, P_k)$ начала выполнять действие $A = a_1 \ldots a_n$, то невозможна такая ситуация что

после выполнения части $a_1 \dots a_j$ $(j < n)$ последовательности элементарных действий A происходит приостановка активности компоненты P_i и передача управления другой компоненте этой параллельной программы. Для того чтобы реальная программа реализовала именно такой порядок выполнения, нужно использовать специальные программные средства (семафоры, локи, и т.п.). Описание данных средств не входит в нашу модель: мы считаем, что при реальном выполнении программы $(P_1, \dots, P_k)$ описанная выше дисциплина неразрывного выполнения последовательностей элементарных действий соблюдается, но не уточняем как именно реализована данная дисциплина.

Будем говорить, что параллельная программа $P = (P_1, \dots, P_k)$ **выполняется с начальным означиванием** $\xi \in X_P^\bullet$, если в момент начала выполнения P значение каждой переменной $x \in X_P$ равно x^ξ, и $\forall\, i = 1, \dots, k \;\; at_{P_i}^\xi = P_i^0$. Запись $\xi \xrightarrow{P} \otimes$ обозначает утверждение: если P выполняется с начальным означиванием ξ, то в некоторый момент времени текущее означивание ξ' удовлетворяет условию $\forall\, i = 1, \dots, k \;\; at_{P_i}^{\xi'} = \otimes$. Будем обозначать такое означивание ξ' записью ξP.

5.2 Спецификация и верификация параллельных программ

Как и для последовательных программ, спецификация параллельной программы м.б. задана в виде предусловия и постусловия. Задача верификации параллельной программы $P = (P_1, \dots, P_k)$ относительно предусловия $Pre \in \mathcal{B}(X_P)$ и постусловия $Post \in \mathcal{B}(X_P)$ заключается в обосновании утверждения $Pre \xrightarrow{P} Post$, которое имеет вид

$$\forall \xi \in X_P^\bullet \quad \text{если} \;\; Pre^\xi = 1, \;\; \text{то} \;\; \xi \xrightarrow{P} \otimes \;\; \text{и} \;\; Post^{\xi P} = 1. \qquad (5.1)$$

Метод инвариантов обоснования $Pre \xrightarrow{P} Post$ для параллельной программы P заключается в построении формулы $I \in \mathcal{B}(X_P)$ (называемой **инвариантом** программы P), фундированного множества L, и выражения $u \in \mathcal{E}(X_P)$, таких, что доказуемы следующие импликации:

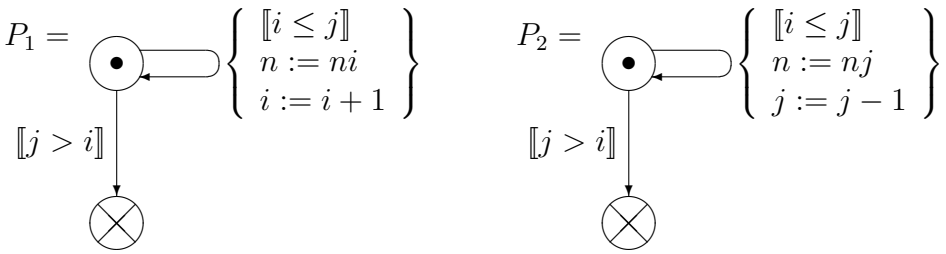
- $\Big(\bigwedge\limits_{i=1}^{k} (at_{P_i} = P_i^0) \Big) \wedge Pre \to I$, $\Big(\bigwedge\limits_{i=1}^{k} (at_{P_i} = P_i^\otimes) \Big) \wedge I \to Post$,
- $I \to (u \in L)$, $I \wedge [\![A_\alpha]\!] \to (A_\alpha I \wedge (u > A_\alpha u))$,
  где α – произвольное ребро какого-либо из ПГ $P_1, \dots, P_k$,

- $I \wedge \Big((v_1, \ldots, v_k) \neq (P_1^{\otimes}, \ldots, P_k^{\otimes})\Big) \to \bigvee\limits_{\alpha \in \langle v_1, \ldots, v_k \rangle} [\![A_\alpha]\!]$, где $v_i \in V_{P_i}$ $(i = 1, \ldots, k)$, и $\langle v_1 \ldots v_k \rangle$ – множество всех ребер с началом в одной из вершин $v_1, \ldots, v_k$.

5.3 Примеры спецификации и верификации параллельных программ

5.3.1 Вычисление факториала

Программа $P = (P_1, P_2)$ предназначена для вычисления $n = k!$, где k – входное значение. Программы P_1 и P_2 имеют следующий вид:



Спецификация: $\begin{cases} Pre = \ (n = 1) \wedge (i = 1) \wedge (j = k), \\ Post = \ (n = k!). \end{cases}$

Для верификации можно использовать следующие I, L, u:

$I = \left[\!\!\left[\begin{array}{l} i \le j + 1 \\ n = (i-1)! \frac{k!}{j!} \\ (at_{P_1} = P_1^{\otimes}) \vee (at_{P_2} = P_2^{\otimes}) \to (i > j) \end{array} \right]\!\!\right]$,

$L = \mathbf{N}$,

$u = j + 1 - i + (at_{P_1} = P_1^0) + (at_{P_2} = P_2^0)$.

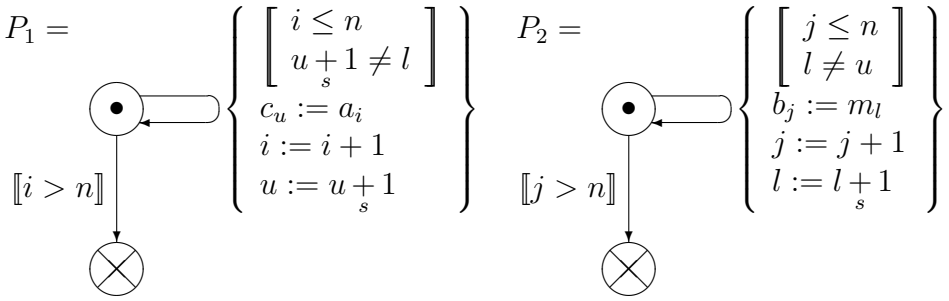
(Значения выражений $at_{P_k} = P_k^0$ $(k = 1, 2)$ рассматриваются в выражении u как натуральные числа 1 и 0.)

5.3.2 Передача сообщений через ограниченный буфер

Программа $P = (P_1, P_2)$ предназначена для записи значений массива $a_{1..n}$ в соответствующие компоненты массива $b_{1..n}$. Компоненты массива b недоступны программе P_1, и компоненты массива a недоступны программе P_2, т.е. выполнение действий вида $b_i := a_i$ невозможно. Для решения указанной выше задачи используются

- массив $c_{0..s-1}$, доступный программам P_1 и P_2, он рассматривается как буфер между P_1 и P_2: в нем содержатся значения, которые посланы программой P_1, но пока не получены программой P_2, и
- доступные обоим программам P_1 и P_2 переменные l и u, принимающие значения в множестве $\{0, \ldots, s-1\}$, значения данных переменных определяют область массива c, хранящую отправленные, но пока еще не доставленные значения, операции с l и u выполняется по модулю s, мы будем обозначать их записями $\underset{s}{+}$ и $\underset{s}{-}$.

Программы P_1 и P_2 имеют следующий вид:



Спецификация:

$$Pre = (i = j = 1) \wedge (u = l = 0) \wedge (n \geq 1) \wedge (s \geq 1),$$
$$Post = \bigwedge_{i=1}^{n} (b_i = a_i).$$

Для верификации можно использовать следующие I, L, u:

$$I = \left[\!\!\left[\begin{array}{l} (i \leq n+1) \wedge (j \leq n+1) \\ a_{1..i-1} = b_{1..j-1} c_{l..u-1} \\ \qquad\qquad\qquad\quad \underset{s}{} \\ (at_{P_1} = P_1^{\otimes}) \rightarrow (i = n+1) \\ (at_{P_2} = P_2^{\otimes}) \rightarrow (j = n+1) \end{array} \right]\!\!\right],$$

$L = \mathbf{N}$, $u = (n+1-i) + (n+1-j) + (at_{P_1} = P_1^0) + (at_{P_2} = P_2^0)$.

Запись $a_{1..i-1} = b_{1..j-1} c_{l..u-1}$ понимается следующим образом: если $i > 1$, $j > 1$, $l \neq u$, то часть $a_{1..i-1}$ массива a является конкатенацией части $b_{1..j-1}$ массива b и части $c_{l..u-1}$ массива c (при этом, если $0 < u < l$, то $c_{l..u-1} \stackrel{\text{def}}{=} c_{l..s-1} c_{0..u-1}$), и в остальных случаях указанная выше запись тоже определяется естественным образом.

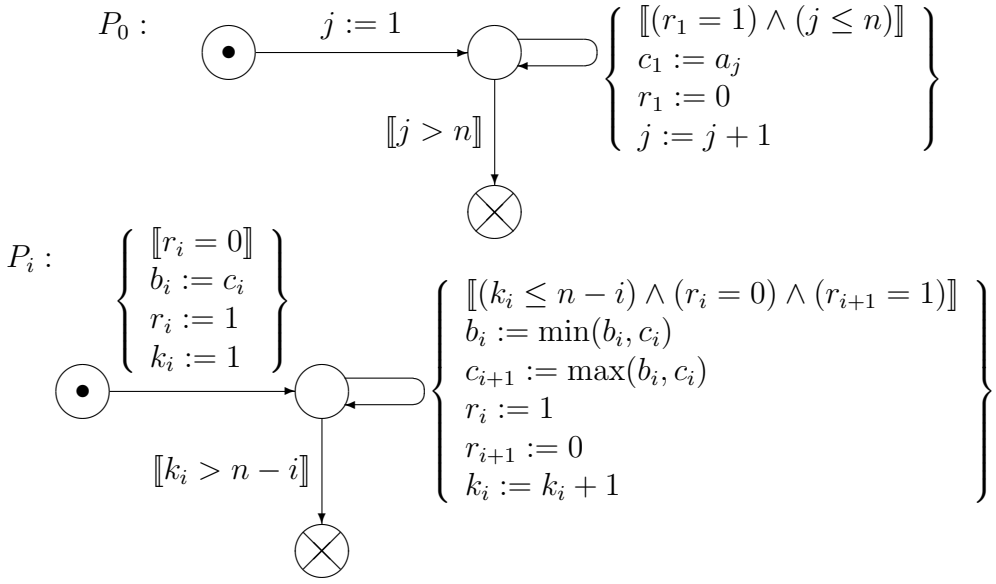
5.3.3 Параллельная сортировка

Программа $P = (P_0, P_1, \ldots, P_n)$ предназначена для сортировки массива $a_{1..n}$, результат записывается в массив $b_{1..n}$.

Как известно, нижняя оценка временно́й сложности любого алгоритма сортировки массива из n элементов на однопроцессорной машине равна $O(n \log_2 n)$, однако в том случае, когда число доступных процессоров для сортировки массива из n элементов больше n, временна́я сложность решения этой задачи может быть понижена до $O(n)$ за счет возможности параллельной работы процессоров. Ниже излагается параллельная программа, которую можно исполнять на $n+1$–процессорной машине, получая при этом указанное выше понижение временно́й сложности.

Вспомогательные массивы: $c_{1..n}$ и булевский массив $r_{1..n}$ ($r_i = 1 \Leftrightarrow$ компонента c_i не инициализирована или уже обработана).

Программы P_0 и P_i $(i = 1, \ldots, n)$ имеют следующий вид:



Спецификация: $\begin{cases} Pre = (r_1 = 1) \wedge \ldots \wedge (r_n = 1), \\ Post = perm(a, b) \wedge ord(b). \end{cases}$

Для верификации можно использовать следующий инвариант:

$$I = \left[\!\!\left[\bigvee_{j=1}^{n} \Big(ord(b_{1..j}) \wedge \bigwedge_{i=j+1}^{n} (k_i = 0) \Big) \right]\!\!\right] \wedge \bigwedge_{j=1}^{n-1} \left[\!\!\left[\begin{array}{l} (r_{j+1} = 0) \to (b_j \le c_{j+1}) \\ \left[\begin{array}{l} k_j = n - j + 1 \\ r_j = r_{j+1} = 1 \end{array} \right] \to (k_{j+1} = n - j) \end{array} \right]\!\!\right].$$

Глава 6

Распределенные программы

6.1 Понятие распределенной программы

Распределенной программой (РП) называется список $(P_1, \ldots, P_k)$, компонентами которого являются процессы с передачей сообщений. Понятие **процесса с передачей сообщений (ППС)** определяется аналогично понятию процессного графа, с единственным отличием: метка каждого ребра представляет собой запись, которая

- либо является последовательностью из $\mathcal{A}_n^*$,
- либо получается вставкой в какую-либо последовательность из $\mathcal{A}_n^*$ одного ввода или вывода, где
  - **ввод** – это запись вида $P?x$, в которой P – имя какого-либо ППС и $x \in \mathcal{X}$, и
  - **вывод** – это запись вида $P!e$, в которой P – имя какого-либо ППС и $e \in \mathcal{E}$,

  причем ввод или вывод в метке ребра не м.б. первым действием (первым действием м.б. только проверка условия).

Если РП имеет вид $(P_1, \ldots, P_k)$, то будем предполагать, что

- для каждого из ППС P_i в этой РП каждый ввод и вывод, входящий в P_i, имеет вид $P_j?x$ или $P_j!e$, где P_j – какой-либо другой ППС, входящий в эту РП,
- если P_i и P_j – различные ППС, входящие в эту РП, то множества X_{P_i} и X_{P_j} переменных, входящих в эти ППС, не пересекаются.

В метках ребер ППС могут использоваться записи вида $P?(x_1, \ldots, x_k)$, понимаемые как последовательности действий вида

$$P?u \quad x_1 := pr_1(u) \quad \ldots \quad x_k := pr_k(u)$$

где u – новая переменная (т.е. не входящая никуда кроме вышеперечисленных действий) типа $(t(x_1), \ldots t(x_k))$, и функции pr_1, …, pr_k вычисляют компоненты кортежей, являющихся аргументами этих функций.

Понятие **выполнения** РП $P = (P_1, \ldots, P_k)$ определяется аналогично понятию выполнения параллельной программы. В каждый момент выполнения РП P определено текущее означивание $\xi \in X_P^\bullet$, где X_P – совокупность всех переменных, входящих в P, и в этот момент

- либо все ППС $P_1, \ldots, P_k$ находятся в состоянии ожидания,
- либо происходит переход в одном из ППС P_i из P по его текущему ребру, причем метка этого ребра не содержит ввода или вывода,
- либо одновременно происходят переходы в паре P_i, P_j различных ППС, входящих в P, причем метки A_i и A_j текущих ребер в P_i и P_j, по которым происходят эти переходы, удовлетворяют условию:
  - A_i и A_j имеют вид $b_i A_i'(P_j?x)A_i''$ и $b_j A_j'(P_i!e)A_j''$ соответственно, где $b_i, b_j \in \mathcal{B}$, $t(x) = t(e)$,
  - текущее означивание ξ удовлетворяет условию $(b_i \wedge b_j)^\xi = 1$,

  и после выполнения этой пары переходов текущее означивание ξ заменяется на новое текущее означивание $\xi' \stackrel{\text{def}}{=} \xi A_i' A_j'(x := e)A_i'' A_j''$.

  Выполнение данной пары переходов можно понимать как синхронную передачу сообщения e от процесса P_j процессу P_i.

Как и для параллельных программ, для РП определяются

- понятие выполнения с начальным означиванием, и
- утверждения, обозначаемые записями вида $\xi \stackrel{P}{\to} \otimes$ и $Pre \stackrel{P}{\to} Post$.

Одной из актуальных открытых проблем современной теоретической информатики является проблема построения эффективных алгоритмов верификации РП (например, на основе метода инвариантов).

6.2 Синхронное и асинхронное взаимодействие процессов с передачей сообщений

Изложенный выше механизм выполнения РП предусматривает **синхронное** взаимодействие между процессами, входящими в эту РП. При таком взаимодействии отправка и получение сообщения рассматриваются как сцепленные компоненты единого акта.

Наряду с синхронным взаимодействием процессов существует также **асинхронное** взаимодействие, при котором отправка сообщения одним процессом и получение этого сообщения другим процессом – это два разных акта. При таком взаимодействии

- процесс–отправитель помещает свое сообщение в канал связи между процессами, через который передаются сообщения, не заботясь о том, будет ли это сообщение получено тем процессом, для которого оно предназначено,
- движение сообщения по каналу от отправителя к получателю может длиться недетерминированное время, и
- процесс–получатель забирает поступившее сообщение из канала, если оно там имеется (если в канале содержится несколько поступивших сообщений, то получатель забирает первое из них), а если в канале нет сообщения, то процесс–получатель переходит в состояние ожидания.

Асинхронную передачу сообщений иногда рассматривают как более общий вид взаимодействия процессов, по сравнению с синхронной передачей, но в действительности дело обстоит наоборот:

- асинхронное взаимодействие процессов можно моделировать на основе синхронного взаимодействия,
- в то время как моделирование синхронного взаимодействия на основе асинхронного взаимодействия невозможно.

Для того, чтобы выразить асинхронный механизм передачи сообщений от процесса P_1 процессу P_2 с использованием механизма синхронного передачи, необходимо представить канал передачи сообщений между этими процессами в виде еще одного процесса *Channel*, с которым процессы P_1 и P_2 взаимодействуют синхронно. В простейшем случае *Channel* можно представить в виде идеального буфера типа FIFO (First

Input – First Output), в который P_1 помещает сообщения, и из которого P_2 принимает сообщения, а поступившие в этот канал сообщения не теряются и сохраняют правильный порядок. Но в ряде случаев канал передачи сообщений представляет собой сложную коммуникационную среду, в которой пересылаемые сообщения могут подвергаться самым разным преобразованиям, и модель этого канала должна учитывать все эти свойства. Например, в канале могут происходить

- искажения пересылаемых сообщений, или их потеря,
- переупорядочение сообщений (P_1 может послать P_2 сначала сообщение m, и затем другое сообщение m', а получатель может получить их в обратном порядке – сначала m', а затем – m, и
- дублирование пересылаемых сообщений, это возможно тогда, когда
  - процесс P_1 посылает процессу P_2 сообщение m,
  - посланное сообщение m двигается от P_1 к P_2 по сложному маршруту, и застревает в некоторой промежуточной точке (например, сервер, на котором хранилось сообщение m в процессе доставки, оказался надолго отключенным),
  - по истечении некоторого таймаута ожидания подтверждения об успешной доставке сообщения m процесс P_1 принимает решение, что сообщение m пропало в канале, и посылает это сообщение повторно,
  - затем исходное сообщение m выходит из той промежуточной точки, в которой оно застряло, после чего оно присутствует в канале уже в двух экземплярах – в виде исходного сообщения, и в виде повторно посланного дубликата.

Иногда для анализа количественных свойств анализируемых РП в модели канала важно учитывать различные количественные характеристики этого канала (например, максимальное число сообщений, которые могут одновременно содержаться в этом канале, вероятности потерь сообщений, искажения, задержки, и т.п.).

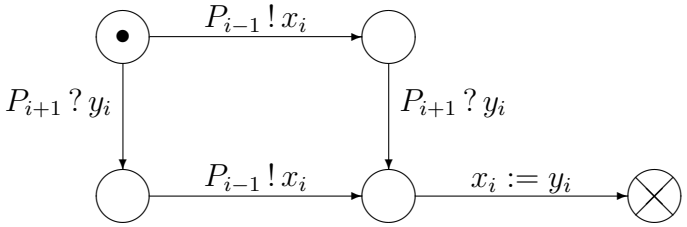
6.3 Примеры распределенных программ

В этом параграфе приводятся примеры РП и их спецификаций. Читателю предлагается самостоятельно разработать метод верификации РП, и применить его для верификации изложенных ниже РП.

6.3.1 Циклическая перестановка значений

В этом пункте описывается РП $P = (P_0, \ldots, P_{n-1})$, где $\forall\, i = 0, \ldots, n-1$ множество X_{P_i} содержит переменную x_i, и задача P заключается в том, чтобы после её завершения $\forall\, i = 0, \ldots, n-1$ в переменной x_i содержалось то значение, которое содержалось до выполнения P в переменной x_{i+1} (операции над индексами рассматриваются по модулю n).

$\forall\, i = 0, \ldots, n-1$ ППС P_i имеет следующий вид:



В описании спецификации РП P мы будем использовать новые переменные $z_0, \ldots, z_{n-1}$ (т.е. данные переменные входят только в спецификацию). Спецификация P имеет вид

$$\begin{cases} Pre = (n \geq 2) \wedge \Big(\bigwedge\limits_{i=0}^{n-1}(x_i = z_i)\Big), \\ Post = \bigwedge\limits_{i=0}^{n-1}(x_i = z_{i+1}). \end{cases}$$

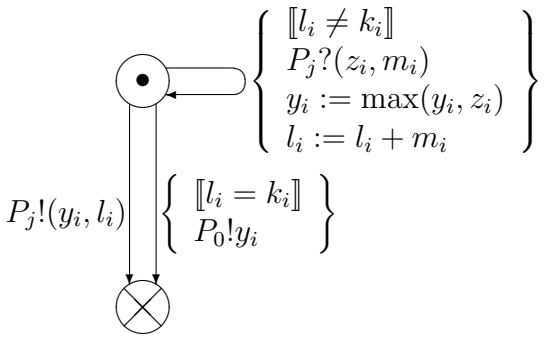
6.3.2 Распределенное вычисление максимума

В следующем примере описывается РП $P = (P_0, P_1, \ldots, P_n)$, где

- X_{P_0} содержит переменную x_0,
- $\forall\, i = 1, \ldots, n$ X_{P_i} содержит переменную x_i, значение которой не изменяется во время выполнения P, а также переменную k_i, причем значения всех переменных $k_1, \ldots, k_n$ равны числу n, и
- задача P заключается в том, чтобы после её завершения в переменной x_0 содержалось значение $\max\limits_{i=1\ldots n} x_i$.

ППС P_0 имеет две вершины – начальную ($\odot$) и терминальную ($\otimes$), и $\forall\, i = 1, \ldots, n$ P_0 содержит ребро из $\odot$ в $\otimes$ с меткой $P_i?x_0$.

$\forall\, i = 1, \ldots, n$ ППС P_i тоже имеет две вершины – начальную ($\odot$) и терминальную ($\otimes$), и схематически изображается диаграммой



в которой используются следующие обозначения:

- стрелка из $\odot$ в $\odot$ изображает $n-1$ ребер, каждое из которых имеет метку, получаемую из нарисованной рядом с этой стрелкой метки заменой j на конкретное значение из $\{1, \ldots, n\} \setminus \{i\}$,
- стрелка из $\odot$ в $\otimes$ с меткой $P_j!(y_i, l_i)$ тоже изображает $n-1$ ребер, метки которых определяются аналогичным образом.

Спецификация РП P:

$$\begin{cases} Pre = (n \geq 2) \wedge \bigwedge\limits_{i=1}^{n} \Big((y_i = x_i) \wedge (l_i = 1) \wedge (k_i = n)\Big), \\ Post = (x_0 = \max\limits_{i=1..n} x_i). \end{cases}$$

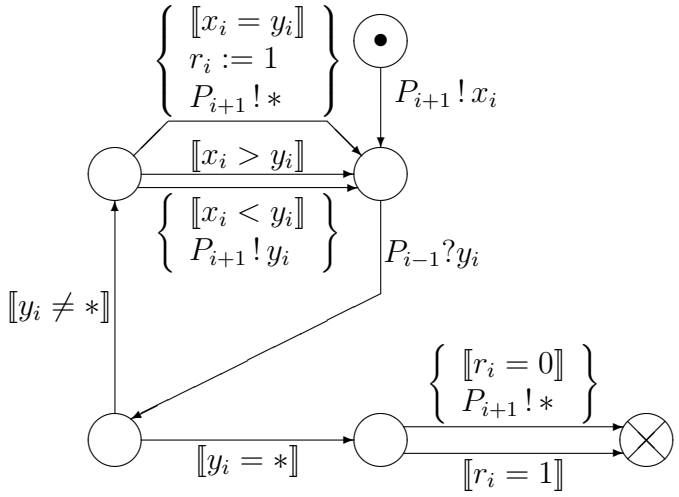
6.3.3 Избрание лидера

Третий пример РП имеет вид $P = (P_0, \ldots, P_{n-1})$, где $\forall\, i = 0, \ldots, n-1$ множество X_{P_i} содержит переменные x_i и r_i, причем значения переменных $x_0, \ldots, x_{n-1}$ различны и не изменяются в процессе выполнения P, и задача P заключается в том, чтобы после её завершения

$$\forall\, i = 0, \ldots, n-1 \quad r_i = \begin{cases} 1, & \text{если } x_i = \max\limits_{j=0..n-1} x_j, \\ 0, & \text{иначе.} \end{cases}$$

На взаимодействие между процессами $P_0, \ldots, P_{n-1}$ накладывается ограничение: $\forall\, i = 0, \ldots, n-1$ P_i может получать сообщения только от P_{i-1} и посылать сообщения только P_{i+1}, где операции над индексами процессов рассматриваются по модулю n.

$\forall\, i = 0, \ldots, n-1$ ППС P_i имеет следующий вид:



В P_i используется вспомогательная переменная y_i, где $t(x_i) = t(y_i)$, и предполагается, что $D_{t(x_i)}$ содержит выделенное значение $*$, которое отличается от всех значений, которые могут принимать $x_0, \ldots, x_{n-1}$.

Спецификация РП P:

$$\begin{cases} Pre = (n \geq 2) \wedge \bigwedge\limits_{i=0}^{n-1} \Big((r_i = 0) \wedge (x_i \neq *) \wedge \big(\bigwedge\limits_{j \neq i} (x_j \neq x_i) \big) \Big), \\ Post = \bigvee\limits_{i=0}^{n-1} \Big((r_i = 1) \wedge \big(\bigwedge\limits_{j \neq i} (r_j = 0) \wedge (x_j < x_i) \big) \Big). \end{cases}$$

Глава 7

Задачи

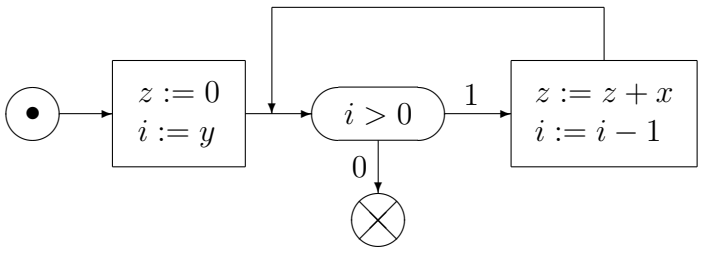
Все представленные задачи заключаются в том, чтобы доказать корректность программ, представленых в виде БС или в операторной форме. Если тип какой-либо переменной в этих программах не указан явно, то он по умолчанию равен `int`. В некоторых задачах приведено несколько программ, соответствующих одной и той же спецификации. Если какие-либо две из излагаемых ниже программ являются реализацией в виде БС и в операторной форме одного и того же алгоритма, они приводятся в одном и том же пункте.

7.1 Задачи без массивов

7.1.1 Произведение двух чисел

$$Pre : y \geq 0, \quad Post : z = xy.$$

1.



$\{z := 0;\ i := y\}$
while $i > 0$ **do** $\{z := z + x;\ i := i - 1\}$

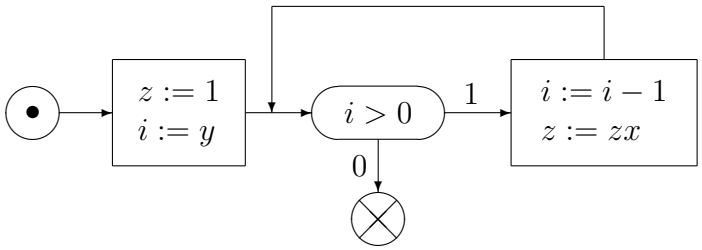
2.

$\{z := 0;\ i := x;\ j := y\}$
while $j \neq 0$ **do** $\left\{ \begin{array}{ll} \textbf{if } j\%2 = 1 & \textbf{then } z := z + i \\ & \textbf{else } \{i := 2i;\ j := j/2\} \end{array} \right\}$

7.1.2 Возведение в степень

$$Pre : y \geq 0, \quad Post : z = x^y.$$

1.



$\{z := 1;\ i := y\}$
while $i > 0$ **do** $\{i := i - 1;\ z := xz\}$

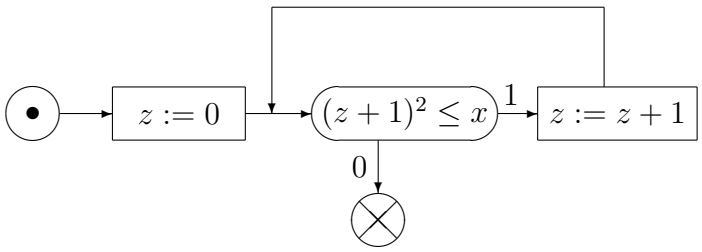
2.

$$\{\ z := 1;\ i := x;\ j := y\ \}$$
$$\textbf{while } j \neq 0 \textbf{ do} \left\{ \begin{array}{ll} \textbf{if } j\%2 = 1 & \textbf{then } \{j := j - 1;\ z := iz\} \\ & \textbf{else } \{i := i^2;\ j := j/2\} \end{array} \right\}$$

7.1.3 Извлечение квадратного корня

$$Pre : x \geq 0, \quad Post : z = \lfloor \sqrt{x} \rfloor.$$

1.



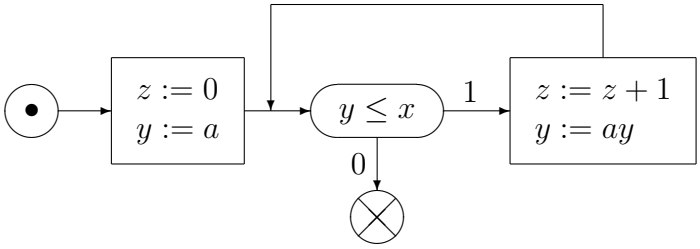
$z := 0$
while $(z + 1)^2 \leq x$ **do** $z := z + 1$

2.

$\{i := 1;\ j := 1;\ a := 0;\ b := 0;\ z := 0\}$
while $j \leq x$ **do** $\{i := 2i;\ j := 4j\}$
$$\textbf{while } i > 1 \textbf{ do} \left\{ \begin{array}{l} \{i := i/2;\ j := j/4;\ b := b/2;\ c := a + b + j\} \\ \textbf{if } c \leq x \textbf{ then } \{a := c;\ b := b + 2j;\ z := z + i\} \end{array} \right\}$$

7.1.4 Извлечение логарифма

$$Pre : x \geq 1, a \geq 2, \quad Post : z = \lfloor \log_a x \rfloor.$$



$\{z := 0;\ y := a\}$
while $y \neq x$ **do** $\{z := z + 1;\ y := ay\}$

Указание: выразить постусловие соотношением $a^z \leq x < a^{z+1}$.

7.1.5 Вычисление частного и остатка от деления целых чисел

$$Pre : (a \geq 0) \wedge (b > 0), \quad Post : (a = bq + r) \wedge (0 \leq r < b).$$

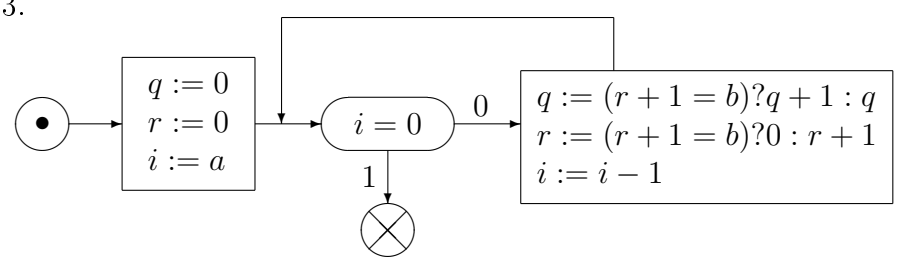
1.

$\{r := a;\ q := 0\}$
while $b \leq r$ **do** $\{r := r - b;\ q := q + 1\}$

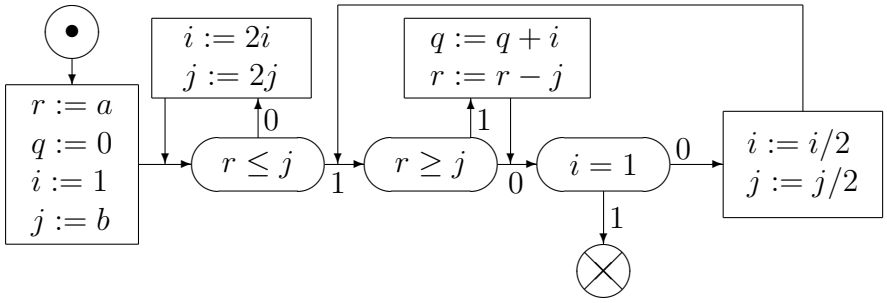
2.

$\{r := a;\ q := 0;\ i := b\}$
while $i \leq r$ **do** $i := 2i$
while $i \neq b$ **do** $\left\{ \begin{array}{l} q := 2q \\ i := i/2 \\ \textbf{if } i \leq r \textbf{ then } \{r := r - i;\ q := q + 1\} \end{array} \right\}$

3.



4. (аппаратная реализация деления целых чисел)



$\{r := a;\ q := 0;\ i := 1;\ j := b\}$
while $r > j$ **do** $\{i := 2i;\ j := 2j\}$
do $\left\{ \begin{array}{l} \textbf{if } r \ge j \textbf{ then } \{q := q + i;\ r := r - j\} \\ \textbf{if } i \ne 1 \textbf{ then } \{i := i/2;\ j := j/2\} \end{array} \right\}$ **while** $i \ne 1$

7.1.6 Наибольший общий делитель

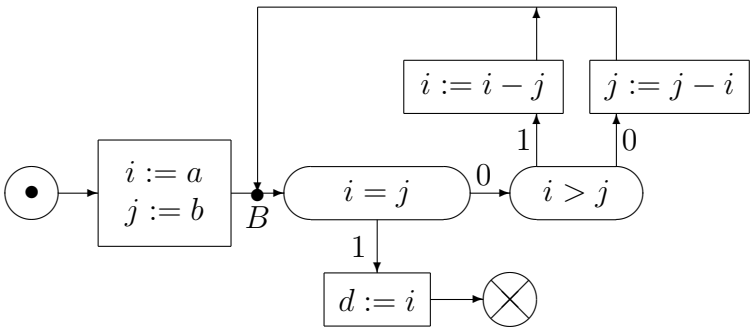
$$Pre : (a > 0) \wedge (b > 0), \quad Post : d = gcd(a, b),$$

где $d = gcd(a, b)$ – сокращенная запись формулы

$$d = \max\{x \mid \exists\, a', b' : a = xa', b = xb'\}$$

(gcd – сокращение от greatest common divisor).

1.



$\{i := a;\ j := b\}$
while $i \ne j$ **do** { **if** $i > j$ **then** $i := i - j$ **else** $j := j - i$ }
$d := i$

Указание: $\varphi_B = \left[\!\!\left[\begin{array}{l} (a > 0) \wedge (b > 0) \\ (i > 0) \wedge (j > 0) \\ gcd(i, j) = gcd(a, b) \end{array} \right]\!\!\right]$, $u_B = \max(y_1, y_2)$.

2.

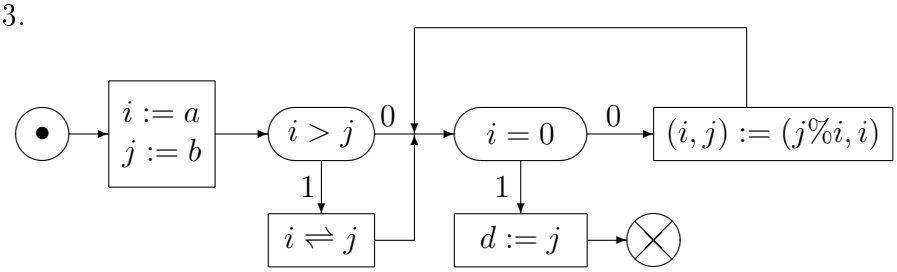
$\{i := a;\ j := b\}$

$$\textbf{while } i \neq j \textbf{ do} \left\{ \begin{array}{l} \textbf{while } i > j \textbf{ do } i := i - j \\ \textbf{while } j > i \textbf{ do } j := j - i \end{array} \right\}$$

$d := i$

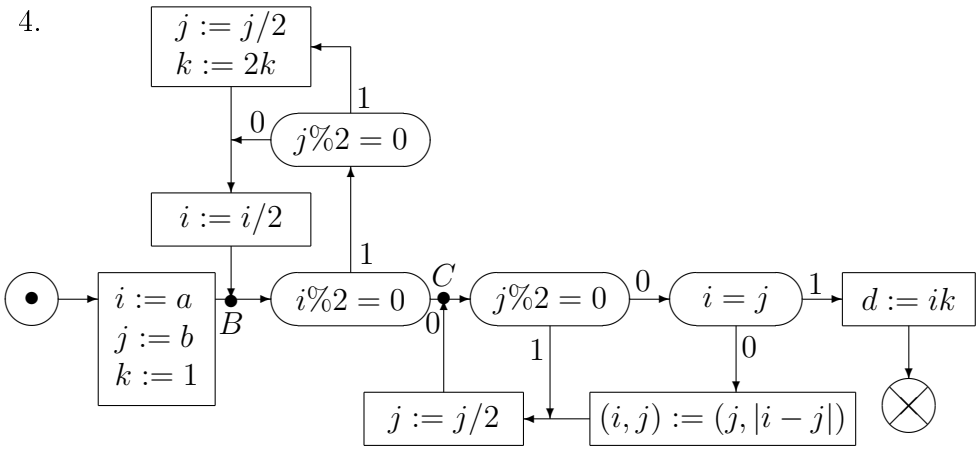
Указание: для доказательства завершаемости в качестве фундированного множества L можно взять множество $\mathbf{N}$ натуральных чисел, и в качестве u – выражение $i + j$ (для всех трех циклов).

3.



$\{i := a;\ j := b\}$
if $(i > j)$ **then** $i \rightleftharpoons j$
while $(i \neq 0)$ **do** $(i, j) := (j\%i, i)$
$d := j$

4.



$\{i := a;\ j := b;\ k := 1\}$

$$\textbf{while } i\%2 = 0 \textbf{ do} \left\{ \begin{array}{l} \textbf{if } j\%2 = 0 \textbf{ then } \{j := j/2;\ k := 2k\} \\ i := i/2 \end{array} \right\}$$

$$\textbf{while } (j\%2 = 0) \vee (i \neq j) \textbf{ do} \left\{ \begin{array}{l} \textbf{if } j\%2 = 1 \textbf{ then } (i, j) := (j, |i - j|) \\ j := j/2 \end{array} \right\}$$

$d := ik$

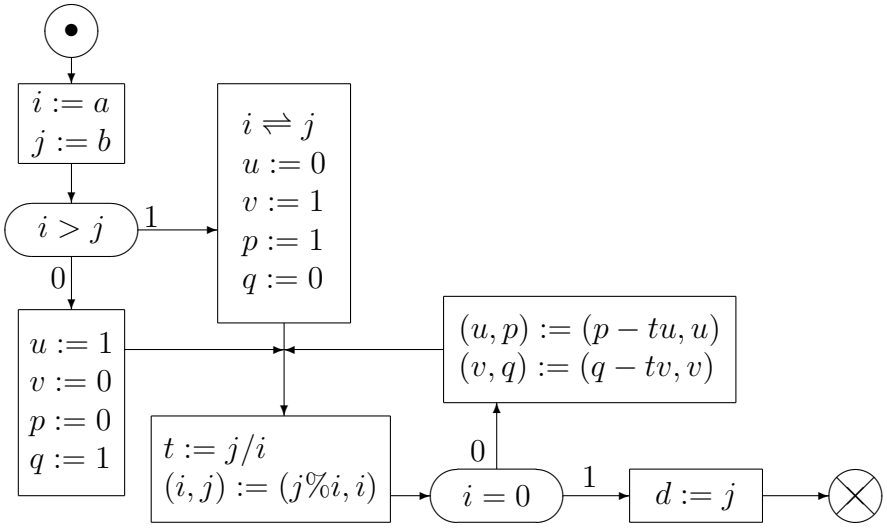
Указание: для доказательства завершаемости рассмотреть следующие инварианты:

$$\varphi_B = \left[\!\!\left[\begin{array}{l} (a>0)\wedge(b>0) \\ (i>0)\wedge(j>0) \\ gcd(i,j)k = gcd(a,b) \end{array} \right]\!\!\right], \quad \varphi_C = \varphi_B \wedge (i\%2 = 1),$$

и определить u_B и u_C следующим образом: $u_B \stackrel{\text{def}}{=} i$, $u_C \stackrel{\text{def}}{=} i + 2j$.

7.1.7 Представление наибольшего общего делителя линейной формой

$Pre : (a > 0) \wedge (b > 0), \quad Post : (d = gcd(a,b)) \wedge (d = ua + vb).$



7.1.8 Наибольший общий делитель и наименьшее общее кратное

$Pre : (a > 0) \wedge (b > 0), \quad Post : (d = gcd(a,b)) \wedge (m = scm(a,b)),$

где $m = scm(a,b)$ – сокращенная запись формулы

$$m = \min\{x \geq 0 \mid \exists\, a', b' : x = aa' = bb'\}$$

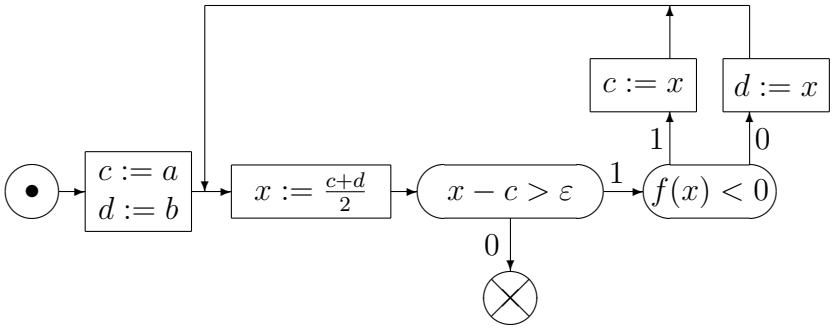
(scm – сокращение от smallest common multiple).

$$\{i := a;\ j := b;\ p := 0;\ q := b\}$$
$$\textbf{while } i \neq j \textbf{ do } \left\{ \begin{array}{l} \textbf{while } i > j \textbf{ do } (i,p) := (i-j, p+q) \\ \textbf{while } j > i \textbf{ do } (j,q) := (j-i, p+q) \end{array} \right\}$$
$$\{d := i;\ m := p+q\}$$

7.1.9 Приближенное решение уравнения

В излагаемой ниже БС все переменные имеют тип `real`, их значениями являются действительные числа.

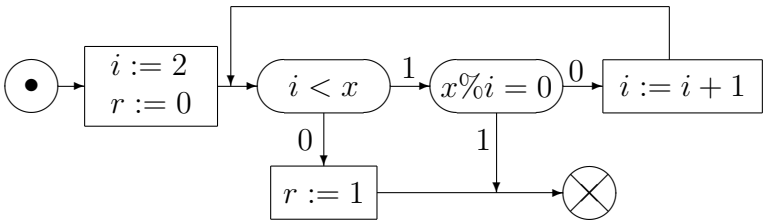
$Pre : (a < b) \wedge (f \in C[a,b]) \wedge (f(a) \le 0) \wedge (f(b) \ge 0) \wedge (\varepsilon > 0)$,
$Post : \exists y \in [a,b] : |x - y| < \varepsilon, f(y) = 0$.



$\{c := a;\ d := b\}$
while $d - c > 2\varepsilon$ **do** $\left\{ \begin{array}{l} x := (c+d)/2 \\ \textbf{if } f(x) < 0 \textbf{ then } c := x \textbf{ else } d := x \end{array} \right\}$
$x := (c+d)/2$

7.1.10 Проверка на простоту

$Pre : x \ge 2$,
$Post : r = \left\{ \begin{array}{l} 1, \text{ если } x - \text{простое, т.е. } \forall\, i = 2, \ldots, x-1 \ \ x\%i \ne 0, \\ 0, \text{ если } x \text{ не является простым числом.} \end{array} \right.$

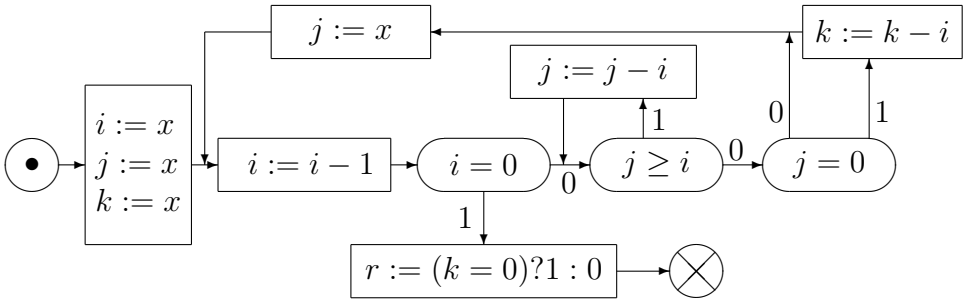


$\{i := 2;\ r := 1\}$
while $(i < x) \wedge (r = 1)$ **do** $\left\{ \begin{array}{ll} \textbf{if } x\%i = 0 & \textbf{then } r := 0 \\ & \textbf{else } i := i + 1 \end{array} \right\}$

7.1.11 Проверка, является ли число совершенным

Целое число называется совершенным, если оно равно сумме своих делителей (например, число $6 = 1 + 2 + 3$ – совершенное).

$$Pre : x > 2, \quad Post : r = \begin{cases} 1, \text{ если } x \sum\limits_{1 \le d < x, d|x} d, \\ 0, \text{ иначе.} \end{cases}$$



$\{i := x - 1;\ j := x;\ k := x\}$

while $i \neq 0$ **do** $\left\{ \begin{array}{l} \textbf{while } j \geq i \textbf{ do } j := j - i \\ \textbf{if } j = 0 \textbf{ then } k := k - i \\ j := x \\ i := i - 1 \end{array} \right\}$

if $k = 0$ **then** $r := 1$ **else** $r := 0$

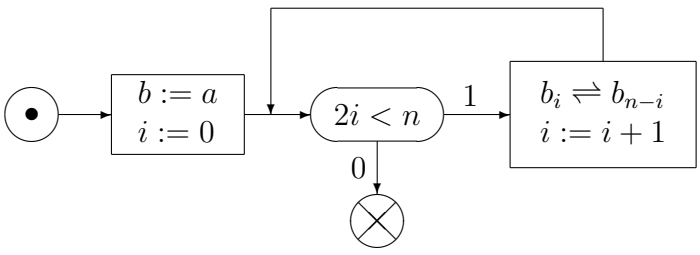
7.2 Задачи с массивами

7.2.1 Инвертирование массива

Входной массив – $a_{0..n}$, результат – массив $b_{0..n}$.

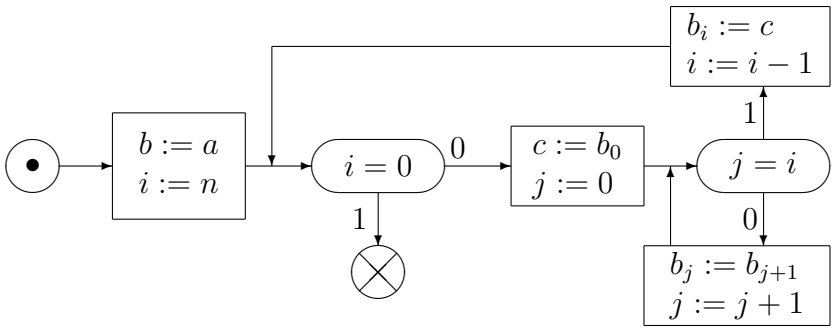
$$Pre : n \geq 0, \quad Post : \forall\, i = 0, \ldots, n \;\; b_i = a_{n-i}.$$

1.



$\{b := a;\ i := 0\}$
while $(2i < n)$ **do** $\{b_i \rightleftharpoons b_{n-i};\ i := i + 1\}$

2.

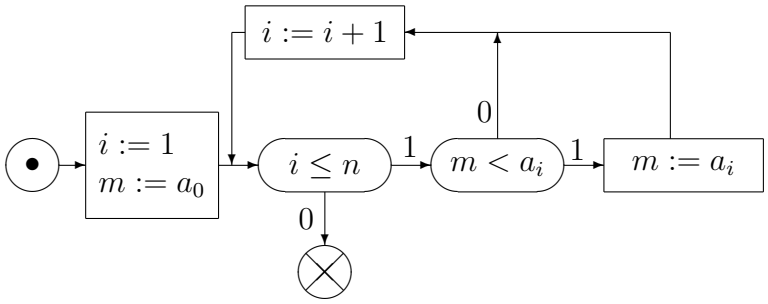


7.2.2 Минимальный элемент массива

Входной массив – $a_{0..n}$.

$$Pre : n \geq 0, \quad Post : m = \min\{a_i \mid i = 0, \ldots, n\}.$$

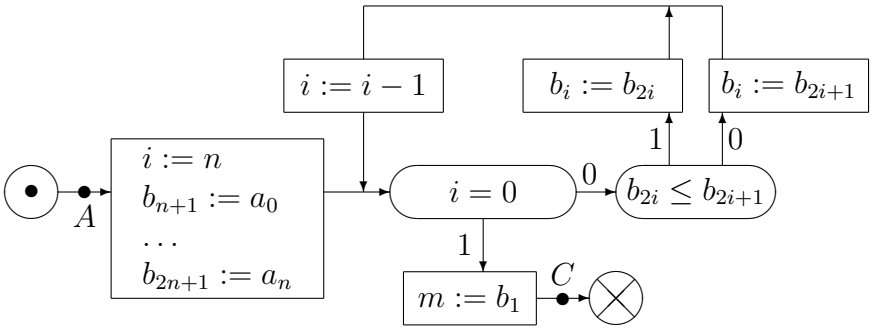
1.



$\{i := 1;\ m := a_0\}$

while $(i \leq n)$ **do** $\left\{ \begin{array}{l} \textbf{if}\ (m < a_i)\ \textbf{then}\ m := a_i \\ i := i + 1 \end{array} \right\}$

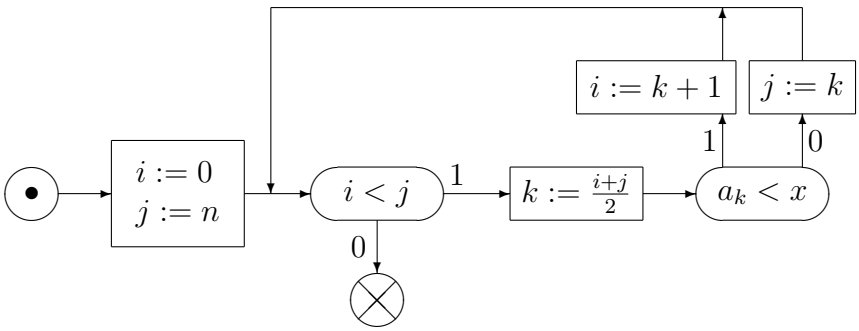
2. (в данной БС используется вспомогательный массив $b_{1..2n+1}$)



7.2.3 Двоичный поиск

Входной массив – $a_{0..n}$, требуется найти в нем положение элемента x.

$$Pre : (n \geq 0) \wedge (a_0 \leq \ldots \leq a_n) \wedge (\exists\, i : x = a_i),$$
$$Post : (0 \leq i \leq n) \wedge (x = a_i).$$



$\{i := 0;\ j := n\}$

while $(i < j)$ **do** $\left\{ \begin{array}{l} k := (i+j)/2 \\ \textbf{if } (a_k < x) \textbf{ then } i := k+1 \textbf{ else } j := k \end{array} \right\}$

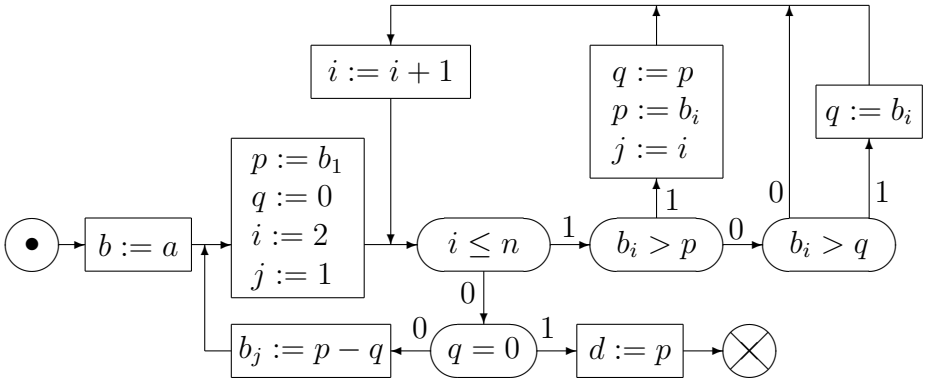
7.2.4 Наибольший общий делитель компонентов массива

Входной массив – $a_{1..n}$.

$$Pre : n \geq 1, \forall\, i\ \ a_i > 0, \quad Post : d = gcd(a_{1..n}),$$

где $d = gcd(a_{1..n})$ – сокращенная запись формулы

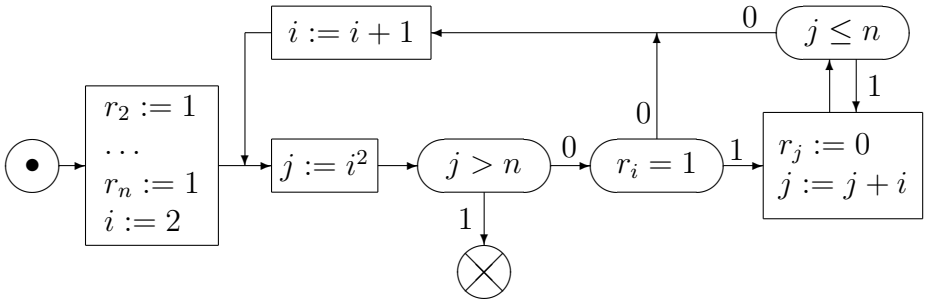
$$d = \max\{x \mid \exists\, a'_1, \ldots, a'_n : a_1 = xa'_1, \ldots, a_n = xa'_n\}.$$



7.2.5 Список простых чисел от 2 до n

Излагаемые ниже программы являются реализацией алгоритма, известного под названием “решето Эратосфена”.

$$Pre : n \geq 2,\ Post : \forall\, k = 2, \ldots, n\ \ r_k = \left\{ \begin{array}{l} 1, \text{ если } k \text{ – простое}, \\ 0, \text{ иначе}. \end{array} \right.$$



$\{r_2 := 1;\ \dots;\ r_n := 1;\ i := 2\}$
while $i^2 \le n$ **do**

$$\left\{ \begin{array}{l} \textbf{if } r_i = 1 \textbf{ then } \left\{ \begin{array}{l} j := i^2 \\ \textbf{while } j \le n \textbf{ do } \{r_j := 0;\ j := j + i\} \end{array} \right\} \\ i := i + 1 \end{array} \right\}$$

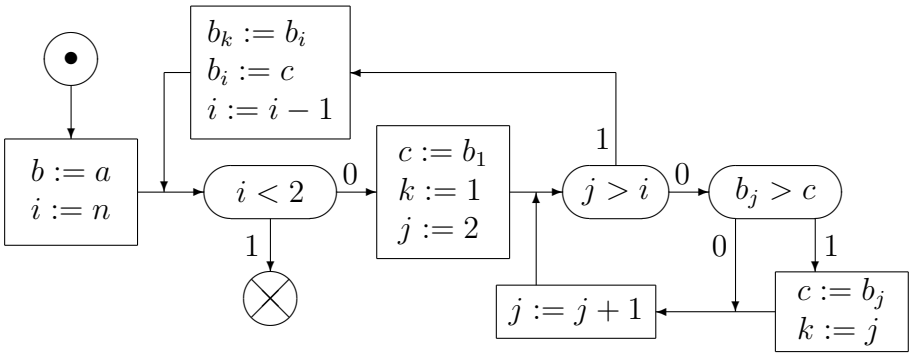
7.2.6 Сортировка массива

Входной массив – $a_{1..n}$. Результат должен быть записан в $b_{1..n}$.

$$Pre : n \ge 1,\ Post : (b = perm(a)) \wedge ord(b)$$

(утверждения вида $b = perm(a)$ и $ord(b)$ определены в пункте 1.3).

1.



2. Логика излагаемой ниже программы заключается в том, что массив $a_{1..n}$, рассматривается как дерево, в котором $\forall\, i = 2, \dots, n$ элемент $a_{i/2}$ – родитель a_i.

$b := a$
for $i = n/2$ **step** -1 **until** 2 **do** $\{p := i;\ q := n;\ P\}$
for $i = n$ **step** -1 **until** 2 **do** $\{p := 1;\ q := i;\ P;\ b_1 \rightleftharpoons b_i\}$

где $P = \left\{ \begin{array}{l} x := b_p \\ L : j := 2p \\ \textbf{if } j \le q \textbf{ then} \\ \quad \left\{ \begin{array}{l} \textbf{if } j < q \textbf{ then } \{ \textbf{ if } b_{j+1} > b_j \textbf{ then } j := j+1 \ \} \\ \textbf{if } b_j > x \textbf{ then } \{b_p := b_j;\ p := j;\ \textbf{go to } L\} \end{array} \right\} \\ b_p := x \end{array} \right\}$

Указание: рассмотреть в качестве инварианта одного из циклов формулу, выражающую следующее утверждение:

$$\forall\, i, j : 1 < j \le i \quad b_{j/2} \ge b_j, \text{ и часть } b_{i..n} \text{ отсортирована.}$$

7.2.7 Перестановка массива с заданным условием

Входной массив – $a_{0..n}$, выходной массив – $b_{0..n}$.

$$\begin{array}{l} Pre : n > 0, \\ Post : (b = perm(a)) \wedge (0 \le j < i \le n) \wedge (b_{0..i-1} \le b_{j+1..n}). \end{array}$$

$$\{b := a;\ r := b_{n/2};\ i := 0;\ j := n\}$$
$$\textbf{while } (i \ne j) \textbf{ do} \left\{ \begin{array}{l} \textbf{while } (b_i < r) \textbf{ do } i := i+1 \\ \textbf{while } (r < b_j) \textbf{ do } j := j-1 \\ \textbf{if } i \le j \textbf{ then } \{b_i \rightleftharpoons b_j;\ i := i+1;\ j := j-1\} \end{array} \right\}$$

Указание: рассмотреть в качестве инварианта большого цикла формулу $I = I_i \wedge I_j$, где $\left\{ \begin{array}{l} I_i = (0 \le i) \wedge (b_{0..i-1} \le r), \\ I_j = (j \le n) \wedge (b_{j+1..n} \ge r). \end{array} \right.$

7.2.8 Перестановка массива в заданном порядке

Входной массив – $a_{0..n}$, выходной массив – $b_{0..n}$.

$$\begin{array}{l} Pre : n \ge 0, \quad f \text{ – биекция на } \{0, \ldots, n\}, \\ Post : (b = perm(a)) \wedge (\forall\, i = 0, \ldots, n \ \ b_i = a_{f(i)}). \end{array}$$

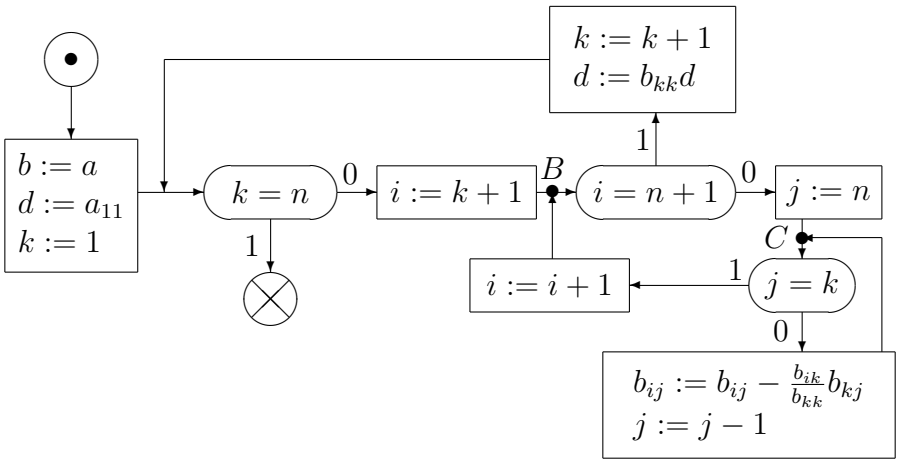
$$\begin{array}{l} b := a \\ \textbf{for } i := 0 \textbf{ step } 1 \textbf{ until } n \textbf{ do} \\ \qquad \left\{ \begin{array}{l} j := f(i) \\ \textbf{while } j < i \textbf{ do } j := f(j) \\ b_i \rightleftharpoons b_j \end{array} \right\} \end{array}$$

7.2.9 Символьное вычисление определителя матрицы

Входная матрица – $a_{1..n,1..n}$, её компоненты – независимые переменные $x_{11}, \ldots x_{nn}$. Все операции (сложение, вычитание, умножение и деление) в излагаемой ниже БС выполняются в поле дробей $\mathbf{R}(x_{11}, \ldots x_{nn})$. Программа вычисляет определитель $det(a)$ входной матрицы путем приведения её в диагональному виду и перемножения элементов на главной диагонали.

$$Pre : n \geq 1, \quad Post : d = det(a) \stackrel{\text{def}}{=} \sum_{\sigma \in S_n} (-1)^\sigma x_{1\sigma(1)} \ldots x_{n\sigma(n)},$$

где S_n – группа всех перестановок на множестве $\{1, \ldots, n\}, \forall \sigma \in S_n \ \ (-1)^\sigma$ – четность перестановки σ.



Указание: при обосновании завершаемости

- доказать, что следующие формулы являются инвариантами в соответствующих точках:
  - $\varphi_B : (1 \leq k \leq n-1) \wedge (2 \leq i \leq n+1)$
  - $\varphi_C : (1 \leq k \leq n-1) \wedge (2 \leq i \leq n) \wedge (k \leq j \leq n)$
- определить N, L и $\{u_n \mid n \in N\}$ следующим образом:
  - $N \stackrel{\text{def}}{=} \{B, C\}$,
  - $L \stackrel{\text{def}}{=} \mathbf{N} \times \mathbf{N} \times \mathbf{N}$ (множество троек натуральных чисел с лексикографическим порядком), и
  - $u_B \stackrel{\text{def}}{=} (n-1-k, n+1-i, n+1)$, $u_C \stackrel{\text{def}}{=} (n-1-k, n+1-i, j)$.

7.3 Верификация параллельных и распределенных программ

1. Верифицировать все параллельные и распределенные программы из пунктов 5.3 и 6.3.

2. Модифицировать параллельную программу сортировки (п. 5.3.3), в предположении что размер сортируемого массива м.б. произвольным, а число доступных процессоров является фиксированным. Верифицировать эту модифицированную программу.

3. Модифицировать РП избрания лидера, изложенный в пункте 6.3.3, в предположении что среди элементов $x_0, \ldots, x_{n-1}$ м.б. совпадающие, и верифицировать эту модифицированную РП.

4. Написать параллельные программы, являющиеся реализацией параллельных алгоритмов из книги [26], сформулировать их спецификации, и верифицировать эти программы относительно сформулированных спецификаций.

5. Написать распределенные программы, являющиеся реализацией распределенных алгоритмов из книги [27], сформулировать их спецификации, и верифицировать эти программы относительно сформулированных спецификаций.

7.4 Другие задачи

В этом параграфе требуется составить программы в виде БС или в операторной форме, предназначенные для решения излагаемых ниже задач, и верифицировать эти программы. Для каждой из этих программ мы указываем требования, которым она должна удовлетворять.

1. Программа получает на вход массив a, результат равен 1, если a упорядочен (т.е. верно $ord(a)$), и 0, иначе.

2. Программа получает на вход пару массивов a и b, результат равен 1, если a содержит подмассив, равный b, и 0, иначе.

3. Программа получает на вход пару массивов a и b, результат равен 1, если каждый элемент массива a входит также и в b, и 0, иначе.

4. Программа получает на вход массив $a_{1..n}$, элементами которого являются положительные натуральные числа, и вычисляет представление наиболелшего общего делителя $gcd(a)$ элементов массива a в виде линейной формы, т.е. результатом работы программы является массив $b_{1..n}$, удовлетворяющий условию $\sum_{i=1}^{n} a_i b_i = gcd(a)$.

5. Программа получает на вход массив a, результатом является массив, получаемый из a удалением копий тех элементов, которые входят в a более одного раза.

6. Программа получает на вход массив, результатом должен быть отсортированный входной массив (рассмотреть алгоритмы сортировки вставкой, слиянием, а также быструю сортировку Хоара).

7. Программа получает на вход массив a, результатом является пара индексов $i \leq j$, такая, что $a_{i..j}$ – максимальный по размеру подмассив массива a, все элементы которого одинаковы.

8. Программа получает на вход массив a, результатом является массив b, получаемый перестановкой элементов a, и такой, что сначала в b идут элементы, имеющие наименьшее число вхождений в a, затем - имеющее большее число вхождений, и т.д.

9. Программа получает на вход массив a, результатом является пара индексов $i \leq j$, такая, что $a_{i..j}$ – максимальный по размеру неубывающий подмассив массива a (т.е. $a_i \leq \ldots \leq a_j$).

10. Программа получает на вход массив a, результатом является элемент, имеющий наибольшее число вхождений в a.

11. Программа получает на вход пару упорядоченных массивов a и b, результатом должен быть упорядоченный массив, множество компонентов которого является пересечением (объединением, разностью) множеств компонентов массивов a и b.

12. Программа получает на вход матрицу, элементы которой являются действительными числами. Результатом должен быть определитель этой матрицы, вычисленный путем приведения исходной матрицы к ступенчатому виду.

13. Программа получает на вход матрицу $a_{1..n,1..n}$, и $\forall i, j = 1, \ldots, n$ коэффициент a_{ij} представляет собой расстояние от вершины с номером i до вершины с номером j некоторого графа, вершины которого занумерованы числами $1, \ldots, n$. Результатом должна быть

длина минимального пути из вершины с номером 1 в вершину с номером n, причем при движении по этому пути номера вершин должны возрастать.

14. Программа получает на вход целое число n, результатом является разложение n на простые множители, т.е. списки простых чисел $p_1, \ldots, p_k$ и натуральных чисел $i_1, \ldots, i_k$, такие что $n = p_1^{i_1} \ldots p_k^{i_k}$.

15. Программа получает на вход ориентированный граф, результат – список сильно связных компонентов этого графа. Программа должна представлять собой реализацию алгоритма Тарьяна [25].

Глава 8

Заключение

В настоящем тексте изложено описание метода инвариантов для верификации различных классов программ, и приведено несколько иллюстраций применения этого метода.

Наиболее актуальная проблема, связанная с применением метода инвариантов для верификации программ, заключается в автоматизации построения инвариантов (или в автоматизации построения доказательства их отсутствия в том случае, когда верифицируемая программа не удовлетворяет своей спецификации). В общем случае эта проблема алгоритмически неразрешима, поэтому данную проблему целесообразно рассматривать в следующей постановке: разработать средства интерактивного построения требуемых инвариантов, которые бы давали программисту рекомендации по поводу того, какой вид могли бы иметь инварианты, т.е. предлагали ему некоторые шаблоны инвариантов, которые он бы уже самостоятельно конкретизировал до формул, обладающих необходимыми свойствами. Некоторые продвижения в решении данной проблемы можно найти в работах [12]–[24].

Другим направлением исследований в этой области является распространение данного метода на другие классы программ, в том числе на

- программы с потенциально неограниченным количеством взаимодействующих процессов (например, MPI–программы),
- программы, при выполнении которых могут порождаться новые процессы (например, аналогичные тем, которые порождаются функцией `fork` в ОС UNIX),
- сети Петри, программы с использованием нейронных сетей, функциональные и логические программы, и т.п.

Литература

[1] Лекции лауреатов премии Тьюринга. - М.: Мир, 1993.

[2] Страничка Р. Флойда в Википедии.
`https://ru.wikipedia.org/wiki/Флойд,_Роберт`

[3] Floyd R. W. Assigning meanings to programs. In J. T. Schwartz, editor, Proc. Symp. Appl. Math., volume 19 of Mathematical Aspects of Computer Science, pages 19-32, Providence, R.I., 1967.

[4] Hoare C. A. R. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576580, October 1969.

[5] Мендельсон Э. Введение в математическую логику. - М.: Наука, 1984.

[6] Андерсон Р. Доказательство правильности программ. - М.: Мир, 1982.

[7] Абрамов С. А. Элементы анализа программ. Частичные функции на множестве состояний. - М.: Наука, 1986.

[8] Непомнящий В. А., Рякин О. М. Прикладные методы верификации программ. - М.: Радио и связь, 1988.

[9] Francez N. Verication of programs. -Addison-Wesley Publishers Ltd., 1992.

[10] Loeckx J., and Sieber K. The Foundations of Program Verication. Wiley. -Teubner, Stuttgart, 1984.

[11] Manna Z. Mathematical theory of computation. -McGraw-Hill, 1974.

[12] Bjorner N., Browne A., and Manna Z. Automatic generation of invariants and intermediate assertions. Theoretical Computer Science, Volume 173, Issue 1, 1997, P. 49–87.

[13] S. Bensalem, Y. Lakhnech, H. Saidi. Powerful techniques for the automatic generation of invariants. Proc. 8th Internat. Conf. on Computer Aided Verification, Lecture Notes in Computer Science, Vol. 1102, Springer, Berlin (1996), pp. 323-335.

[14] R. Chadha, D.A. Plaisted. On the mechanical derivation of loop invariants. J. Symbol. Comput., 15 (5) (1993), pp. 705-744

[15] P. Cousot, R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fix points. 4th ACM Symp. Principles of Programming Languages, ACM Press, New York (1977), pp. 238-252

[16] P. Cousot, N. Halbwachs. Automatic discovery of linear restraints among the variables of a program. 5th ACM Symp. Principles of Programming Languages (1978), pp. 84-97

[17] D. Dill, H. Wong-Toi. Verification of real-time systems by successive over and under approximation. Proc. 7th Internat. Conf. on Computer Aided Verification. Lecture Notes in Computer Science, Springer, Berlin (1995), pp. 409-422.

[18] S.M. German, B. Wegbreit. A synthesizer of inductive assertions. IEEE Trans. Software Eng., 1 (1975), pp. 68-75.

[19] N. Halbwachs, P. Raymond, Y.-E. Proy. Verification of linear hybrid systems by means of convex approximations. 1st Internat. Static Analysis Symp., Lecture Notes in Computer Science, Vol. 864, Springer, Berlin (1994), pp. 223-237.

[20] S. Bensalem, M. Bozga, J.-C. Fernandez, L. Ghirvu, and Y. Lakhnech. A transformational approach for generating non-linear invariants. In SAS, 2000, pp. 58–74.

[21] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using grobner bases. 2004.

[22] A. Tiwari, H. Rueb, H. Saidi, and N. Shankar. A technique for invariant generation. In TACAS 2001 – Tools and Algorithms for the Construction and Analysis of Systems, ser. LNCS, vol. 2031. Genova, Italy: Springer-Verlag, Apr. 2001, pp. 113–127.

[23] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of

likely invariants. Science of Computer Programming, vol. 69, no. 1–3, pp. 35–45, Dec. 2007.

[24] A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In Computer Aided Verification. Springer, 2009, pp. 634–640.

[25] https://ru.wikipedia.org/wiki/Алгоритм_Тарьяна

[26] S. G. Akl. The Design and Analysis of Parallel Algorithms. Prentice-Hall, 1989.

[27] Тель Ж. Введение в распределенные алгоритмы. М.: МЦНМО, 2009.