электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.421

Теоретические основания синтеза элементарных инструкций императивных языков программирования на основе множеств присваивания

***Минашин Г.А.**, студент*
кафедра «Системы обработки информации»
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана

Научный руководитель: Гапанюк Ю. Е., к. т. н., доцент
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
chernen@bmstu.ru

1 Введение

Уже продолжительное время с начала интенсивного развития электронно-вычислительной техники исследуются и разрабатываются методы автоматизированного и автоматического синтеза программного обеспечения (далее – ПО) [5]. При этом речь идёт не об инструментах, облегчающих работу программиста, как например, системы автодополнения современных интегрированных сред разработки, системы помощи при реорганизации кода, системы отслеживания ошибок. В этой статье имеются в виду подходы, которые позволяют на некоторых стадиях разработки и внедрения ПО полностью возложить обязанности программистов на плечи компьютеров. Очевидные выгоды таких приёмов заключаются в предоставлении людям возможности сосредоточиться на творческих и более абстрактных вопросах [3], решение которых пока что не может быть автоматизировано.

Синтез программного кода должен проводиться на основе некоторых представлений о функциях разрабатываемой системы. Здесь, как и во многих других работах, такие представления формализуются в виде спецификаций функций, выраженных терминами логики второго порядка (либо только первого порядка). В базовом виде спецификация представляет собой тройку вида $\{P\}C\{Q\}$, где C – программа, работа которой специфицируется, P – предусловие, Q – постусловие.

Предусловие и постусловия являются предикатами относительно переменных программы. С помощью предусловия записывается, как различные переменные программы связаны друг с другом до начала работы. Объект, вызывающий программу, должен гарантировать выполнимость предусловия. С помощью постусловия записывается

состояние, в котором окажутся переменные программы после её окончания. Если предусловие истинно, то программа должна гарантировать выполнимость постусловия.

Например, спецификация $\{b \neq 0\} x \leftarrow a / b \{x = a / b\}$, описывающая программу, состоящую из единственной инструкции присваивания $x \leftarrow a / b$, сообщает следующее: в любом случае, когда переменная b не равна 0, выполнение программы приводит к тому, что становится истинным выражение $x = a / b$.

В данной работе рассматривается вопрос связи между элементарными инструкциями императивного языка программирования и спецификациями, выраженными в терминах логики первого порядка (без квантификации сложных выражений), устанавливающейся с помощью механизма множеств присваивания.

2 Исследуемые объекты

Будем использовать в дальнейшем два фундаментальных множества: множество истинностных значений («истина» и «ложь») и множество действительных чисел. Рассмотрим некоторое постусловие, заданное программистом. Оно содержит переменные программы и выражения, результат вычисления которых принадлежит двум упомянутым фундаментальным множествам.

На верхнем уровне постусловие должно представлять выражение булевого (истинностного) типа. Из соображений упрощения с одной стороны и обобщения результатов с другой будем считать, что переменные, с которыми работает программа, принадлежат некоторому множеству, на основе которого построено поле вида $Field = (Type, add, mul, sub, div, 0, 1)$, где *add* и *mul* – две основные двухместные операции, *sub* и *div* – одноместные операции взятия обратного элемента, а 0 и 1 – нейтральные элементы.

В поле выполняются следующие основные тождества [1].

1. Ассоциативность сложения: *add(a, add(b, c)) = add(add(a, b), c)*.
2. Коммутативность сложения: *add(a, b) = add(b, a)*.
3. Нейтральность нуля относительно сложения*: add(a,* 0*) = a*.
4. Для каждого элемента поля a существует элемент *sub(a) такой, что add(a, sub(a))* = 0.
5. Ассоциативность умножения: *mul(a, mul(b, c)) = mul(mul(a, b), c)*.
6. Коммутативность умножения: *mul(a, b) = mul(b, a)*.
7. Нейтральность единицы относительно умножения: *mul(a,* 1*) = a*.

8. Для каждого отличного от 0 элемента поля *a* существует элемент *div(a)* такой, что *mul(a, div(a))* = 0.
9. Дистрибутивность умножения относительно сложения: *mul(a, add(b, c))* = *mul(add(a, b), add(a, c))*.

Поле представляет особый интерес для исследования в данном случае, так как в его терминах могут быть выражены и решены многие уравнения, возникающие в практике программирования.

Кроме этого условимся, что в поле, к которому принадлежат переменные и операторы программы, установлен порядок так, что относительно каждых двух его элементов *a* и *b* можно сказать, что $a = b$, либо $a < b$, либо $b < a$. Далее для удобства эти бинарные отношения будем записывать с помощью префиксной нотации: *eq(a, b)* и *lt(a, b)*.

Заметим, что условие, характеризующее состояние переменных программы, должно являться некоторой системой только что рассмотренных бинарных отношений. Эти отношения могут быть связаны операторами булевой логики: дизъюнкция, конъюнкция и отрицание (в префиксной записи соответственно: *or(a, b)*, *and(a, b)*, *not(a)*).

При этом отрицание можно полностью исключить из формул, пользуясь законами де Моргана: *not(and(a, b)) = or(not(a), not(b))* и *not(or(a, b)) = and(not(a), not(b))*. «Продавив» таким образом отрицание к терминальным элементам формулы, воспользуемся соображениями *not(eq(a, b)) = or(lt(a, b), lt(b, a))* и *not(lt(a, b)) = or(eq(a, b), lt(b, a))*. Таким образом, были рассмотрены все возможные формулы, включающие отрицание.

Для последующего рассмотрения введём понятие нормализованной формулы. Будем считать представление формулы нормализованным, если оно приведено к виду $b_{1.1}b_{1.2}\ldots b_{1.n1} \vee b_{2.1}b_{2.2}\ldots b_{2.n2} \vee \ldots \vee b_{m.1}b_{m.2}\ldots b_{m.nm}$, где $b_{i.j} = b_{i,j}(\vec{x})$ – выражение вида $eq(e(\vec{x}),0)$ или $lt(e(\vec{x}),0)$, в котором $e(\vec{x})$ – некоторый многочлен от переменных программы.

Наконец, опишем, что мы понимаем под инструкцией. Инструкция императивного языка программирования – команда, выполнение которой, возможно, меняет значения переменных программы установленным образом. Базовая элементарная инструкция любого императивного языка программирования – это присваивание вида $x_1, x_2, \ldots, x_n \leftarrow e_1(\vec{x}), e_2(\vec{x}), \ldots, e_n(\vec{x})$. Такая запись обозначает следующую последовательность действий: 1) производится вычисление значений всех выражений $e_i(\vec{x})$, 2) переменным программы присваиваются новые значения. Это означает, что

вычисление новых значений выполняется до проведения хотя бы одного присваивания.

Итак, мы считаем, что гипотетическая программа оперирует переменными, функциями и значениями, относящимися либо к одному из двух фундаментальных множеств, либо к известному нам полю. Выражения логики первого порядка с участием переменных такой программы могут быть представлены в определённой нормальной форме. Перейдём к рассмотрению особенностей конструирования простейших программ на основе спецификации.

3 Множества присваивания

Множество присваивания – абстракция вида $AR[f(\vec{x})] = (Set_1, Set_2, \ldots, Set_n)$, которая означает, что переменной x_i следует присвоить какое-либо значение из множества Set_i для того, чтобы выполнялась булевая функция $f(\vec{x})$. Присваивание должно выполниться для всех переменных. Если значение какой-то переменной нужно оставить прежним, это показывается с помощью $Set_i = \{x_i\}$.

Существуют два крайних случая значения некоторого Set_i. Во-первых, может быть, что $Set_i = Type$. Тогда ясно, что переменная x_i может иметь любое значение. Во-вторых, может быть, что $Set_i = \emptyset$. Таким способом будем обозначать ситуацию сбоя: если все присваивания должны быть выполнены одновременно, а одно из множеств пусто, то выполнить присваивание не представляется возможным; подобные ситуации будут возникать, когда поиск подходящих множеств будет заходить в тупик.

Как было установлено выше, выражения относительно переменных программы, принадлежащих известному полю, могут быть построены с помощью двух базовых операций – *eq(a, b)* и *lt(a, b)*, и с помощью двух операций-связок – *or(a, b)* и *and(a, b)*. Чтобы учесть их влияние на множества, введём две дополнительные операции: 1) выбор, обозначаемый символом $\oplus$, и 2) композиция, обозначаемая символом $\otimes$.

Выбор означает, что в качестве результирующего множества нужно выбрать какое-либо одно из данных. Композиция определяется, как покомпонентное пересечение множеств:

$$(Set_1^{\,1}, Set_2^{\,1}, \ldots, Set_n^{\,1}) \otimes (Set_1^{\,2}, Set_2^{\,2}, \ldots, Set_n^{\,2}) = (Set_1^{\,1} \cap Set_1^{\,2}, Set_2^{\,1} \cap Set_2^{\,2}, \ldots, Set_n^{\,1} \cap Set_n^{\,2}).$$

Далее будет показаны причины, по которым этим операторам даётся именно такое значение.

Рассмотрим, какие множества производят базовые операции: $AR[lt(x_1, x_2)] = ((-\infty; x_2); x_2) \oplus (x_1; (x_1; +\infty))$, $AR[eq(x_1, x_2)] = (x_1; x_1) \oplus (x_2; x_2)$. Логические

связки определяются по следующим правилам: $AR[or(x_1, x_2)] = AR[x_1] \oplus AR[x_2]$ и $AR[and(x_1, x_2)] = AR[x_1] \otimes AR[x_2]$.

Учитывая символьный характер множеств присваивания, эти правила нужно определять осторожно. Например, если бы выбор двух множеств присваивания сводился к покомпонентной дизъюнкции (объединению), могло бы возникнуть следующее противоречие.

$$AR[lt(x_1, x_2)] = ((-\infty; x_2); x_2) \oplus (x_1; (x_1; +\infty)) = ((-\infty; x_2) \cup x_1; x_2 \cup (x_1; +\infty)) \supseteq (x_1; x_2),$$

то есть для того, чтобы установить состояние $lt(x_1, x_2)$ переменные можно оставить неизменными. Это, конечно же, невозможно в случае, если только условие $lt(x_1, x_2)$ уже не выполнялось до этого.

3.1 Представление множеств

Как только на этапе проектирования вводятся множества, как объект обработки, встаёт вопрос о том, с помощью каких примитивов представлять эти множества в программном обеспечении [4]. Этот вопрос не является ключевым в данном случае, однако было бы неплохо фиксировать какой-либо способ представления множеств сейчас, чтобы знать, какие возможные ограничения на нас он наложит.

Пусть множество присваивания состоит из интервалов вида $\{f_1(\vec{x}); f_2(\vec{x})\}$, где вместо фигурных скобок в зависимости от того, включительная данная граница или нет, подставляются квадратные или круглые скобки соответственно. Например, интервал $[f_1(\vec{x}); f_2(\vec{x}))$ эквивалентен множеству $\{y : f_1(\vec{x}) \le y < f_2(\vec{x})\}$.

В целях удобства в поле *Field* выберем какой-нибудь опорный элемент *a*, являющийся наиболее «простым» для восприятия. Например, для большинства расчётов с действительными числами таким элементом будет являться $a = 0$. Для измерения близости элементов к опорному установим некоторую меру $d(x, y)$.

Если какой-нибудь интервал из рассматриваемого Set_i содержит опорный элемент *a*, то его и следует выбрать в качестве правой части оператора присваивания для переменной x_i. Иначе нужно выбрать интервал, ближайший к *a* и выбрать некоторый элемент из этого интервала. Если рассматривается полубесконечный интервал, и $f_1(\vec{x})$ (для определённости) является индикатором «минус бесконечности», можно взять элемент $(f_2(\vec{x})$. Аналогично в случае «плюс бесконечности». Если обе границы конечны, можно выбрать, например, элемент $(f_1(\vec{x}) + f_2(\vec{x}))/2$. Литерал «2», разумеется, является заимствованным из терминологии действительных чисел и может быть представлен в

некотором произвольном поле, как (1 + 1).

Кроме этого встаёт задача представления набора интервалов, что и будет являться множеством. В качестве такого возможного представления можно взять следующий механизм. Выберем какой-либо из интервалов множества начальным и зададим правило, по которому можно перейти к следующему или предыдущему интервалу относительно данного. Интервал считается следующим, если его левая граница больше, чем правая граница данного интервала (или границы равны, но при этом левая граница следующего интервала исключающая, а правая граница данного – включающая, либо наоборот). Аналогично определяется предыдущий интервал.

Таким образом, имея сведения о некотором базовом интервале, теоретически можно достигнуть любого другого интервала множества. Этот способ вносит ограничения на множества, которыми можно пользоваться в работе: подходят только такие, для которых можно (а лучше – просто) задать правила перехода по цепочке интервалов вперёд и назад. Ещё такой способ подразумевает такую крайне неэффективную компьютерную операцию, как перебор. Тем не менее, этот способ оправдан при необходимости получения некоторого значения из множества, представленного в виде набора интервалов и тем более при введении теоретико-множественных операций (пересечения, объединения, исключения).

Очевидно, можно установить и другие методы выбора некоторого значения из интервала, но это уже составляет другой вопрос; пока что мы показали, что такие методы существуют, и показали, какие ограничения могут возникать в связи с их спецификой.

3.2 Дальнейшая модификация

Модифицируем определение множеств присваиваний, расширив его до правил присваиваний следующим образом. При рассмотрении любой булевой функции $f(\vec{x})$ никаких операций над переменными программы проводить не требуется, если они уже удовлетворяют $f(\vec{x})$. Таким образом, правила присваивания для элементарных отношений следует формулировать следующим образом:

$$AR[lt(x_1, x_2)] = ((-\infty; x_2); x_2) \oplus (x_1; (x_1; +\infty)) \oplus lt(x_1, x_2),$$

$$AR[eq(x_1, x_2)] = (x_1; x_1) \oplus (x_2; x_2) \oplus eq(x_1, x_2).$$

Здесь добавленные функции означают, что при их выполнении присваивание производить не требуется, так как используется оператор выбора. Использование оператора композиции будет означать, что требуется и выполнение всех указанных

булевых функций, и выполнение всех указанных присваиваний.

При построении более сложных правил присваивания следует руководствоваться следующими свойствами введённых операторов (они аналогичны свойствам сложения и умножения и следуют из свойств теоретико-множественных операций).

1. Коммутативность: $AR_1 \oplus AR_2 = AR_2 \oplus AR_1$ и $AR_1 \otimes AR_2 = AR_2 \otimes AR_1$.

2. Ассоциативность:

$$(AR_1 \oplus AR_2) \oplus AR_3 = AR_1 \oplus (AR_2 \oplus AR_3),$$

$$(AR_1 \otimes AR_2) \otimes AR_3 = AR_1 \otimes (AR_2 \otimes AR_3).$$

3. $AR_1 \otimes (AR_2 \oplus AR_3) = AR_1 \otimes AR_2 \oplus AR_1 \otimes AR_3)$.

Используя последовательные преобразования и раскрытия скобок можно привести правило присваивания к виду $AR_1 \otimes AR_2 \otimes \ldots \oplus AR_{n1+1} \otimes AR_{n1+2} \otimes \ldots$, где каждое правило AR_i является либо простейшим множеством присваивания вида $(Set_1, Set_2, \ldots, Set_n)$, либо условием вида $f(\vec{x})$.

Для дальнейшего упрощения рассмотрим дополнительные свойства.

1. Выбор условий: $f_1(\vec{x}) \oplus f_2(\vec{x}) = or(f_1(\vec{x}), f_2(\vec{x}))$.

2. Композиция условий: $f_1(\vec{x}) \otimes f_2(\vec{x}) = and(f_1(\vec{x}), f_2(\vec{x}))$.

Учитывая эти свойства, можно рассматривать правила присваивания вида $AR_1{}^C \otimes AR_1{}^A \oplus \ldots \oplus AR_2{}^C \otimes AR_2{}^A \otimes \ldots$, где AR_i^C – условие вида $f(\vec{x})$, а AR_i^A – простейшее множество присваивания вида $(Set_1, Set_2, \ldots, Set_n)$.

Такое правило можно интерпретировать следующим образом. Пусть сначала *i = 1*. Если во время выполнения программы истинно AR_i^C, то нужно установить переменным программы значения из диапазона AR_i^A. Иначе следует перейти к рассмотрению правила $AR_{i+1}{}^C \otimes AR_{i+1}{}^A$. В случае исчерпания всех доступных правил программа завершается с ошибкой (происходит «авост» – аварийная остановка), либо выполняется специально заготовленная на этот случай инструкция.

С помощью рассмотренного аппарата можно моделировать и ситуации, когда изменение переменных программы не требуется (для этого нужно воспользоваться, например, множеством, в котором $Set_i = \{x_i\}$). Вообще данный инструмент можно интерпретировать, как аналог набора защищённых инструкций или условный оператор: правила AR_i^C становятся условиями срабатывания *i*-го набора команд, а AR_i^A – инструкциями, которые исполняются в случае истинности этого условия.

Для примера рассмотрим постусловие $lt(x,0)\otimes\{x,1\}\oplus lt(x,10)\otimes\{x,2\}$. Его можно представить, как состоящее из двух компонентов, каждый из которых включает «условие» и соответствующее «действие»: 1) $lt(x,0)\otimes\{x,1\}$ и 2) $lt(x,10)\otimes\{x,2\}$. Для реализации этого на современном Си-подобном языке программирования мы могли бы написать программу, подобную той, что представлена на рис. 1.

```
if(x < 0)
    y = 1;
else if(x < 10)
    y = 2;
else
    throw new Exception("Incorrect condition");
```

Рис. 1. Программа, частично реализующая $lt(x,0)\otimes\{x,1\}\oplus lt(x,10)\otimes\{x,2\}$

Однако, такая программа не полностью соответствует рассмотренной спецификации. Дело в том, что второй оператор присваивания (`y = 2`) может выполниться только в случае $0\le x<10$, в то время как согласно спецификации это может произойти в случае $x<10$, что включает и многие другие возможности. Хотя явного противоречия здесь нет (ничего такого, что бы не соответствовало спецификации, не происходит), по возможности лучше использовать специальные «защищённые» инструкции, которые в полной мере аналогичны установленной форме множеств присваивания. Например, воспользуемся командной выбора в форме, использованной в [2] (рис. 2).

```
if
    x < 0 => y = 1;
    x < 10 => y = 2;
fi
```

Рис. 2. Команда выбора

В такой команде разные варианты проверяются и выполняются не последовательно, а в произвольном порядке. То есть, первым может быть проверено условие $x < 10$, и выполнена соответствующая инструкция. Если ни одно из условий не выполняется, происходит авост. Так что для соответствия обычному условному оператору введённую команду выбора нужно всегда дополнять особенным вариантом, условие которого является отрицанием дизъюнкции всех остальных условий, а действие – пустым

оператором.

Замечания, касавшиеся условного оператора, также касаются и Си-подобного оператора `switch`. Кроме этого, нужно помнить о том, что обычно он работает только со значениями, определёнными на этапе компиляции.

3.3 Влияние инструкций на состояние программы

Пусть установлено, что перед некоторой инструкцией *i* выполняется условие $P = f(\vec{x})$. Условие, которое будет выполняться после применения инструкции, буем обозначать *Q*. Вначале выполняется условие *T* (истина), если специально не оговорены дополнительные обстоятельства. Ниже будут сделаны замечания относительно крайних значений условий (истина и ложь).

Выделим следующие элементарные инструкции и рассмотрим, каким образом, будучи исполненными, они изменяют состояние программы.

1. Пустая инструкция (*nop*): $Q = P = f(\vec{x})$. Эта инструкция не производит никаких преобразований.

2. Инструкция аварийной остановки (*terminate*): $Q = F$ (ложь). Условие, которое оценивается, как ложное, обычно означает, что программа выполнила какую-то недопустимую операцию.

3. Обмен значениями $x_i \leftrightarrow x_j$: Q получается из P единовременным обменом местами всех вхождений x_i и x_j. С использованием дополнительной временной переменной t этот процесс можно представить, как $Q = P[t/x_i][x_i/x_j][x_j/t]$. Здесь и далее запись вида $P[e/x]$ будет обозначать предикат, полученный из P одновременной заменой всех вхождений переменной *x* на выражение *e*.

4. Присваивание вида $x_i \leftarrow e(\vec{x})$: $Q = P[e_{xi}^{-1}(\vec{x}')/x_i]$. $e_{xi}^{-1}(\vec{x}')$ получается следующим образом. Составим уравнение $x_i = e_{xi}^{-1}(\vec{x}')$ и постараемся решить его относительно x_i (x_i' в это время считается особенной переменной, не связанной с x_i). Решение такого уравнения и будем обозначать $e_{xi}^{-1}(\vec{x}')$. Если же $e(\vec{x})$ не включает x_i, ситуация упрощается: в *P* следует исключить функции, зависящие от x_i, и заменить их на предикат вида $eq(x_i, e(\vec{x}))$.

Рассмотрим пример, в котором $P = eq(add(mul(a,x),b),0)$ или на «обычном» языке $P = \{ax + b = 0\}$. Найдём условие, которое будет верно после инструкции

$x \leftarrow 2x+1$. Решим простое уравнение $x'=2x+1$ относительно x: $x=(x'-1)/2$ и выполним замену всех вхождений переменной x в P на $((x-1)/2)$. Получим $Q=\{a(x-1)/2+b=0\}=\{ax-a+2b=0\}$.

Решение уравнения $x_i=e_{xi}^{-1}(\vec{x}')$ составляет отдельную нетривиальную задачу. Дело в том, что это уравнение должно быть решено в символьном виде, а не численно, то есть должна быть найдена формула, включающая только введённые выше операции и переменные программы. Например, четвёртая степень для алгебраических уравнений ($ax^4+bx^3+cx^2+dx+e=0$) является наивысшей, при которой существует аналитическое решение в радикалах в общем виде (то есть при любом значении коэффициентов).

Сделаем некоторые пояснения относительно интерпретации условий, представленных в виде предикатов, и их крайних значениях – истине и лжи. Условие $f_1(\vec{x})$ означает, что из всех возможных комбинаций значений переменных программы сейчас могут быть представлены только те, которые удовлетворяют $f_1(\vec{x})$. То есть, вводится ограничение на пространство значений.

Отсутствие всякого ограничения обозначается T (истина), что соответствует универсальному множеству U. Введение условия $f_1(\vec{x})$ можно интерпретировать как конъюнкцию $T \wedge f_1(\vec{x})=f_1(\vec{x})$, что соответствует операции пересечения множеств $U \cap U_{f1}=U_{f1}$.

Постепенно вводя всё новые и новые ограничения, можно добиться такого, что ни одна возможная комбинация переменных программы не будет удовлетворять им. Это значит, что условию соответствует пустое множество переменных – $\varnothing$, а в терминах алгебры логики – F (ложь).

Известно так же, что T играет роль нейтрального элемента относительно конъюнкции и аннулирующего элемента относительно дизъюнкции, а F, наоборот, играет роль нейтрального элемента относительно дизъюнкции и аннулирующего элемента относительно конъюнкции. Аналогичные законы действует и для множеств, где аналогом T является универсальное множество, а F – пустое множество.

4 Выводы

Вопросы автоматизации синтеза программного обеспечения являются важным направлением исследований в области информационных технологий. Работы ведутся как по направлению обычных средств автоматизированного проектирования и инструментов,

облегчающих рутинные рабочие процессы, так и по направлению интеллектуального конструирования алгоритма программы. В статье рассмотрена связь между спецификацией программного обеспечения и элементарными инструкциями языка программирования.

В качестве спецификации выбран популярный механизм формулирования предусловий и постусловий программы с помощью исчисления предикатов. На основе заданной таким образом спецификации формируются простейшие инструкции некоторого абстрактного императивного языка программирования, в частности, операции присваивания. С другой стороны, так же показано влияние инструкций на состояние переменных, которыми оперирует программа.

Полученные теоретические основания автоматической генерации простейших программ на основе формальной спецификации программного обеспечения позволяют развивать создание реальных инструментов, способных создавать самостоятельно целые программные модули. В целом область автоматического синтеза программ является крайне перспективным направлением информационных технологий с точек зрения экономики и надёжности.

Список литературы

1. Белоусов А. И., Ткачев С. Б. Дискретная математика: Учебник для вузов / под ред. В. С. Зарубина, А. П. Крищенко. 3-е изд., стереотип. М.: Изд-во МГТУ им. Н. Э. Баумана, 2004. 744 с.
2. Грис Д. Наука программирования. М.: Мир, 1984. 416 с., ил.
3. Минашин Г. А. Место приёмов дедуктивной верификации в автоматическом синтезе алгоритмов // Молодежный научно-технический вестник. 2013. № 12. Режим доступа: http://sntbul.bmstu.ru/doc/640887.html (дата обращения: 02.04.2014).
4. Новиков Ф. А. Дискретная математика для программистов. Учебник для вузов. 2-е изд. СПб.: Питер, 2007. 364 с.
5. Manna Z., Waldinger R., Fundamentals of deductive program synthesis. Computer and Systems Sciences, Springer-Verlag, Berlin, 1991, 62 p.