УДК: 004.05

Название: Предметно-ориентированные системы переходов: объектная модель и язык

Автор(ы):

Ануреев И.С. (Институт систем информатики СО РАН)

Аннотация: В статье представлены объектная модель и язык предметно-ориентированных систем переходов — нового формализма, предназначенного для спецификации и апробации формальных методов обеспечения надежности программного обеспечения.

Ключевые слова: предметно-ориентированные системы переходов, семантика, верификация, онтология

1. Введение. Обеспечение надежности программного обеспечения (ПО) — актуальная задача теории и практики программирования. Важную роль в этом играют формальные методы. В настоящее время имеется довольно много инструментов разработки надежного ПО, базирующихся на формальных методах. Они покрывают многие аспекты его разработки, от проектирования и прототипирования программных систем (ПС) до их формальной спецификации и верификации.

Однако, если в Semantic Web прослеживается тенденция к интеграции разнородных данных и сервисов, в разработке надежного ПО мы по-прежнему имеем дело с разрозненным набором отдельных инструментов, каждый из которых покрывает лишь отдельный специфический аспект разработки и, как правило, расчитан на использование лишь с небольшим числом компьютерных языков. Также заметен разрыв между огромным потенциалом развитых формальных методов и, за редким исключением, игрушечными примерами их применения [10]. Среди трудностей, препятствующих широкому распространению формальных методов, отметим сложность их изучения, высокую цену внедрения и неверие в них у представителей программной индустрии. Недостаточное внимание также уделяется технологическим аспектам разработки формальной семантики компьютерных языков, которая играет важную роль в обеспечении надежности ПО.

В [2, 3, 6] предложен унифицированный подход к обеспечению надежности программного обеспечения, который охватывает такие этапы разработки ПО, как прототипирование, проектирование, спецификация и верификация ПС. Этот подход также был использован для разработки формальной операционной семантики и логики безопасности (варианта аксиоматической семантики) компьютерных языков [5]. Он базируется на языке описания специализированных систем переходов. С помощью этих систем описываются модели ПС (в случае разработки семантики компьютерного языка описывается модель абстрактной машины этого языка.). Модель представляет собой пару — предметно-ориентированный язык, описывающий логику функционирования ПС, и его выполнимую семантику, описываемую специализированной системой переходов. Поэтому мы называем

эти системы переходов предметно-ориентированными системами переходов (ПОСП). На языке ПОСП специфицируются формальные методы обеспечения надежности ПО, применяемые к моделям ПС, что позволяет достаточно быстро внедрять их в процесс разработки ПО. ПОСП можно также рассматривать как "технологические" машины абстрактных состояний [8], в которых формализованы язык и объектная модель состояний и правил переходов. При этом, в отличие от языков реализации машин абстрактных состояний ASML [7] и XASM [9], обеспечивается более высокий уровень абстракции при спецификации ПС.

В [1, 4, 6] был представлен язык описания ПОСП Atoment и подъязыки для описания отдельных видов ПОСП, ориентированных на решение тех или иных задач разработки надежного ПО. В этой статье мы представляем общую объектную модель всех видов ПОСП и новую более полную версию языка Atoment, базирующуюся на этой модели.

Работа выполнена при финансовой поддержке РФФИ, проект № 11-01-0028-а «Интегрированный мультиязыковый подход к верификации императивных программ» и междисциплинарного интеграционного проекта СО РАН № 3 «Принципы построения онтологии на основе концептуализаций средствами логических дескриптивных языков».

2. Предварительные понятия и обозначения.

Пусть Int, Nat, Nat^0 и $Bool$ обозначают множество целых чисел, множество натуральных чисел, $Nat \cup \{0\}$ и $\{true, false\}$, соответственно.

Пусть A^* (A^+) — множество всех конечных (непустых) последовательностей, состоящих из элементов множества A. Пусть seq_{emp} обозначает пустую последовательность. Пусть a_1 ... a_n обозначает последовательность из элементов a_1, ..., a_n. Пусть $len(a)$ и $a.i$ обозначают длину и i-й элемент последовательности или кортежа a, соответственно.

Пусть $a_{\in X}$ обозначает элемент a множества X, а $a_{\notin X}$ — элемент, не принадлежащий множеству X. Пусть $a_{\subseteq X}$ обозначает подмножество a множества X. Пусть $pset(X)$ — множество всех подмножеств множества X.

Пусть $X \to Y$ ($X \to_t Y$) обозначает множество всех (тотальных) функций из X в Y. Пусть $dom(f)$ обозначает область определения функции f. Будем считать, что $f(x) = und$, если $x \notin dom(f)$. Пусть $dom(f) \cap dom(g) = \emptyset$. Объединение $f \cup g$ функций f и g определяется как функция h такая, что $dom(h) = dom(f) \cup dom(g)$, $h(x) = f(x)$ для $x \in dom(f)$ и $h(x) = g(x)$ для $x \in dom(g)$. Пусть $im(f)$ обозначает образ функции f, т. е. множество $\{f(x) \mid x \in dom(f)\}$. Пусть $im(f, X)$ обозначает образ функции f относительно множества X, т. е. множество $\{f(x) \mid x \in X\}$.

Логическая функция $oDif$ определяется следующим образом: $oDif(f, g, M) = true$ тогда и только тогда, когда $f(x) = g(x)$ для $x \notin M$. Таким образом, значения функций f и g могут отличаться только на элементах множества M.

Говорят, что множество A определяется конструкторами f_1, ..., f_n, если выполнены

следующие свойства:

- для любого $a \in A$ существуют $1 \leq i \leq n$ и $b \in dom(f_i)$ такие, что $a = f_i(b)$;
- для любых $1 \leq i \leq n$, $1 \leq j \leq n$, $a \in dom(f_i)$, и $b \in dom(f_j)$, если $f_i(a) = f_j(b)$, то $i = j$, и $a = b$.

Говорят, что множество A определяется конструкторами f_1, ..., f_n с ограничением $P \in A \to Bool$, если A определяется f_1, ..., f_n, и для любого $a \in A$ выполнено $P(a) = true$. Говорят, что a содержит b, если $a = b$, или существуют конструктор f и элементы x_1, ..., x_n, y_1, ..., y_m, c такие, что $a = f(x_1, ..., x_n, c, y_1, ..., y_m)$, и c содержит b.

Пусть $a \in A$ и $a = f_i(b)$. Расширим функции len и . на множества, определяемые конструкторами, следующим образом: если b — кортеж или последовательность, то $len(a) = len(b)$, и $a.k = b.k$. Например, $len(f_1(a, b)) = 2$, и $f_1(a, b).1 = a$ для кортежа (a, b), и $len(f_2(a)) = 3$, и $f_2(a).1 = b$ для последовательности $a = b\ c\ d$.

Система переходов определяется как пара $(Conf, tr)$, где $Conf$ — множество, $tr \in Conf \times Conf \to Bool$. Элементы $Conf$ называются конфигурациями. Функция tr называется отношением перехода.

3. Базовые понятия теории предметно-ориентированных систем переходов. Опишем виды объектов, из которых строится объектная модель ПОСП.

3.1. Атомы, символы, элементы. Пусть At — множество объектов, называемых атомами. Пусть $Int \subseteq At$ и $true, und \in At$.

Пусть $+v, \text{-}v \in At$. Атомы $+v$ и $\text{-}v$ называются спецификаторами аргументов. Пусть $ArgSpec = \{+v, \text{-}v\}$. Элементы множества $\{x \in At^+ \mid$ если $len(x) \neq 0$, то существует $1 \leq i \leq len(x)$ такое, что $x.i \notin ArgSpec\}$ называются символами. Пусть Sym обозначает множество символов. Число вхождений спецификаторов аргументов в символ f называется местностью f и обозначается $arity(f)$.

Множество El элементов определяется конструкторами $El_{at} \in At \to El$, $El_{seq} \in El^* \to El$, и $El_{fun} \in \cup_{n \in Nat_0}(El^n \to El) \to El$. Элементы $e_{\in im(El_{at})}$, $e_{\in im(El_{seq})}$ и $e_{\in im(El_{fun})}$ называется атомарным элементом, символьным вызовом и функциональным элементом, соответственно.

Далее, если не оговорено противное, буквы e и p (возможно, с индексами и штрихами) обозначают элементы и последовательности элементов, соответственно.

Пусть $a \in At$, $M \subseteq At$ и $b = b_1\ ...\ b_n \in At^+$. Тогда a_{el}, M_{el} и b_{el} обозначают $El_{at}(a)$, $\{a_{el} \mid a \in M\}$ и $(b_1)_{el}\ ...\ (b_n)_{el}$, соответственно. Для $e = i_{el}$, где $i \in Int$, пусть e_{int} обозначает i. Пусть $[p_1\ ...\ p_n]$ и $/[b_1\ ...\ b_n]/$ обозначают $El_{seq}(p_1\ ...\ p_n)$ и $[(b_1)_{el}\ ...\ (b_n)_{el}]$, соответственно. Пусть $/k[a_1\ ...\ a_k\ p_1\ ...\ p_n]$, $[p_1\ ...\ p_n\ b_1\ ...\ b_m]m/$ и $/k[a_1\ ...\ a_k\ p_1\ ...\ p_n\ b_1\ ...\ b_m]m/$ обозначают

$[(a_1)_{el}\ ...\ (a_k)_{el}\ p_1\ ...\ p_n]$, $[p_1\ ...\ p_n\ (b_1)_{el}\ ...\ (b_m)_{el}]$ и $[(a_1)_{el}\ ...\ (a_k)_{el}\ p_1\ ...\ p_n\ (b_1)_{el}\ ...\ (b_m)_{el}]$, соответственно.

Пусть f — функция, $dom(f) \subseteq El$, и $M \in dom(f)$. Пусть $nar(f, M)$ обозначает функцию g такую, что $g(x) = f(x)$ для $x \in M$, и $g(x) = und_{el}$ для $x \in dom(f) \setminus M$.

3.2. Контексты вычисления. Пусть $EvCont$ — множество объектов, называемых контекстами вычисления. Определение контекстов вычисления может различаться в различных ситуациях. Далее буква q (возможно, с индексами и штрихами) обозначает контекст вычисления.

3.3. Состояния. Пусть $StSym \subseteq Sym$. Состояние s относительно базиса $(At, StSym)$ определяется как функция из $StSym \times Int \to_t \cup_{n \in Nat_0}(El^n \to_t El)$ такая, что $s(f, i) \in El^{arity(f)} \to_t El$, и выполнено свойство конечности состояния: $\{p \mid s(f, i)(p) \neq und_{el}, f \in StSym, i \in Int \text{ и } p \in El^{arity(f)}\}$ конечно. Пусть St — множество всех состояний относительно базиса $(At, StSym)$. Пусть $EvCont = St \times Int$. Элемент $s(f, i)$ называется интерпретацией (значением) символа f в контексте (s, i).

3.4. Контексты состояний. Функция $c \in StSym \to_t Int$ называется контекстом состояния. Пусть $StCont$ — множество всех контекстов состояний.

Элемент e называется представлением контекста состояния c и обозначается $contRep(c)$, если $e = El_{fun}(g)$, $g \in El \to El$, $g(El_{seq}(f_{el})) = (c(f))_{el}$ для $f \in StSym$, и $g(e) = und_{el}$ для $e \in El \setminus \{El_{seq}(f_{el}) \mid f \in StSym\}$. Пусть $n > 0$ и $f_1, ..., f_n \in StSym$.

Далее, если не оговорено противное, буквы s и c (возможно, с индексами и штрихами) обозначают состояния и контексты состояния, соответственно.

3.5. Интерпретации символов. Пусть $PreSym \subseteq Sym$. Интерпретация символов τ относительно базиса $(At, PreSym)$ определяется как функция из $PreSym \times EvCont \to_t \cup_{n \in Nat_0}(El^n \to_t El)$ такая, что $\tau(f, q) \in El^{arity(f)} \to_t El$ для любых $f \in PreSym$, и $q \in EvCont$. Элемент $\tau(f, q)$ называется значением символа f при интерпретации τ в контексте q.

3.6. Элементы как примеры символов. Множество $TypArg$ типизированных аргументов определяется конструктором $TypArg \in (El \times ArgSpec)^* \to TypArg$. Элемент e — пример $f_{\in Sym}$ относительно $a_{\in TypArg}$ тогда и только тогда, когда выполнено первое подходящее свойство:

- если $e = [e']$, $f = u_{\in ArgSpec}$, и $a = TypArg((e', u))$, то $true$;
- если $e = [e'\ p_{\in El^+}]$, $f = u_{\in ArgSpec}\ v_{\in At^*}$, и $a = TypArg((e', u)\ d)$, то $[p]$ — пример v относительно $TypArg(d)$;
- если $e = /[f_{\in At}]/$, то $true$;

- если $e = /1[b_{\in At}\ \ p_{\in El^+}]$, и $f = b\ w$, то $[p]$ — пример w относительно a;
- $false$.

Один и тот же элемент может быть примером для разных символов, поэтому возможен конфликт при выборе символа, соответствующего этому элементу. Мы считаем, что определена функция $match_{sym} \in pset(Sym) \times El \to StSym \times TypArg$, разрешающая этот конфликт, такая, что, если $match_{sym}(Sy, e) = (f, a)$, то e — пример f относительно a. Эта функция зависит от множества допустимых символов Sy. Элемент e называется примером f, если $match_{sym}(Sy, e) = (f, a)$. Элемент e называется вызовом символа f, если e — пример f. Элемент e называется вызовом символа f с аргументами a, если $match_{sym}(Sy, e) = (f, a)$.

3.7. Семантика элементов. Функция означивания элементов $val \in El \times EvCont \to El$ относительно базиса $(PreSym, \tau)$, где $EvCont = St \times StCont$, определяется следующим образом (применяется первое подходящее правило):

- $val(e_{\in im(El_{at}) \cup im(El_{fun})}, q) = e$;
- если $match_{sym}(StSym \cup PreSym, e) = (f, a)$, и $f \in StSym \setminus PreSym$, то $val(e, q) = q.1(f, q.2(f))(argVal(a, q))$;
- если $match_{sym}(StSym \cup PreSym, e) = (f, a)$, $f \in StSym \cap PreSym$, и $q.1(f, q.2(f))(argVal(a, q)) = und_{el}$, то $val(e, q) = \tau(f, q)(argVal(a, q))$;
- если $match_{sym}(StSym \cup PreSym, e) = (f, a)$, и $f \in ContSym \cap PreSym$, то $val(e, q) = q.1(f, q.2(f))(argVal(a, q))$;
- если $match_{sym}(StSym \cup PreSym, e) = (f, a)$, и $f \in PreSym$, то $val(e, q) = \tau(f, q)(argVal(a, q))$;
- $val(e, q) = und_{el}$.

Элемент $val(e, q)$ называется значением элемента e в контексте q. Функция $argVal \in TypArg \times EvCont \to El^*$ возвращает значения аргументов символа f и определяется следующим образом (применяется первое подходящее правило):

- $argVal(TypArg(seq_{emp}), q) = seq_{emp}$;
- $argVal(TypArg((/1[+v\ e], w)\ d), q) = val(e, q)\ argVal(TypArg(d), q)$;
- $argVal(TypArg((/[\text{-}v\ e], w)\ d), q) = e\ argVal(TypArg(d), q)$;
- $argVal(TypArg((e, +v)\ d), q) = val(e, q)\ argVal(TypArg(d), q)$;

- $argVal(TypArg((e, \text{-}v)\ d), q) = e\ argVal(TypArg(d), q)$.

Исходя из определений функций val и $argVal$, можно отметить следующее. Если элемент соответствует аргументу символа f со спецификатором $+v$, то функция $q.1(f, q.2(f))$ получает на вход значение этого элемента. Если элемент соответствует аргументу символа f со спецификатором -v, то функция $q.1(f, q.2(f))$ получает на вход сам этот элемент.

3.8. Подстановки. Функция $\sigma \in At \to El^*$ называется подстановкой. Пусть Sub — множество всех подстановок. Если $dom(\sigma) = \{x_1, ..., x_n\}$, то σ может записываться как $((x_1, \sigma(x_1)), ..., (x_n, \sigma(x_n)))$. Функция подстановки $sub \in El^* \times Sub \to El^*$ относительно σ определяется следующим образом (применяется первое подходящее правило):

- $sub(seq_{emp}, \sigma) = seq_{emp}$;
- $sub(El_{at}(u_{\in dom(\sigma)}), \sigma) = \sigma(u)$;
- $sub(e_{\in im(El_{at}) \cup im(El_{fun})}, \sigma) = e$;
- $sub([p], \sigma) = [sub(p, \sigma)]$;
- $sub(e\ p, \sigma) = sub(e, \sigma)\ sub(p, \sigma)$.

3.9. Конфигурации. Конфигурация t относительно базиса $(At, StSym, PreSym, \tau)$ определяется как тройка (p, s, c). Компоненты p, s и c конфигурации специфицируют, что выполняется, над чем и в каком контексте, соответственно.

Далее, если не оговорено противное, буква t (возможно, с индексами и штрихами) обозначает конфигурацию.

3.10. Правила перехода. Пусть $+s \in At$. Элементы вида u_{el} и $/[+s\ u]/$, где $u \in At$, называются спецификаторами переменных образца. Говорят, что эти элементы специфицируют переменную u. Пусть $VarSpec$ — множество всех спецификаторов переменных образца. Значение переменной u, определяемой спецификатором $El_{at}(u)$, принадлежит El. Значение переменной u, определяемой спецификатором $/[+s\ u]/$, принадлежит El^*.

Множество Rul правил перехода определяется конструктором $Rul \in El \times VarSpec^* \times El^* \to Rul$ со следующим ограничением: если $r \in Rul$, $1 \le i, j \le len(r.2)$, $r.2.i$ и $r.2.j$ специфицируют одну и ту же переменную, то $i = j$.

Пусть $r \in Rul$. Элемент $r.1$, переменные, входящие в $r.2$, и последовательность $r.3$ называются образцом, переменными образца и телом правила r, соответственно.

Говорят, что r применимо к e относительно $\sigma_{\in Sub}$, если $sub(r.1, \sigma) = e$, $dom(\sigma)$ — множество переменных образца правила r, $\sigma(u) \in El$ для $u_{el} \in r.2$, и $\sigma(u) \in El^*$ для $/[+s\ u]/ \in r.2$.

Правило может быть применимо к одному и тому же элементу относительно разных подстановок, поэтому возможен конфликт при выборе подстановки, соответствующей этому элементу. Мы считаем, что определена функция $match_{sub} \in El \times Rul \to Sub$, разрешающая этот конфликт, такая, что, если $match_{sub}(e, r) \neq und$, то r применимо к e относительно $match_{sub}(e, r)$.

На практике встречается случай, когда требуется подставить в качестве значения переменной образца не сопоставленный элемент, а его значение. Пусть $++v \in At$. Когда требуется сопоставить переменной образца элемент e и вычислить его значение, этот элемент заменяется в сопоставляемом выражении на $/1[++v\ e]$. Расширим определение функции $match_{sym}$ на этот случай. Соответствующая функция $match_{sub} \in El \times Rul \times Conf \to Sub$ определяется следующим образом:

- если $match_{sub}(e, r) = und$, то $match_{sub}(e, r, t) = und$;
- $dom(match_{sub}(e, r, t)) = dom(match_{sub}(e, r))$;
- для любого $u \in dom(match_{sub}(e, r))$ выполнено первое подходящее свойство:
  - если $u \in r.2$ и $match_{sub}(e, r)(u) = /1[++v\ e]$, то $match_{sub}(e, r, t)(u) = val(e, (t.2, t.3))$;
  - если $u \in r.2$, то $match_{sub}(e, r, t)(u) = match_{sub}(e, r)(u)$;
  - если $/[+s\ u]/ \in r.2$, то $len(match_{sub}(e, r, t)(u)) = len(match_{sub}(e, r)(u))$, и для любого $1 \leq i \leq len(match_{sub}(e, r)(u))$ выполнено первое подходящее свойство:
    - если $match_{sub}(e, r)(u).i = /1[++v\ e]$, то $match_{sub}(e, r, t)(u).i = val(e, (t.2, t.3))$;
    - $match_{sub}(e, r, t)(u).i = match_{sub}(e, r)(u).i$.

Говорят, что r применимо к e в конфигурации t, если $match_{sub}(e, r, t) \neq und$.

3.11. Определение ПОСП. Пусть $tr_{interp} \in Conf \times Conf \to Bool$ и $PredEl \subseteq El$. Система переходов $\rho = (Conf, tr)$ называется предметно-ориентированной относительно базиса $(At, StSym, PreSym, \tau, RulSet_{\subseteq Rul}, match_{sym}, match_{sub}, PredEl, tr_{interp})$, если выполнены следующие свойства:

- $Conf$ — множество всех конфигураций относительно $(At, StSym, PreSym, \tau)$;
- $tr(t, t') = true$ тогда и только тогда, когда $t.1 = e\ p$, и выполнено одно из двух условий: $e \in PredEl$ и $tr_{interp}(t, t') = true$, или $e \notin PredEl$, и существует $r_{\in RulSet}$ такое, что $tr(t, t', r) = true$. Функция tr с дополнительным аргументом r определяется следующим образом: $tr(t, t', r) = true$ тогда и только тогда, когда r применимо к e в t, $t'.1 = sub(r.3, match_{sub}(e, r, t))\ p$, $t'.2 = t.2$ и $t'.3 = t.3$.

Предметная ориентированность системы переходов ρ определяется ее базисом. Пусть $base(\rho)$ обозначает базис ρ. Функция tr_{interp} называется интерпретированным отношением перехода, а элементы $PredEl$ — предопределенными элементами. Пусть $tr(t, t') = true$. Тогда $t.1$ ($t.2$, $t.3$) и $t'.1$ ($t'.2$, $t'.3$) называются входной и выходной управляющими последовательностями (входным и выходным состояниями, входным и выходным контекстами) перехода, соответственно.

3.12. Понятия, связанные с переходом в ПОСП. Конфигурация t называется заключительной, если не существует t' такой, что $tr(t, t') = true$. Конфигурация t называется заключительной относительно $r \in RulSet$, если не существует t' такой, что $tr(t, t', r) = true$. Трассой называется конечная последовательность конфигураций t_1 ... t_n такая, что $tr(t_i, t_{i+1}) = true$ для $1 \leq i \leq n - 1$. Пусть $Trace$ — множество всех трасс.

В следующих разделах, если не оговорено противное, используются следующие сокращения. Пусть $f \in At$. Пусть A_f обозначает $(f, t.3(f))$ для символов вида f. Пусть A_f обозначает $(f\ \text{-}v, t.3(f\ \text{-}v))$ для символов вида f -v. Пусть v_f и v'_f обозначают $t.2(A_f)$ и $t'.2(A_f)$, соответственно.

4. Предопределенные символы. В этом разделе описываются общезначимые предопределенные символы из $PreSym$.

4.1. Символ -vv -v. Пусть -$vv \in At$. Символ -vv -v имеет следующую семантику: $\tau(\text{-}vv\ \text{-}v, q)(e) = e$.

4.2. Символ +vv +v. Пусть +$vv \in At$. Символ +vv +v имеет следующую семантику: $\tau(+vv\ +v, q)(e) = val(e, q)$.

4.3. Нумерованные элементы. Пусть $el \in At$. Символ el +v имеет следующую семантику:

- если $e \in Int_{el}$, то $\tau(el\ +v, q)(e) = /1[el\ e]$;
- в противном случае $\tau(el\ +v, q)(e) = und_{el}$.

Элементы вида $/[el\ i_{\in Int}]/$ называются нумерованными элементами.

Пусть $(x)^n$ обозначает последовательность из n вхождений атома x.

4.4. Контекстные символы. Пусть $cont \in At$. Семейство символов $\{cont\ +v\ (\text{-}v)^n\}_{n \in Nat}$ имеет следующую семантику:

- если $n \geq 1$ и $q \in St \times StSym$, то $\tau(cont\ +v\ (\text{-}v)^n, q)((i_{\in Int})_{el}, e_1, ..., e_n) = val([e_1\ ....\ e_n], (q.1, q.2, i))$;
- в противном случае $\tau(cont\ +v\ (\text{-}v)^n, q)(e, e_1, ..., e_n) = und_{el}$.

Элементы вида $/1[cont\ p]/$ называются контекстными элементами.

Пусть $EvCont = St \times StCont \times Int$. Пусть $q' = (q.1, q.2)$. Функция $val \in El \times EvCont \to El$ относительно базиса $(PreSym, \tau)$ определяется следующим образом (применяется первое подходящее правило):

- $val(e_{\in im(El_{at}) \cup im(El_{fun})}, q) = e$;
- если $match_{sym}(StSym \cup PreSym, e) = (f, a)$, и $f \in StSym \setminus PreSym$, то $val(e, q) = q.1(f, q.3)(argVal(a, q'))$;
- если $match_{sym}(StSym \cup PreSym, e) = (f, a)$, $f \in StSym \cap PreSym$, и $s(f, q.3)(argVal(a, q')) = und_{el}$, то $val(e, q) = \tau(f, q)(argVal(a, q'))$;
- если $match_{sym}(StSym \cup PreSym, e) = (f, a)$, и $f \in ContSym \cap PreSym$, то $val(e, q) = q.1(f, q.3)(argVal(a, q'))$;
- если $match_{sym}(StSym \cup PreSym, e) = (f, a)$, и $f \in PreSym$, то $val(e, q) = \tau(f, q)(argVal(a, q'))$;
- $val(e, q) = und_{el}$.

4.5. Функциональные элементы. Пусть $sym \in At$. Семейство символов $\{sym\ (\text{-}v)^n\}_{n \in Nat}$ имеет следующую семантику:

- если $n \geq 1$, $f_{\in StSym} = e_1\ ...\ e_n$, и $q \in St \times StSym$, то $\tau(sym\ (\text{-}v)^n, q)(e_1, ..., e_n) = El_{fun}(q.1(f, q.2(f)))$;
- если $n \geq 1$, $f_{\in StSym} = e_1\ ...\ e_n$, и $q \in St \times StSym \times Int$, то $\tau(sym\ (\text{-}v)^n, q)(e_1, ..., e_n) = El_{fun}(q.1(f, q.3))$;
- в противном случае $\tau(sym\ (\text{-}v)^n, q)(e_1, ..., e_n) = und_{el}$.

Элементы вида $/[sym\ f_{\in Sym}]/$ называются функциональными элементами.

4.6. Атомы и символы. Пусть $is, atom, sym \in At$. Символы *-v is atom* и *-v is sym* описывают атомы и символы, соответственно. Они имеют следующую семантику:

- $s(\text{-}v\ is\ atom)(x) = true$ тогда и только тогда, когда $x \in At$;
- $s(\text{-}v\ is\ sym)(x) = true$ тогда и только тогда, когда $x \in Sym$.

4.7. Символы $oDif\ +v\ +v\ +v$ **и** $oDifSeq\ +v\ +v\ +v$**.** Символ $oDif\ +v\ +v\ +v$ имеет следующую семантику: $\tau(oDif\ +v\ +v\ +v, q)(x, y, z) = true$ тогда и только тогда, когда выполнено первое подходящее свойство:

- если $x = El_{fun}(x'), x' \in El^n \to_t El, y = El_{fun}(y'), y' \in El^n \to_t El$, то $oDif(x', y', \{z\}) = true$;
- $false$.

Символ $oDifSeq\ +v\ +v\ +v$ имеет следующую семантику: $\tau(oDifSeq\ +v\ +v\ +v, q)(x, y, z) = true$ тогда и только тогда, когда выполнено первое подходящее свойство:

- если $x = El_{fun}(x')$, $x' \in El^n \to_t El$, $y = El_{fun}(y')$, $y' \in El^n \to_t El$ и $z = [z_1\ ...\ z_n]$, то $oDif(x', y', \{z_1, ..., z_n\}) = true$;
- $false$.

4.8. Ветвление по неопределенности. Пусть $else \in At$. Символ $+v\ else\ -v$ называется ветвлением по неопределенности и имеет следующую семантику:

- если $e = und_{el}$, то $\tau(+v\ else\ -v, q)(e, e') = val(e', q)$;
- В противном случае $\tau(+v\ else\ -v, q)(e, e') = e$.

Вариант $+v\ elsev\ +v$ этого символа имеет семантику

- если $e = und_{el}$, то $\tau(+v\ elsev\ +v, q)(e, e') = e'$;
- в противном случае $\tau(+v\ elsev\ +v, q)(e, e') = e$.

4.9. Условный элемент. Пусть $if, then, else \in At$. Символ $if\ +v\ then\ -v\ else\ -v$ называется условным элементом и имеет следующую семантику:

- если $e = true_{el}$, то $\tau(if\ +v\ then\ -v\ else\ -v, q)(e, e', e'') = val(e', q)$;
- в противном случае $\tau(if\ +v\ then\ -v\ else\ -v, q)(e, e', e'') = val(e'', q)$.

4.10. Нормализация постусловия. Пусть $ver \in At$. Символ $postNorm\ -v\ +v$ называется нормализацией постусловия и используется в дедуктивных ПОСП для нормализации постусловия при проверке корректности тела функции. Пусть $len(p) = n$. Этот символ имеет следующую семантику (применяется первый подходящий случай):

- $\tau(postNorm\ -v, q)(e, i_{\notin Int}) = und_{el}$;
- $\tau(postNorm\ -v, q)(e_{\in im(El_{at}) \cup im(El_{fun})}, i) = e$;
- $\tau(postNorm\ -v, q)([nextSt\ p], i) = [\tau(postNorm\ -v, q)(p_1, i)\ ...\ \tau(postNorm\text{-}v, q)(p_n, i)]$;
- $\tau(postNorm\ -v, q)([p], i) = [ver\ i\ i\ \tau(postNorm\ -v, q)(p_1, i)\ ...\ \tau(postNorm\text{-}v, q)(p_n, i)]$.

4.11. Операции над целыми числами. Целые числа задаются элементами вида i_{el}, где $i \in \{..., -2, -1, 0, 1, 2, ...\}$.

Пусть $+, -, *, \text{div}, \text{mod} \in At$. Арифметические операции сложения, вычитания, умножения, нахождения целой части от деления и нахождения остатка от деления определяются соответствующими символами $+v$ + $+v$, $+v$ - $+v$, $+v$ * $+v$, $+v$ div $+v$ и $+v$ mod $+v$.

Пусть $=, !=, <, <=, >, >= \in At$. Арифметические отношения над целыми числами определяются соответствующими символами $+v$ = $+v$, $+v$!= $+v$, $+v$ < $+v$, $+v$ <= $+v$, $+v$ > $+v$ и $+v$ >= $+v$.

4.12. Логические операции. Пусть $false \in At$. Логические константы задаются элементами $true_{el}$ и $false_{el}$.

Пусть $and, or, not, =>, <=> \in At$. Логические связки конъюнкция, дизъюнкция, отрицание, импликация и эквивалентность определяются соответствующими символами $+v$ and $+v$, $+v$ or $+v$, not $+v$, $+v$ => $+v$ и $+v$ <=> $+v$. В качестве ложного значения выступает любой элемент, не равный $true_{el}$. Выражение (e_1 *and ... and* e_n) является сокращением для ((... (e_1 *and* e_2) *and ...* e_{n-1}) *and* e_n). Аналогичное сокращение используется для *or*.

Пусть $forall, exists \in At$. Кванторы всеобщности и существования $\forall xA$ и $\exists xA$ определяются символами forall -v -v и exist -v -v, соответственно. Выражение $forall\ x_1\ ...\ x_n\ e$ является сокращением для ($forall\ x_1\ ...\ (forall\ x_{n-1}\ (forall\ x_n\ e))$). Аналогичное сокращение используется для *exist*.

5. Базовые виды предметно-ориентированных систем переходов. В этом разделе рассматриваются базовые виды ПОСП. Конкретные, используемые на практике ПОСП обычно определяются как комбинации базовых видов ПОСП.

5.1. ПОСП со счетчиком. Пусть $count \in At$. ПОСП ρ называется ПОСП со счетчиком, если $count \in StSym \setminus PreSym$, и $v_{count} \in \emptyset \to Int_{el}$ для любой $t \in Conf$. Символ *count* называется счетчиком;

5.2. ПОСП с возвращаемым значением. Пусть $val \in At$. ПОСП ρ называется ПОСП с возвращаемым значением, если $val \in StSym \setminus PreSym$. Символ *val* называется спецификатором возвращаемого значения. Говорят, что управляющая последовательность p возвращает значение $v_{\in El}$ в состоянии s в контексте c, если $tr((p, s, c), t') = true$ для некоторой конфигурации t' и $t'.2(val, t'.3(val))() = v$.

5.3. ПОСП с инкрементом счетчика. Пусть $count{+}{+} \in At$. ПОСП ρ со счетчиком и возвращаемым значением называется ПОСП с инкрементом счетчика, если $PredEl$ включает элемент $/[count{+}{+}]/$, называемый инкрементом счетчика, со следующей семантикой: $tr_{interp}(t, t') = true$, где $t.1 = /[count{+}{+}]/\ p$, тогда и только тогда, когда $t'.1 = p$, $oDif(t'.2, t.2, \{A_{count}, A_{val}\}) = true$, $v'_{count}() = ((v_{count}())_{int} + 1)_{el}$, $v'_{val}() = v'_{count}()$ и $t'.3 = t.3$.

5.4. ПОСП с генерацией нумерованных элементов. Пусть $newEl \in At$. ПОСП ρ со счетчиком и возвращаемым значением называется ПОСП с генерацией нумерованных элементов, если $PredEl$ включает элемент $/[newEl]/$, называемый генератором нумерованного элемента, со следующей семантикой: $tr_{interp}(t, t') = true$, где $t.1 = /[newEl]/\ p$, тогда и только тогда, когда $t'.1 = p$, $oDif(t'.2, t.2, \{A_{count}, A_{val}\}) = true$, $v'_{count}() = ((v_{count}())_{int} + 1)_{el}$, $v'_{val}() = /1[el\ v'_{count}()]$ и $t'.3 = t.3$.

5.5. Условные ПОСП. ПОСП ρ называется условной ПОСП, если выполнены следующие свойства:

- Rul определяется конструктором $Rul_{cond} \in El \times VarSpec^* \times El^* \times El \to Rul$. Дополнительная компонента $r.4$ называется условием r, а правило r — условным правилом;
- отношение применимости правила переопределяется следующим образом: r применимо к e в t, если $match_{sub}(e, r, t) \neq und$, и $val(sub(r.4, match_{sub}(e, r, t)), (t.2, t.3)) = true_{el}$.

Таким образом, условное правило применимо только в том случае, если выполнено его условие.

5.6. ПОСП с переменными истории. Пусть $hvar \in At$. ПОСП ρ со счетчиком называется ПОСП с переменными истории, если выполнены следующие свойства:

- для любого $i \in Int$ выполнено свойство $hvar\ i \in StSym \setminus PreSym$. Элемент вида $/[hvar\ i]/$ называется переменной истории ПОСП ρ, а $i_{\in Int}$ — индексом этой переменной. В переменных истории хранятся промежуточные результаты (история) функционирования ρ;
- Rul определяется конструктором $Rul_{hvar} \in El \times VarSpec^* \times El^* \times At^* \to Rul$ со следующим ограничением: множества переменных образца и элементов $r.4$ не пересекаются для любого $r_{\in RulSet}$. Дополнительная компонента $r.4$ называется декларацией переменных истории и используется для добавления переменных истории $/[hvar\ i]/$ с индексами $i \in Int$ такими, что $(v_{count}())_{int} < i \leq (v_{count}())_{int} + len(r.4)$, в управляющие последовательности. Элементы $r.4$ называются переменными истории правила r;
- пусть функция $hvarSub \in Rul \times St \to Sub$ определяется следующим образом: $dom(hvarSub)$ — множество переменных истории r и $hvarSub(r, s)(r.4.j) = /[hvar\ (v_{count}())_{int} + j]/$ для $1 \leq j \leq len(r.4)$. Отношение tr переопределяется следующим образом: $tr(t, t', r) = true$ тогда и только тогда, когда $t.1 = e\ p$, r применимо к e в t, $t'.1 = sub(r.3, match_{sub}(e, r, t) \cup hvarSub(r, t.2))\ p$, $oDif(t'.2, t.2, \{A_{count}\}) = true$, $v'_{count}() = ((v_{count}())_{int} + len(r.4))_{el}$, и $t'.3 = t.3$.

5.7. ПОСП с частично интерпретированными телами правил. ПОСП ρ называется ПОСП с частично интерпретированными телами правил, если tr переопределяется следующим образом: $tr(t, t', r) = true$ тогда и только тогда, когда $t.1 = e\ p$, и существует $r_{\in RulSet}$ такое, что r применимо к e в t, $t'.1 = partVal(sub(r.3, match_{sub}(e, r, t)), (t.2, t.3))\ p$, $t'.2 = t.2$, и $t'.3 = t.3$;

Пусть $EvCont = St \times StCont$. Функция $partVal \in El^* \times EvCont \to El^*$ вычисляет элементы вида $/1[interp\ e]$ в теле правила r и определяется следующим образом (применяется первое подходящее правило):

- $partVal(seq_{emp}, q) = seq_{emp}$;
- $partVal(e_{\in im(El_{at}) \cup im(El_{fun})}, q) = e$;
- $partVal(/1[interp\ e], q) = val(partVal(e, q), q)$;
- $partVal([p], q) = [partVal(p, q)]$;
- $partVal(e\ p, q) = partVal(e, s, c)\ partVal(p, q)$.

5.8. ПОСП с определениями символов. Эти ПОСП используются для спецификации предопределенных символов. Значение символа, сопоставленного с образцом, определяется как значение, возвращаемое телом правила. ПОСП ρ называется ПОСП с определениями символов, если выполнены следующие свойства:

- наряду с обычными правилами используются правила, в которых множество спецификаторов переменных $VarSpec$ переопределяется следующим образом. Элементы вида $/[\text{-}v\ u]/$ и u, где $u \in At$, называются спецификаторами переменных образца. Пусть $r \in RulSet$. Определяемый правилом r символ f является результатом замены в образце правила r переменных со спецификаторами $/[\text{-}v\ u]/$ и u на спецификаторы аргументов $\text{-}v$ и $+v$, соответственно.
- значение $val(e, q)$ вызова функции f в контексте $q_{\in St \times StCont}$ определяется следующим образом (применяется первое подходящее правило):
  - если $tr((e, q.1, q.2), t') = true$, и $tr((e, q.1, q.2), t'') = true$, где t', t'' — заключительные конфигурации, и $val((val), (t'.2, t'.3)) \neq val((val), (t''.2, t''.3))$, то $val(e, q) = und_{el}$;
  - если не существует заключительной конфигурации t' такой, что $tr((e, q.1, q.2), t') = true$, то $val(e, q) = und_{el}$;
  - если $tr((e, q.1, q.2), t') = true$, где t' — заключительная конфигурация, то $val(e, q) = val((val), (t'.2, t'.3))$.

Заметим, что при вычислении значения вызова символа f, определяемого правилом ПОСП ρ, после вычисления, следующие переходы выполняются в контексте, в котором вычислялся f, т. е. происходит откат из конфигурации, полученной в результате вычисления f, в исходную конфигурацию.

5.9. Бэктрекинг в ПОСП. Использование бэктрекинга в ПОСП расширяет выразительные возможности этих систем. Формально он определяется как отношение перехода на последовательностях расширенных конфигураций.

Пусть $EConf$ — множество четверок $(p, s, c, R_{\subseteq RulSet})$, называемых расширенными конфигурациями. Четвертая компонента определяет, какие правила перехода уже применялись при переходе из конфигурации (p, s, c). Пусть $BackSym$ — конечное множество символов. Оно специфицирует символы, значения которых сохраняются при возвратах. Функция $ifSt(s, s')$ возвращает состояние и определяется следующим образом:

- если $f \in BackSym$, то $ifSt(s, s')(f) = s'(f)$;
- если $f \in StSym \setminus BackSym$, то $ifSt(s, s')(f) = s(f)$.

Выделяется два вида бэктрекинга — управляемый бэктрекинг и полный перебор с возвратом.

5.10. ПОСП с управляемым бэктрекингом. Пусть $backtrack, branch \in At$. Пусть $ba(s, c)$ и $fi(s, c)$ обозначают $(backtrack_{el}, s, c, \emptyset)$ и $(seq_{emp}, s, c, \emptyset)$, соответственно. Пусть $w \in im(El_{seq})^*$, и $t, t' \in EConf$. Отношение перехода $tr \in EConf^* \times EConf^* \to Bool$ называется управляемым бэктрекингом, если $tr(u, u') = true$ тогда и только тогда, когда выполнено первое подходящее свойство:

- $u = u''\ (backtrack_{el}\ p, s, c, R)$, где $p \neq seq_{emp}$, или $R \neq \emptyset$, и $u' = u''\ ba(s, c)$;
- $u = u''\ (/1[branch\ [p']\ w]\ p, s, c, R)\ ba(s'', c'')$, и $u' = u''\ (/1[branch\ w]\ p, s, c, R)\ (p'\ p, ifSt(s, s''), c, \emptyset)$;
- $u = u''\ t\ (/[branch]/\ p, s, c, R)\ ba(s'', c'')$, и $u' = u''\ t\ ba(s'', c'')$;
- $u = (/[branch]/\ p, s, c, R)\ ba(s'', c'')$, и $u' = ba(ifSt(s, s''), c)$;
- $u = u''\ (/1[branch\ [p']\ w]\ p, s, c, R)$, и $u' = u''\ (/1[branch\ w]\ p, s, c, R)\ (p'\ p, s, c, \emptyset)$;
- $u = u''\ (/[branch]/\ p, s, c, R)$, и $u' = u''\ fi(s, c)$;
- $u = u''\ (e\ p, s, c, R)\ ba(s'', c'')$, $e \notin PredEl$, $r \in RulSet \setminus R$, $tr((e\ p, ifSt(s, s''), c), (p', s', c'), r) = true$, и $u' = u''\ (e\ p, s, c, R \cup \{r\})\ (p', s', c', \emptyset)$;
- $u = u''\ t\ t'\ ba(s'', c'')$, и $u' = u''\ t\ ba(s'', c'')$;

- $u = (p, s, c, R)\ ba(s'', c'')$, и $u' = ba(ifSt(s, s''), c)$;
- $u = ba(s, c)$, и $u' = fi(s, c)$;
- $u = u''\ (e\ p, s, c, R)$, $e \notin PredEl$, $r \in RulSet \setminus R$, $tr((e\ p, s, c), (p', s', c'), r) = true$, и $u' = u''\ (e\ p, s, c, R \cup \{r\})\ (p', s', c', \emptyset)$;
- если $u = u''\ (e\ p, s, c, R)$, $e \in PredEl$, $tr_{interp}((e\ p, s, c), (p', s', c')) = true$, и $u' = u''\ (ep, s, c, R)\ (p', s', c', \emptyset)$;
- $false$.

Элемент $backtrack_{el}$, называемый условием возврата, инициирует возврат в ПОСП. Элемент $/1[branch\ p]$ называется элементом ветвления и используется для выполнения перебора вариантов с возвратом.

ПОСП называется ПОСП с управляемым бэктрекингом, если tr на расширенных конфигурациях является управляемым бэктрекингом.

5.11. ПОСП с полным перебором. Отношение перехода $tr \in EConf^* \times EConf^* \to Bool$ называется полным перебором с возвратом, если $tr(u, u') = true$ тогда и только тогда, когда выполнено первое подходящее свойство:

- $u = u''\ (p, s, c, R)$, где $p \neq seq_{emp}$, или $R \neq \emptyset$, (p, s, c, R) — заключительная конфигурация и $u' = u''\ fi(s, c)$;
- $u = u''\ ([/1[cases\ p']\ p, s, c, R)\ u'''$, где $p \neq seq_{emp}$, или $R \neq \emptyset$, (p, s, c, R) — заключительная конфигурация, и $u' = u''\ ([/1[casesRest\ ()\ p']\ p, s, c, R)\ u'''$;
- $u = u''\ ([/1[casesRest\ (w_1\ ...\ w_n)1[if\ e\ then_{el}\ p']\ p'']\ p, s, c, R)\ fi(s'', c'')$, и $u' = u''\ (/1[casesRest\ (w_1\ ...\ w_n\ e)\ p'']\ p, s, c, R)$ $(/1[assume\ ((not(w_1\ and\ ...\ and\ w_n))\ and\ e)]\ p'\ p, ifSt(s, s''), c, \emptyset)$;
- $u = u''\ ([/1[casesRest\ (w_1\ ...\ w_n)1[else\ p']\ p'']\ p, s, c, R)\ fi(s'', c'')$, и $u' = u''\ (/1[assume\ (not(w_1\ and\ ...\ and\ w_n))]\ p'\ p, ifSt(s, s''), c, \emptyset)$;
- $u = (/[casesRest]/\ p, s, c, R)\ fi(s'', c'')$, и $u' = fi(ifSt(s, s''), c)$;
- $u = u''\ (/[casesRest]/\ p, s, c, R)\ fi(s'', c'')$, и $u' = u''\ fi(s'', c'')$;
- $u = u''\ ([/1[casesRest\ (w_1\ ...\ w_n)\ /1[if\ e\ then_{el}\ p']\ p'']\ p, s, c, R)$, и $u' = u''\ (/1[casesRest\ (w_1\ ...\ w_n\ e)\ p'']\ p, s, c, R)$ $(/1[assume\ ((not(w_1\ and\ ...\ and\ w_n))\ and\ e)]\ p'\ p, s, c, \emptyset)$;

- $u = u''$ $([/1[casesRest\ (w_1\ ...\ w_n)\ /1[else\ p']\ p'']\ p, s, c, R)$, и $u' = u''\ (/1[assume\ (not(w_1\ and\ ...\ and\ w_n))]\ p'\ p, s, c, \emptyset)$;
- $u = u''\ (/[casesRest]/\ p, s, R)$, и $u' = u''\ fi(s, c)$;
- $u = u''\ (/1[branch\ [p']\ w]\ p, s, c, R)\ fi(s'', c'')$, и $u' = u''\ (/1[branch\ w]\ p, s, c, R)\ (p'\ p, ifSt(s, s''), c, \emptyset)$;
- $u = (/[branch]/\ p, s, c, R)\ fi(s'', c'')$, и $u' = fi(ifSt(s, s''), c)$;
- $u = u''\ (/[branch]/\ p, s, c, R)\ fi(s'', c'')$, и $u' = u''\ fi(s'', c'')$;
- $u = u''\ (/1[branch\ [p']\ w]\ p, s, c, R)$, и $u' = u''\ (/1[branch\ w]\ p, s, c, R)\ (p'\ p, s, c, \emptyset)$;
- $u = u''\ (/[branch]/\ p, s, R)$, и $u' = u''\ fi(s, c)$;
- $u = u''\ (e\ p, s, c, R)\ fi(s'', c'')$, $e \notin PredEl$, $r \in RulSet \setminus R$, $tr((e\ p, ifSt(s, s''), c), (p', s', c'), r) = true$, и $u' = u''\ (e\ p, s, c, R \cup \{r\})\ (p', s', c', \emptyset)$;
- $u = u''\ t_{\in EConf}\ t'_{\in EConf}\ fi(s'', c'')$, и $u' = u''\ t\ fi(s'', c'')$;
- $u = (p, s, c, R)\ fi(s', c')$, и $u' = fi(ifSt(s, s'), c)$;
- $u = u''\ (e\ p, s, c, R)$, $e \notin PredEl$, $r \in RulSet \setminus R$, $tr((e\ p, s, c), (p', s', c'), r) = true$, и $u' = u''\ (e\ p, s, c, R \cup \{r\})\ (p', s', c', \emptyset)$;
- если $u = tr(u''\ (e\ p, s, c, R)$, $e \in PredEl$, $tr_{interp}((e\ p, s, c), (p', s', c')) = true$, и $u' = u''\ (e\ p, s, c, R)\ (p', s', c', \emptyset)$;
- $false$.

При полном переборе с возвратом семантика выражений $/1[branch\ p]$ и $/1[cases\ p]$ изменяется по сравнению с управляемым бэктрекингом. В этом случае элемент $/1[branch\ p]$ называется элементом полного ветвления и используется для выполнения полного перебора вариантов с возвратом. Элемент

$$/1[cases\ /1[if\ e_1\ then_{el}\ p_1]\ ...\ /1[if\ e_n\ then_{el}\ p_n]\ /1[else\ p]]$$

называется дедуктивным условным ветвлением и используется для перебора вариантов $p_1, ..., p_n, p$ с условиями $p_1, ..., p_n, (not\ (p_1\ and\ ...\ and\ p_n))$. Такое ветвление называется дедуктивным, т.к. определяется через выражение $/1[assume\ x]$, которое в этом случае имеет семантику дедуктивного условия продолжения.

ПОСП называется ПОСП с полным перебором, если tr на расширенных конфигурациях является полным перебором с возвратом.

5.12. Версионные ПОСП. Пусть $st, cont \in At$. ПОСП ρ называется версионной ПОСП, если выполнены следующие свойства:

- $base(\rho)$ включает дополнительно конечные множества $VerSym_{\subseteq Sym}$ и $ContFreePreSym_{\subseteq PreSym}$. Элементы множеств $VerSym$ и $ContFreePreSym$ называются версионными символами и свободными от контекста предопределенными символами, соответственно. Если $f \in ContFreePreSym$, то $\tau(f, q) = \tau(f, q')$ для любых q и q', т. е. интерпретация свободного от контекста символа не зависит от контекста вычисления.

- st -$v \in StSym \setminus PreSym$, и $v_{st}(/[f]/) \in Int_{el}$ для любого $f \in VerSym$, и любой $t \in Conf$;

- $cont$ -$v \in StSym \setminus PreSym$, и $v_{count}(/[f]/) \in Int_{el}$ для любого $f \in VerSym$, и любой $t \in Conf$.

Версионные ПОСП используются для моделирования выполнения других ПОСП. В частности, они используются для декларативного описания изменений состояния и контекста в виде формулы, называемой предусловием, в дедуктивных системах с прямым прослеживанием (раздел 5.13). Множество символов состояния моделируемой системы специфицируется множеством версионных символов. Конфигурации трассы моделируемой ПОСП, в которых изменилось состояние или контекст по сравнению с предыдущей конфигурацией, нумеруются последовательными целыми числами. Символ st -v специфицирует наибольший номер конфигурации, в которой, возможно, было модифицировано значение версионного символа. Символ $cont$ -v специфицирует текущий контекст моделируемой ПОСП и называется версионным контекстом.

5.13. Дедуктивные ПОСП с прямым прослеживанием. Дедуктивные ПОСП используются в реализации аксиоматической семантики и логики безопасности программ. Пусть $pre, verCond \in At$. Версионная ПОСП ρ с полным перебором называется дедуктивной ПОСП с прямым прослеживанием, если выполнены следующие свойства:

- $pre \in StSym \setminus PreSym$;

- $verCond \in StSym \setminus PreSym$, и $v_{verCond}() \in im(El_{seq})$ для любой $t \in Conf$;

- $verCond \in BackSym$.

Символ pre, называемый предусловием, специфицирует в виде формулы изменения состояния и контекста моделируемой системы. Символ $verCond$ хранит последовательность порожденных условий корректности.

5.14. ПОСП с обратным проходом. ПОСП ρ называется ПОСП с обратным проходом, если tr переопределяется следующим образом: $tr(t, t') = true$ тогда и только тогда, когда $t.1 = p\ e$ и выполнено любое из двух условий: $e \in PredEl$, и $tr_{interp}(t, t') = true$, или $e \notin PredEl$, и существует $r_{\in RulSet}$ такое, что $tr(t, t', r) = true$. Функция tr с дополнительным аргументом r определяется следующим образом: $tr(t, t', r) = true$ тогда и только тогда, когда r применимо к e в t, $t'.1 = p\ sub(r.3, match_{sub}(e, r, t))$, $t'.2 = t.2$, и $t'.3 = t.3$.

5.15. Онтологические ПОСП. Пусть $EvCont = St \times Int$. Пусть $is, ontSym$, $instSym \in At$. ПОСП ρ называется онтологической ПОСП, если выполнены следующие свойства:

- $\text{-}v\ is\ ontSym \in StSym \setminus PreSym \neq \emptyset$. Символ $f_{\in StSym}$ называется онтологическим символом в q, если $q.1(\text{-}v\ is\ ontSym, q.2)(/[f]/) = true_{el}$. Онтологические символы специфицируют элементы (понятия, атрибуты, отношения и т. п.) онтологии системы, описываемой с помощью ρ;

- $\text{-}v\ is\ instSym \in StSym \setminus PreSym$. Символ $f \in StSym$ называется символом экземпляризации в q, если $s(\text{-}v\ is\ instSym, q)(/[f]/) = true_{el}$. Символы экземпляризации специфицируют связи элементов онтологии с экземплярами;

- $q.1(\text{-}v\ is\ instSym, q.2)(/[+v\ is\ \text{-}v]/) = true_{el}$. Свойство $q.1(+v\ is\ \text{-}v, q.2)(x, y) = true_{el}$ означает, что x — экземпляр понятия y в q;

- Rul определяется конструктором $Rul_{ont} \in El \times El^* \to Rul$ со следующим ограничением: $r.1$ имеет вид $/[x\ is\ y]/$, где $x, y \in At$, для каждого r вида $Rul_{ont}(...)$. Семантика правила $Rul_{ont}(/[x\ is\ y]/, z)$ совпадает с семантикой условного правила $Rul_{cond}(x_{el}, x, z, /[x\ is\ y]/)$.

Рассмотрим примеры онтологических символов:

- $q.1(\text{-}v\ isConcept, q.2)(x) = true_{el}$ означает, что x — понятие в q;

- $q.1(\text{-}v\ isAttributeOf\ \text{-}v, q)(x, y) = true_{el}$ означает, что x — атрибут понятия y в q;

- $q.1(\text{-}v\ isAttributeOf\ \text{-}v\ ofType\ \text{-}v, q.2)(x, y, z) = true_{el}$ означает, что x — атрибут понятия y типа z в (s, i). Типом является некоторое понятие. Например, $q.1(+v\ isAttributeOf\ y\ ofType\ z, q.2)(x, y, int_{el}]) = true_{el}$ означает, что x — целочисленный атрибут понятия y в q.

Рассмотрим примеры символов экземпляризации:

- $q.1(\text{-}v\ of\ +v, q.2)(x, y)$ возвращает значение атрибута x экземпляра y некоторого понятия в q;

- $q.1(\text{-}v\ of\ +v\ of\ \text{-}v, q.2)(x, y, z)$ возвращает значение атрибута x экземпляра y понятия z в q. Этот символ используется, чтобы разрешить конфликт в случае, когда y является экземпляром нескольких понятий, имеющих атрибут x.

5.16. ПОСП с сопоставлением с образцом на последовательностях. ПОСП ρ называется ПОСП с сопоставлением с образцом на последовательностях, если выполнены следующие свойства:

- конструктор Rul переопределяется по первому аргументу: $Rul \in El^+ \times VarSpec^* \times El^* \to Rul$. Таким образом, образцы правил таких ПОСП — непустые последовательности элементов;

- $r_{\in Rul}$ применимо к $p_{\in El^{len(r.1)}}$ относительно $\sigma_{\in Sub}$, если $sub(r.1, \sigma) = p$, $dom(\sigma)$ — множество переменных образца правила r, $\sigma(u) \in El$ для $u_{el} \in r.2$, и $\sigma(u) \in El^*$ для $/[+s\ u]/ \in r.2$;

- функция $match_{sub}$ переопределяется таким образом, что $match_{sub} \in El^+ \times Rul \to Sub$, и, если $match_{sub}(p, r) \neq und$, то $len(p) = len(r.1)$, и r применимо к p относительно $match_{sub}(p, r)$. Функция $match_{sub}$ расширяется на $El^+ \times Rul \times Conf \to Sub$ так же, как и в случае обычных правил ПОСП. Говорят, что r применимо к p в t, если $match_{sub}(p, r, t) \neq und$.

- Функция tr определяется следующим образом: $tr(t, t', r) = true$ тогда и только тогда, когда $t.1 = p''_{\in El^+}\ p$, r применимо к p'', $t'.1 = sub(r.3, match_{sub}(p'', r, t))\ p$, $t.2 = t'.2$, и $t.3 = t'.3$.

6. Комбинированные виды предметно-ориентированных систем переходов. На практике, как правило, используются комбинации базовых видов ПОСП. Рассмотрим некоторые из них.

6.1. Условные ПОСП с переменными истории. Эти ПОСП обладают следующим дополнительным свойством: вместо конструкторов Rul_{cond} и Rul_{hvar} используется комбинированный конструктор $Rul_{cond+hvar} \in El \times VarSpec^* \times El^* \times El \times At^* \to Rul$, где дополнительные компоненты $r.4$ и $r.5$ являются условием и декларацией переменных истории правила r, соответственно.

6.2. Условные ПОСП с частично интерпретированными телами правил. Эти ПОСП обладают следующими дополнительными свойствами:

- отношение применимости правила переопределяется следующим образом: r применимо к e в t, если $match_{sub}(e, r, t) \neq und$, и $val(partVal(sub(r.4, match_{sub}(e, r, t)), (t.2, t.3)), (t.2, t.3)) = true_{el}$;

- tr переопределяется следующим образом: $tr(t, t', r) = true$ тогда и только тогда, когда $t.1 = e\ p$, и существует $r_{\in RulSet}$ такое, что r применимо к e в t, $t'.1 = partVal(sub(r.3, match_{sub}(e, r, t)), (t.2, t.3))\ p$, $t'.2 = t.2$, и $t'.3 = t.3$.

6.3. ПОСП с переменными истории с частично интерпретированными телами правил. В таких ПОСП сначала выполняется подстановка переменных истории, а затем выполняется частичная интерпретация тел правил.

6.4. ПОСП с полным управляемым перебором. Эти ПОСП комбинируют ПОСП с управляемым бэктрекингом и ПОСП с полным перебором. Семантика выражений $/1[branch\ p]$ и $/1[cases\ p]$ определяется как в ПОСП с полным перебором. Если требуется их интерпретация в ПОСП с управляемым перебором, вместо этих выражений используются выражения $/1[branch^*\ p]$ и $/1[cases^*\ p]$.

6.5. ПОСП с управляемым бэктрекингом и со счетчиком. Эти ПОСП обладают следующим дополнительным свойством: $count \in BackSym$.

6.6. ПОСП с управляемым бэктрекингом и с переменными истории. Эти ПОСП обладают следующим дополнительным свойством: $hvar\ i \in BackSym$ для любого $i \in Int$.

6.7. ПОСП с полным перебором и со счетчиком. Эти ПОСП обладают следующим дополнительным свойством: $count \in BackSym$.

6.8. ПОСП с полным перебором и с переменными истории. Эти ПОСП обладают следующим дополнительным свойством: $hvar\ i \in BackSym$ для любого $i \in Int$.

6.9. Версионные ПОСП с переменными истории. Для этих ПОСП выполнены следующие свойства:

- $hvar\ i \in VerSym$ для любого $i \in Int$;
- свойство конечности состояния заменяется на свойство частичной конечности состояния: $\{p \mid s(f, i)(p) \neq und_{el}, f \in StSym, i \in Int \text{ и } p \in El^{arity(f)}\} \setminus \{/[hvar\ i]/ \mid i \in Int\}$ конечно.

7. Предопределенные элементы. В этом разделе определены часто встречающиеся предопределенные элементы.

Будем считать, если не оговорено противное, что $x, y, z \in El$.

7.1. Останов. Пусть $stop \in At$. Элемент $/[stop]/$ называется остановом и имеет следующую семантику: $tr_{interp}(t, t'') = true$, где $t.1.1 = /[stop]/$, тогда и только тогда, когда $t'.1 = seq_{emp}$, $t'.2 = t.2$ и $t'.3 = t.3$.

7.2. Небезопасное завершение. Пусть $fail \in At$. Элемент $/[fail]/$ называется называется небезопасным завершением. Он специфицирует некорректное завершение ПОСП.

Конфигурация t называется небезопасной, если $t.1.1 = /[fail]/$. В противном случае t называется безопасной. Отношение tr должно удовлетворять следующему свойству: если t — небезопасная конфигурация, то t — заключительная конфигурация. Трасса, последний элемент которой — небезопасная конфигурация, называется небезопасной.

7.3. Условие продолжения. Пусть $assume \in At$. Элемент $/1[assume\ x]$ называется условием продолжения и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[assume\ x]\ p$, тогда и только тогда, когда $t'.2 = t.2$, $t'.3 = t.3$, и выполнено первое подходящее свойство:

- если $val(x, (t.2, t.3)) = true_{el}$, то $t'.1 = p$;
- $t'.1 = backtrack_{el}\ p$.

Условие продолжения базируется на элементе $backtrack_{el}$ и используется в ПОСП с управляемым бэктрекингом.

Пусть A^f обозначает $(f, t.3(f))$.

7.4. Условие модификации. Пусть $modify, nextSt \in At$. Элемент $/1[modify\ x]$ называется условием модификации и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[modify\ x]\ p$, тогда и только тогда, когда $t'.3 = t.3$ и выполнено первое подходящее свойство:

- если $val(x, (t.2, t.3, t'.2)) = true_{el}$, и $M = \{A^f \mid f \in StSym, match_{sym}(StSym, e) = (f, a)$, и x содержит $/1[nextSt\ \ e]$ для некоторых e и $a\} \neq \emptyset$, то $t'.1 = p$, и $oDif(t'.2, t.2, M) = true$;
- $t'.1 = backtrack_{el}\ p$, и $t'.2 = t.2$.

Вхождение выражения $/1[nextSt\ p]$ в x указывает на то, что значение выражения $[p]$ определяется в выходном состоянии $t'.2$.

Пусть $EvCont = St \times StCont \times St$. Функция $val \in El \times EvCont \to El$ определяется следующим образом (применяется первое подходящее правило):

- $val(e_{\in im(El_{at}) \cup im(El_{fun})}, q) = e$;
- если $match_{sym}(StSym \cup PreSym, [p]) = (f, a)$, и $f \in StSym \setminus PreSym$, то $val(/1[nextSt\ p], q) = q.3(f, q.2(f))(argVal(a, q))$;
- если $match_{sym}(StSym \cup PreSym, [p]) = (f, a)$, $f \in StSym \cap PreSym$, и $q.3(f, q.2(f))(argVal(a, q)) = und_{el}$, то $val(/1[nextSt\ p], q) = \tau(f, (q.3, q))(argVal(a, q))$;
- если $match_{sym}(StSym \cup PreSym, [p]) = (f, a)$, и $f \in ContSym \cap PreSym$, то $val(/1[nextSt\ p], q) = q.3(f, q.2(f))(argVal(a, q))$;

- если $match_{sym}(StSym \cup PreSym, [p]) = (f, a)$, и $f \in PreSym$, то $val(/1[nextSt\ p], q) = \tau(f, (q.3, q))(argVal(a, q))$;
- $val(/1[nextSt\ p], q) = und_{el}$;
- если $match_{sym}(StSym \cup PreSym, e) = (f, a)$, и $f \in StSym \setminus PreSym$, то $val(e, q) = q.1(f, q.2(f))(argVal(a, q))$;
- если $match_{sym}(StSym \cup PreSym, e) = (f, a)$, $f \in StSym \cap PreSym$, и $q.1(f, q.2(f))(argVal(a, q)) = und_{el}$, то $val(e, q) = \tau(f, (q.1, q))(argVal(a, q))$;
- если $match_{sym}(StSym \cup PreSym, e) = (f, a)$, и $f \in ContSym \cap PreSym$, то $val(e, q) = q.1(f, q.2(f))(argVal(a, q))$;
- если $match_{sym}(StSym \cup PreSym, e) = (f, a)$, и $f \in PreSym$, то $val(e, q) = \tau(f, (q.1, q))(argVal(a, q))$;
- $val(e, q) = und_{el}$.

7.5. Модификация символа. Пусть $::= \in At$. Элемент $[x ::=_{el} y]$ называется модификацией символа и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = [x ::=_{el} y]\ p$, тогда и только тогда, когда $t'.3 = t.3$, и выполнено первое подходящее свойство:

- если $match_{sym}(StSym, x) = (f, a)$, $f \in StSym$, $a' = argVal(a, (t.2, t.3))$, то $oDif(t'.2, t.2, \{A^f\}) = true$, $oDif(t'.2(A^f), t.2(A^f), \{a'\}) = true$, и $t'.2(A^f)(a') = val(y, (t.2, t.3))$;
- $t'.1 = p$ и $t'.2 = t.2$.

Для специального распространенного случай $[x ::= und]2/$ используется сокращение $[x ::=]1/$.

7.6. Условие безопасности. Пусть $assert \in At$. Элемент $/1[assert\ x]$ называется условием безопасности и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[assert\ x]\ p$, тогда и только тогда, когда $t'.2 = t.2$, $t'.3 = t.3$, и выполнено первое подходящее свойство:

- если $val(x, (t.2, t.3)) = true_{el}$, то $t'.1 = p$;
- $t'.1 = /[fail]/$.

7.7. Перезапуск. Пусть $restart \in At$. Элемент $/1[restart\ [x]]$ называется перезапуском и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1.1 = /1[restart\ [x]]$, тогда и только тогда, когда $t'.1 = x$, $t'.2(f, i)(p) = und_{el}$ для любых $f \in StSym \setminus \{count\}$, $i \in Int$

и $p \in El^{arity(f)}$, $t'.2(count, 0)() = 0_{el}$, $t'.2(count, i)() = und_{el}$ для любого $i \in Int \setminus \{0\}$, и $t'.3(f) = 1$ для любого $f \in StSym$.

7.8. Условное ветвление. Пусть $cases \in At$. Элемент вида

$$/1[cases\ /1[if\ e_1\ then_{el}\ p_1]\ ...\ /1[if\ e_n\ then_{el}\ p_n]]$$

или

$$/1[cases\ /1[if\ e_1\ then_{el}\ p_1]\ ...\ /1[if\ e_n\ then_{el}\ p_n]\ /1[else\ p]]$$

называется условным ветвлением и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[cases\ p''']\ p$, тогда и только тогда, когда $t.2' = t.2$, $t.3' = t.3$, и выполнено первое подходящее свойство:

- если $t.1 = /1[cases\ [if\ e\ then\ p']\ p'']\ p$, и $val(e, (t.2, t.3)) = true_{el}$, то $t.1' = p'\ p$;
- если $t.1 = /1[cases\ [if\ e\ then\ p']\ p'']\ p$, и $val(e, (t.2, t.3)) \neq true_{el}$, то $t.1' = /1[cases\ p'']\ p$;
- если $t.1 = /1[cases\ [else\ p']]\ p$, то $t.1' = p'\ p$;
- если $t.1 = /[cases]/\ p$, то $t.1' = backtrack_{el}$.

7.9. Ветвление по образцу. Выражение b вида $/1[if\ e\ var\ w\ then_{el}\ p]$, где $w \in VarSpec^*$, называется ветвью сопоставления с образцом, выражение e — образцом ветви b, переменные образца из w — переменными образца ветви b, и p — телом ветви b. Пусть $body(b)$ обозначает тело ветви b. Пусть $MatchBranch$ — множество всех ветвей сопоставления с образцом. Говорят, что $b_{\in MatchBranch}$ применима к e относительно $\sigma_{\in Sub}$, если $sub(e, \sigma) = e$, $dom(\sigma)$ — множество переменных образца ветви b, $\sigma(u) \in El$ для $u_{el} \in w$, и $\sigma(u) \in El^*$ для $/[+s\ u]/ \in w$.

Правило может быть применимо к одному и тому же элементу относительно разных подстановок, поэтому возможен конфликт при выборе подстановки, соответствующей этому элементу. Мы считаем, что определена функция $match_{sub} \in El \times MatchBranch \to Sub$, разрешающая этот конфликт, такая, что, если $match_{sub}(e, b) \neq und$, то b применима к e относительно $match_{sub}(e, b)$.

Пусть $matchCases \in At$. Пусть $z' = val(z'', (t.2, t.3))$ для $z = /1[{+}{+}v\ z'']$ и $z' = z$ в противном случае. Элемент вида $/1[matchCases\ z\ b_1\ ...\ b_n]$ или $/1[matchCases\ z\ b_1\ ...\ b_n\ /1[else\ p']]$, где $z \in El$ и $b_i \in MatchBranch$, называется ветвлением по образцу и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[matchCases\ z\ p'''\ ]\ p$, тогда и только тогда, когда $t.2' = t.2$, $t.3' = t.3$, и выполнено первое подходящее свойство:

- если $t.1 = /1[matchCases\ z\ b\ p'']\ p$, и $match_{sub}(z', b) \neq und$, то $t.1' = match_{sub}(z', b)(body(b))\ p$;
- если $t.1 = /1[matchCases\ z\ b\ p'']\ p$, и $match_{sub}(z', b) = und$, то $t.1' = /1[matchCases\ p'']\ p$;
- если $t.1 = /1[matchCases\ z\ [else\ p']]\ p$, то $t.1' = p'\ p$;
- если $t.1 = /[matchCases]\ z/\ p$, то $t.1' = backtrack_{el}$.

Элемент z называется спецификатором сопоставляемого выражения.

7.10. Элементы управления контекстом. Пусть $getCont, setCont, newCont \in At$. Пусть $n > 0$, $f_1, ..., f_n \in StSym$, и F обозначает $/[f_1]/\ ...\ /[f_n]/$.

Элемент $/[getCont]/$ возвращает представление текущего контекста и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /[getCont]/\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.3 = t.3$, $oDif(t'.2, t.2, \{A_{val}\}) = true$, и $v'_{val} = contRep(t.3)$.

Элемент $/1[getCont\ F]$ возвращает представление текущего контекста, суженное на $\{/[f_1]/, ..., /[f_n]/\}$, и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[getCont\ F]\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.3 = t.3$, $oDif(t'.2, t.2, \{A_{val}\}) = true$, и $v'_{val} = nar(contRep(t.3), \{/[f_1]/, ..., /[f_n]/\})$.

Элемент $/1[setCont\ x]$ устанавливает текущий контекст равным $val(x, (t.2, t.3))$ и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[setCont\ x]\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.2 = t.2$, и выполнено первое подходящее свойство:

- если $val(x, (t.2, t3)) = El_{fun}(g)$, и $g(/[f]/) \in Int_{el} \cup \{und_{el}\}$ для любого $f \in StSym$, то для любого $f \in StSym$ выполнено первое подходящее свойство:
  - если $g(/[f]/) \neq und_{el}$, то $t'.3(f) = (g(/[f]/))_{int}$;
  - $t'.3(f) = t.3(f)$;
- $t'.3 = t.3$.

Элемент $/[newCont]/$ меняет контекст всех символов из $StSym$ и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /[newCont]/\ p$, тогда и только тогда, когда $t'.1 = p$, $oDif(t'.2, t.2, \{A_{count}\}) = true$, $v'_{count}() = ((v_{count})_{int} + 1)_{el}$, $oDif(t'.3, t.3, StSym) = true$, и $t'.3(f) = (v'_{count}())_{int}$ для любого $f \in StSym$.

Элемент $/1[newCont/F]$ меняет контекст символов $f_1, ..., f_n$ и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[newCont/F]\ p$, тогда и только тогда, когда $t'.1 = p$, $oDif(t'.2, t.2, \{A_{count}\}) = true$, $v'_{count}() = ((v_{count}())_{int} + 1)_{el}$, $oDif(t'.3, t.3, \{f_1, ..., f_n\}) = true$, и $t'.3(f_i) = (v'_{count}())_{int}$ для любого $1 \leq i \leq n$.

7.11. Вычисление элемента. Пусть $elVal \in At$. Элемент $/1[elVal\ x]$ возвращает значение элемента x и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[elVal\ x]\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.3 = t.3$, $oDif(t'.2, t.2, \{A_{val}\}) = true$, и $v'_{val}() = val(x, (t.2, t.3))$.

Следующая группа правил переопределяет элементы $/1[assume\ x]$, $/1[modify\ x]$, $[x ::=_{el} y]$, $/[fail]/$, $/1[assert\ x]$, $/[getCont]/$, $/1[getCont\ p]$, $/1[setCont\ p]$, $/[newCont]/$, $/1[newCont\ p]$, $/[count{+}{+}]/$, $/[newEl]/$ и $/1[elVal\ e]$ для дедуктивных ПОСП с прямым прослеживанием. Поэтому эти элементы при таком их определении называются дедуктивными. В случае, если в дедуктивных ПОСП требуется использовать элементы с исходной семантикой (будем называть их операционными элементами), то вместо них используются элементы $/1[assume^*\ x]$, $/1[modify^*\ x]$, $[x ::=^*_{el} y]$, $/[fail^*]/$, $/1[assert^*\ x]$, $/[getCont^*]/$, $/1[getCont^*\ p]$, $/1[setCont^*\ p]$, $/[newCont^*]/$, $/1[newCont^*\ p]$, $/[count{+}{+}^*]/$, $/[newEl^*]/$ и $/1[elVal^*\ e]$.

7.12. Дедуктивное условие продолжения. Элемент $/1[assume\ x]$ называется дедуктивным условием продолжения и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[assume\ x]\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.3 = t.3$, $oDif(t'.2, t.2, \{A_{pre}\}) = true$, и $v_{pre}() = [v'_{pre}()\ and_{el}\ add_{ver}(x, t.2)]$.

Пусть $EvCont = St$. Пусть $i_f = (q(A_{st})(/[f]/))_{int}$ и $j_f = (q(A_{cont})(/[f]/))_{int}$. Функция $add_{ver} \in El \times EvCont \to El$ расставляет в первом аргументе версии у всех символов, входящих в $VerSym$, и определяется следующим образом (применяется первое подходящее правило):

- $add_{ver}(e_{\in im(El_{at}) \cup im(El_{fun})}, q) = e$;
- если $i, j \in Int$, $match_{sym}(VerSym \cup PreSym, [p]) = (f, a)$, и $f \in ContFreePreSym \setminus VerSym$, то $add_{ver}(/3[ver\ i\ j\ p], q) = /3[addSeq_{ver}(p, q, f)]$;
- если $i, j \in Int$, и $match_{sym}(VerSym \cup PreSym, [p]) = (f, a)$, то $add_{ver}(/3[ver\ i\ j\ p], q) = /3[ver\ i\ j\ addSeq_{ver}(p, q, f)]$;
- если $i, j \in Int$, то $add_{ver}(/3[ver\ i\ j\ p], q) = und_{el}$;
- если $match_{sym}(VerSym \cup PreSym, [p]) = (f, a)$, и $f \in ContFreePreSym \setminus VerSym$, то $add_{ver}([p], q) = [addSeq_{ver}(p, q, f)]$;
- если $match_{sym}(VerSym \cup PreSym, [p]) = (f, a)$, то $add_{ver}([p], q) = /3[ver\ i_f\ j_f\ addSeq_{ver}(p, q, f)]$;
- $add_{ver}(e, q) = und_{el}$.

Функция $addSeq_{ver} \in El^* \times EvCont \to El^*$ определяется следующим образом (применяется первое подходящее правило):

- $addSeq_{ver}(seq_{emp}, q, f) = seq_{emp}$;
- $addSeq_{ver}(/1[\text{-}v\ e]\ p, q, a\ f) = e\ addSeq_{ver}(p, q, f)$;
- $addSeq_{ver}(/1[+v\ e]\ p, q, a\ f) = /1[+v\ add_{ver(e,q)}]\ addSeq_{ver}(p, q, f)$;
- $addSeq_{ver}(e\ p, q, +v\ f) = add_{ver(e,q)}\ addSeq_{ver}(p, q, f)$;
- $addSeq_{ver}(e\ p, q, a\ f) = e\ addSeq_{ver}(p, q, f)$.

Пусть Frm обозначает свойство: $oDif(t'.2, t.2, \{A_{count}, A_{st}, A_{pre}\}) = true$ и $v'_{count}() = ((v_{count}())_{int} + 1)_{el}$.

7.13. Дедуктивное условие модификации. Элемент $/1[modify\ x]$ называется дедуктивным условием модификации и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[modify\ x]\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.3 = t.3$, и выполнено первое подходящее свойство:

- если $M = \{/[f]/ \mid f \in VerSym, match_{sym}(VerSym, e) = (f, a),$ и x содержит $/1[nextSt\ e]$ для некоторых e и $a\} \neq \emptyset$, то Frm истинно, и выполнены следующие свойства:
  - $oDif(v'_{st}, v_{st}, M) = true$;
  - если $e \in M$, то $v'_{st}(e) = v'_{count}()$;
  - $v'_{pre}() = [v_{pre}()\ and_{el}\ add_{ver}(x, (t.2, t'.2))]$;
- $t'.2 = t.2$.

Пусть $EvCont = St \times St$. Пусть $i_f = (q.1(A_{st})(/[f]/))_{int}$, $i'_f = (q.2(A_{st})(/[f]/))_{int}$ и $j_f = (q.1(A_{cont})(/[f]/))_{int}$. Функция $add_{ver} \in El^* \times EvCont \to El^*$ расставляет в первом аргументе версии у всех символов, входящих в $VerSym$, и определяется следующим образом (применяется первое подходящее правило):

- $add_{ver}(e_{\in im(El_{at}) \cup im(El_{fun})}, q) = e$;
- если $i, j \in Int$, $match_{sym}(VerSym \cup PreSym, [p]) = (f, a)$, и $f \in ContFreePreSym \setminus VerSym$, то $add_{ver}(/3[ver\ i\ j\ p], q) = /3[addSeq_{ver}(p, q, f)]$;
- если $i, j \in Int$, и $match_{sym}(VerSym \cup PreSym, [p]) = (f, a)$, то $add_{ver}(/3[ver\ i\ j\ p], q) = /3[ver\ i\ j\ addSeq_{ver}(p, q, f)]$;

- если $i, j \in Int$, то $add_{ver}(/3[ver\ i\ j\ p], q) = und_{el}$;
- если $match_{sym}(VerSym \cup PreSym, [p]) = (f, a)$, и $f \in ContFreePreSym \setminus VerSym$, то $add_{ver}([p], q) = [addSeq_{ver}(p, q, f)]$;
- если $match_{sym}(VerSym \cup PreSym, [p]) = (f, a)$, то $add_{ver}(/1[nextSt\ p], q) = /3[ver\ i'_f\ j_f\ addSeq_{ver}(p, q, f)]$;
- если $match_{sym}(VerSym \cup PreSym, [p]) = (f, a)$, то $add_{ver}([p], q) = /3[ver\ i_f\ j_f\ addSeq_{ver}(p, q, f)]$;
- $add_{ver}(e, q) = und_{el}$.

Пусть n', i_f и j_f обозначают $(v'_{count}())_{int}$, $(v_{st}(/[f]/))_{int}$ и $(v_{cont}(/[f]/))_{int}$, соответственно.

7.14. Дедуктивная модификация символа. Пусть $x = [p']$. Элемент $[x ::=_{el} y]$ называется дедуктивной модификацией символа и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = [x ::=_{el} y]\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.3 = t.3$, и выполнено первое подходящее свойство:

- если $f \in VerSym$ и $match_{sym}(VerSym, x) = (f, a)$, то Frm истинно, и выполнены следующие свойства:
  - $oDif(v'_{st}, v_{st}, \{/[f/]\}) = true$;
  - a' — последовательность аргументов вызова x;
  - $t'.2(A_{st})(/[f/]) = n_{el}$, и $t'.2(A_{st})(/[f/]) = n'_{el}$;
  - если $arity(f) > 0$, то
    $v'_{pre}() = [v_{pre}()\ and_{el}\ /1[oDif\ /[ver\ n\ j_f\ sym\ f]/\ /[ver\ n'\ j_f\ sym\ f]\ a']\ and_{el}$
    $[add_{ver}(/1[nextSt\ p'], t.2, t'.2)\ =_{el}\ add_{ver}(y, t.2)]]$;
  - если $arity(f) = 0$, то
    $v'_{pre}() = [v_{pre}()\ and_{el}\ [add_{ver}([nextSt\ p'], (t.2, t'.2))\ =_{el}\ add_{ver}(y, t.2)]]$;
- $t'.2 = t.2$.

7.15. Операционно-дедуктивная модификация символа. Пусть $x = [p']$. Элемент $[x ::=^{**}_{el} y]$ называется операционно-дедуктивной (или смешанной) модификацией символа и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = [x ::=^{**}_{el} y]\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.3 = t.3$, и выполнено первое подходящее свойство:

- если $f \in VerSym$, и $match_{sym}(VerSym, x) = (f, a)$, то Frm истинно, и выполнены следующие свойства:

- $oDif(v'_{st}, v_{st}, \{/[f/]\}) = true$;
- a' — последовательность аргументов вызова x;
- $t'.2(A_{st})(/[f/]) = n_{el}$, и $t'.2(A_{st})(/[f/]) = n'_{el}$;
- если $arity(f) > 0$, то
  $v'_{pre}() = [v_{pre}()\ and_{el}\ /1[oDif\ /[ver\ n\ j_f\ sym\ f]/\ /[ver\ n'\ j_f\ sym\ f]\ a']\ and_{el}$
  $[add_{ver}(/1[nextSt\ p'], t.2, t'.2)\ =_{el}\ val(y, (t.2, t.3))]]$;
- если $arity(f) = 0$, то
  $v'_{pre}() = [v_{pre}()\ and_{el}\ [add_{ver}([nextSt\ p'], (t.2, t'.2))\ =_{el}\ val(y, (t.2, t.3))]]$.

- $t'.2 = t.2$.

Этот элемент позволяет передавать информацию из операционной семантики при дедуктивной верификации и тем самым комбинировать операционную семантику с дедуктивным выводом.

7.16. Дедуктивное небезопасное завершение. Пусть $A_{verCond}$ обозначает $(verCond, t.3(verCond))$. Элемент $/[fail]/$ называется называется дедуктивным небезопасным завершением и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1.1 = /[fail]/$, тогда и только тогда, когда $t'.1 = seq_{emp}$, $t'.3 = t.3$, $oDif(t'.2, t.2, \{A_{verCond}\}) = true$, $v'_{verCond}() = [p'\ /1[not\ v_{pre}()]]$, где $v_{verCond}() = [p']$.

7.17. Дедуктивное условие безопасности. Элемент $/1[assert\ x]$ называется дедуктивным условием безопасности и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[assert\ x]\ p$, тогда и только тогда, когда

$$t'.1 = /1[cases\ /1[if\ x\ then_{el}\ p]\ /1[else\ /[fail]/]],$$

$t'.2 = t.2$ и $t'.3 = t.3$.

7.18. Дедуктивные элементы управления контекстом. Пусть $n > 0$, $f_1, ..., f_n \in VerSym$ и F обозначает $/[f_1]/\ ...\ /[f_n]/$.

Элемент $/[getCont]/$ возвращает значение версионного контекста и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /[getCont]/\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.3 = t.3$, $oDif(t'.2, t.2, \{A_{val}\}) = true$, и $v'_{val} = El_{fun}(v_{cont})$.

Элемент $/1[getCont\ F]$ возвращает значение версионного контекста, суженное на $\{/[f_1]/, ..., /[f_n]/\}$, и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[getCont\ F]\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.3 = t.3$, $oDif(t'.2, t.2, \{A_{val}\}) = true$, и $v'_{val} = nar(El_{fun}(v_{cont}), \{/[f_1]/, ..., /[f_n]/\})$.

Элемент $/1[setCont\ x]$ устанавливает версионный контекст равным $val(x, (t.2, t.3))$ и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[setCont\ x]\ p$, тогда и только

тогда, когда $t'.1 = p$, $t'.3 = t.3$, $oDif(t'.2, t.2, \{A_{cont}\}) = true$, и выполнено первое подходящее свойство:

- если $val(x, (t.2, t3)) = El_{fun}(g)$ и $g(/[f]/) \in Int_{el} \cup \{und_{el}\}$ для любого $f \in VerSym$, то для любого $f \in SVerSym$ выполнено первое подходящее свойство:
  - если $g(/[f]/) \neq und_{el}$, то $v'_{cont}(/[f]/) = g(/[f]/)$;
  - $v'_{cont}(/[f]/) = v_{cont}(/[f]/)$;
- $v'_{cont} = v_{cont}$.

Элемент $/[newCont]/$ меняет версионный контекст всех символов из $VerSym$ и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /[newCont]/\ p$, тогда и только тогда, когда $t'.1 = p$, $oDif(t'.2, t.2, \{A_{count}, A_{cont}\}) = true$, $v'_{count}() = ((v_{count})_{int} + 1)_{el}$, $oDif(v'_{cont}, v_{cont}, VerSym) = true$, и $v'_{cont}(/[f]/) = v'_{count}()$ для $f \in VerSym$.

Элемент $/1[newCont/F]$ меняет контекст символов $f_1, ..., f_n$ и имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[newCont/F]\ p$, тогда и только тогда, когда $t'.1 = p$, $oDif(t'.2, t.2, \{A_{count, A_{cont}}\}) = true$, $v'_{count}() = ((v_{count}())_{int} + 1)_{el}$, $oDif(v'_{cont}, v_{cont}, \{/[f_1]/, ..., /[f_n]/\}) = true$, и $v'_{cont}(/[f_i]/) = v'_{count}()$ для $1 \leq i \leq n$.

7.19. Дедуктивный инкремент счетчика. Элемент $/[count{+}{+}]/$ имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /[count{+}{+}]/\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.3 = t.3$, Frm истинно, и выполнены следующие свойства:

- $oDif(v'_{st}, v_{st}, \{/[count]/\}) = true$;
- $v'_{count} = v'_{val} = n'_{el}$;
- $v'_{pre}() = [v_{pre}()\ and_{el}\ [/[ver\ n'\ j_{count}\ count]/\ =_{el}\ [/[ver\ i_{count}\ j_{count}\ count]/\ +\ 1]2/]\ and_{el}\ [/[ver\ n'\ j_{val}\ val]/\ =_{el}\ /[ver\ n'\ j_{count}\ count]/]]$.

7.20. Дедуктивный генератор нумерованного элемента. Элемент $/[newEl]/$ имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /[newEl]/\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.3 = t.3$, Frm истинно, и выполнены следующие свойства:

- $oDif(v'_{st}, v_{st}, \{/[count]/, /[val]/\}) = true$;
- $v'_{count} = v'_{val} = n'_{el}$;
- $v'_{pre}() =$
  $[v_{pre}()\ and_{el}$
  $[/[ver\ n'\ j_{count}\ count]/\ =_{el}\ [/[ver\ i_{count}\ j_{count}\ count]/\ +\ 1]2/\ and_{el}$
  $[/[ver\ n'\ j_{val}\ val]/\ =_{el}\ /1[el\ /[ver\ n'\ j_{count}\ count]/]]]$.

7.21. Декларативное вычисление элемента. Элемент $/1[elVal\ x]$ имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[elVal\ x]\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.3 = t.3$, Frm истинно, и выполнены следующие свойства:

- $oDif(v'_{st}, v_{st}, \{/[val]/\}) = true$;
- $v'_{val} = n'_{el}$;
- $v'_{pre}() = [v_{pre}()\ and_{el}\ [/[ver\ n'\ j_{val}\ val]/\ =_{el}\ add_{ver}(x, t.2)]]$.

7.22. Элементы управления предусловием. Пусть $setPre \in At$. Элемент $/1[setPre\ x]$ имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[setPre\ x]\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.3 = t.3$, $oDif(t'.2, t.2, \{A_{pre}\}) = true$, и $v'_{pre}() = add_{ver}(x, t.2)$.

Пусть $newPre \in At$. Элемент $/1[newPre\ x]$ имеет следующую семантику: $tr_{interp}(t, t') = true$, где $t.1 = /1[newPre\ x]\ p$, тогда и только тогда, когда $t'.1 = p$, $t'.3 = t.3$, и выполнены следующие условия:

- $oDif(t'.2, t.2, \{A_{count}, A_{st}, A_{cont}, A_{pre}, A_{val}\}) = true$;
- $v'_{count}() = ((v_{count}())_{int} + 1)_{el}$;
- $oDif(v'_{st}, v_{st}, \{/[f]/ \mid f \in VerSym\}) = true$;
- $oDif(v'_{cont}, v_{cont}, \{/[f]/ \mid f \in VerSym\}) = true$;
- если $f \in VerSym$, то $v'_{st}(/[f]/) = v'_{cont}(/[f]/) = v'_{count}()$;
- $v'_{val}() = v'_{count}()$;
- $v'_{pre}() = add_{ver}(x, t'.2)$.

8. Язык предметно-ориентированных систем переходов Atoment. Язык Atoment предназначен для спецификации ПОСП. Его предыдущие версии рассматривались в [1, 4, 6]. Базовыми конструкциями языка Atoment, с помощью которых описываются элементы объектной системы ПОСП, являются атомы и выражения. Семантика конструкций языка Atoment определяется семантикой соответствующих элементов объектной модели ПОСП.

8.1. Служебные символы. К служебным символам языка Atoment относятся пробельные символы (пробел, переход на новую строку и т. п.), скобки `(` и `)`, символ `"`.

8.2. Атомы. Множество LAt атомов языка Atoment определяется следующим образом:

- если u — произвольная последовательность Unicode-символов, за исключением служебных символов, то $u \in LAt$;

- если u — произвольная последовательность Unicode-символов, в которой каждому вхождению символа " предшествует экранирующий символ \ (если требуется вставить \, то он, как обычно, удваивается), то "u"$\in LAt$.

8.3. Выражения. Пусть t_i — последовательности пробельных символов. Множество Exp выражений языка Atoment определяется следующим образом:

- если $a \in LAt$, то $a \in Exp$;

- если $e_1, ..., e_n \in Exp$ — выражения, то $(t_0 e_1 t_1 ... e_n t_n) \in Exp$. Последовательность t_i может быть пустой только в случае, если она граничит хотя бы с одной скобкой. Выражение вида (t_0) называется пустым выражением.

Далее для простоты будем опускать t_i в определениях, считая, что пробел соответствует последовательности из одного или более пробельных символов. Так, $(t_0 e_1 t_1 ... e_n t_n)$ в упрощенном виде записывается как $(e_1 \ ... \ e_n)$.

Опишем соответствие между базовыми конструкциями языка Atoment и элементами объектной модели ПОСП. Пусть $e_1, ..., e_n \in Exp$ и $a, a_1, ..., a_n \in LAt$.

8.4. Соответствие атомов. Мы считаем, что множества атомов LAt и At языка Atoment и объектной модели ПОСП, соответственно, совпадают.

8.5. Соответствие выражений и элементов. Определим отображение $red \in Exp \to El$ выражений языка Atoment в элементы объектной модели ПОСП:

- $red(a) = El_{at}(a)$;

- если El_a — конструктор выражений, то $red((:a \ e_1 \ ... \ e_n)) = El_a(red(e_1)...red(e_n))$;

- $red((e_1 \ ... \ e_n)) = [e_1...e_n]$. В частности, $red(()) = [\,]$.

Например, выражение `(assume (A and B))` соответствует элементу $/1[assume \ /[A \ and \ B]/]$, а выражение `(A (:seq B C) (:at D))` — элементу $/1[A \ /[B \ C]/ \ D]/1$.

8.6. Правила переходов. Правило перехода $Rul(x, y, z)$ описывается выражением `(if` x' `var` y' `then` z'`)`, где последовательности x', y' и z' специфицируют x, y и z, соответственно. Признаком завершения последовательности x' является атом `var` (если правило содержит переменные образца) или атом `then` (если правило не содержит переменных образца). Признаком завершения последовательности y' является атом `then`.

8.7. Условные правила переходов. Условное правило перехода $Rul_{cond}(x, y, z, u)$ описывается выражением `(if` x' `var` y' `where` u' `then` z'`)`, где последовательности x', y', u' и атом z' специфицируют x, y, u, z, соответственно. Признаком завершения последовательности y' является атом `where`.

8.8. Правила переходов с переменными истории. Правило перехода с переменными истории $Rul_{hvar}(x, y, z, u)$ описывается выражением `(if` x' `var` y' `hvar` u' `then` z'`)`, где последовательности x', y', z' и u' специфицируют x, y, z и u, соответственно. Признаком завершения последовательности x' является атом `var` (если правило содержит переменные образца), атом `hvar` (если правило не содержит переменных образца и содержит переменные истории) или атом `then`. Признаком завершения последовательности y' является атом `hvar` (если правило содержит переменные истории) или атом `then` (если правило не содержит переменных истории). Признаком завершения последовательности u' является атом `then`.

8.9. Условные правила переходов с переменными истории. Условное правило перехода с переменными истории $Rul_{cond+hvar}(x, y, z, u, v)$ описывается выражением `(if` x' `var` y' `where` u' `hvar` v' `then` z'`)`, где выражения x', y', z', u' и v' специфицируют x, y, z, u и v, соответственно. Признаком завершения последовательности y' является атом `where`.

8.10. Онтологические правила переходов. Онтологическое правило перехода $Rul_{ont}(x, y)$ описывается выражением `(if` x' `then` y'`)`, где выражение x' и последовательность y' специфицируют x и y, соответственно.

8.11. ПОСП. ПОСП определяется как выражение вида `(dsts` a `kind` (b) `StSym` (c_1) `PreSym` (c_2) `ContFreePreSym` (c_3) `VerSym` (c_4) `OntSym` (c_5) `InstSym` (c_6) `rules` d`)`, где a — атом, обозначающий имя ПОСП, b — последовательность атомов, описывающих вид ПОСП, $c_1, ..., c_6$ — последовательности выражений вида $/[f]/$, где $f \in Sym$, представляющих символы состояния, предопределенные символы, свободные от контекста предопределенные символы, версионные символы, онтологические символы и символы экземпляризации, соответственно; d — последовательность правил переходов. Последовательность b может включать атомы `backward` и `fullBacktrack`, обозначающие ПОСП с обратным проходом и ПОСП с полным перебором вариантов, соответственно.

ПОСП по умолчанию является условной, неонтологической, неверсионной ПОСП прямого прохода с частично интерпретированными телами правил с возвращаемым значением, с инкрементом счетчика, с генерацией нумерованных элементов, с переменными истории, с управляемым бэктрекингом и с сопоставлением с образцом на последовательностях. Множество предопределенных символов является общим для всех ПОСП, описываемых в Atoment.

9. Заключение. ПОСП — системы переходов специального вида, предназначенные для определения предметно-ориентированных языков, используемых для решения задач разработки семантики языков программирования и проектирования, спецификации, прототипирования и верификации программных систем. ПОСП составляют основу комплексного подхода к решению указанных задач.

В этой работе — первой из цикла работ, посвященных ПОСП, — описаны объектная модель и язык ПОСП. В следующих работах цикла будут рассмотрены примеры применения ПОСП к решению упомянутых выше задач.

Список литературы

1. Ануреев И.С. Типовые примеры использования языка Atoment // Моделирование и анализ информационных систем. 2011. Т. 18. №4. С. 7–20.
2. Непомнящий В.А., Ануреев И.С., Атучин М.М., Марьясов И.В., Петров А.А., Промский А.В. Верификация С-программ в мультиязыковой системе Спектр // Моделирование и анализ информационных систем. 2010. Т. 17. №4, С. 88–100.
3. Anureev I.S. Integrated approach to analysis and verification of imperative programs // Joint NCC&IIS Bulletin, Series Computer Science. 2011. Vol. 32. P. 1–18.
4. Anureev I.S. Introduction to the Atoment language // Joint NCC&IIS Bulletin, Series Computer Science. 2010. Vol. 31. P. 1–16.
5. Anureev I.S., Maryasov I.V., Nepomniaschy V.A. Two-level Mixed Verification Method of C-light Programs in Terms of Safety Logic. Joint NCC&IIS Bulletin, Series Computer Science. 2012. Vol. 34. P. 23–42.
6. Anureev I.S. Program Specific Transition Systems. Joint NCC&IIS Bulletin, Series Computer Science. 2012. Vol. 34. P. 1-21.
7. AsmL: The Abstract State Machine Language. Reference Manual, 2002. http://research.microsoft.com/en-us/projects/asml/
8. Gurevich Y. Abstract State Machines: An Overview of the Project. Foundations of Information and Knowledge Systems (FoIKS): Proc. Third Internat. Symp. Lect. Notes Comput. Sci. 2004. Vol. 2942. P. 6–13.
9. Matthias Anlauff. XasM — An Extensible, Component-Based Abstract State Machines Language. http://xasm.sourceforge.net/XasmAnl00/XasmAnl00.html
10. Parnas D.L. Really Rethinking Formal Methods. Computer. IEEE Computer Society. 2010. Vol. 43 (1). P. 28–34.

UDK: 004.05

Title: Domain-specific transition systems: object model and language

Author(s):

Igor S. Anureev (A.P. Ershov Institute of Informatics Systems)

Abstract: This paper presents the object model and the language of domain-specific transition systems, a new formalism designed for specification and approbation of formal methods which ensure software reliability.

Keywords: domain-specific transition systems, semantics, verification, ontology

УДК: 004.052, 510.643

Название: Верификация мультиагентной модели протокола скользящего окна

Автор:

Гаранина Н.О. (Институт систем информатики им. А.П. Ершова СО РАН)

Аннотация: Многие варианты коммуникационного протокола скользящего окна, который предназначен обеспечивать надежную передачу данных по ненадежным каналам, были специфицированы и верифицированы с использованием различных техник проверки, таких как доказательство теорем, проверка моделей и их комбинаций. В данной работе предлагается рассмотреть спецификацию протокола скользящего окна как мультиагентной модели. Темпоральные и эпистемические свойства протокола сформулированы с помощью логики знаний и времени.

Ключевые слова: мультиагентные системы, коммуникационные протоколы, логика знаний и времени

1. Введение. Протокол скользящего окна [16] — это коммуникационный протокол, обеспечивающий передачу данных от отправителя к получателю по связывающему их каналу. В силу того, что отправитель, получатель и канал коммуникации работают асинхронно и имеют разные мощности ресурсов, такие как скорость передачи данных, объем используемой памяти, скорость обработки информации и т.п., установить надежность протокола является непростой задачей. Для верификации разнообразных вариантов данного протокола применялись различные формальные и неформальные методы: доказательство вручную [11], различные техники автоматической проверки моделей [18], включая проверку с использованием сетей Петри, использование систем автоматического доказательства теорем [4], комбинированные техники верификации [13].

Отдельный интерес представляет подход к верификации данного протокола, трактующий его как протокол, основанный на знаниях. В работе [8] рассмотрено семейство основанных на знаниях протоколов передачи данных. Вручную доказана корректность этого семейства протоколов, и показано, что стандартные протоколы передачи данных, такие как синхронный AUY-протокол [2], AB-протокол [3], протокол Стеннинга [14], являются реализациями соответствующих протоколов, основанных на знаниях. Протоколы, основанные на знаниях, дают более ясное понимание того, что происходит в протоколах, и позволяют обосновать поведение отправителя и получателя, которые рассматриваются как агенты системы. Используя данный подход, авторы работы [15] вручную провели эпистемический анализ варианта модели TCP-протокола, где показали, что их модель протокола корректна, если канал может переупорядочивать и удалять сообщения. Кроме того, доказано, что для каждого агента глубина его знаний о содержании сообщения может достигать порядкового номера этого сообщения, но при этом содержание не является

общеизвестным.

Однако, насколько известно, исследования по проверке (model checking) мультиагентной модели протокола скользящего окна до сих пор не проводились. В работе [12] рассматривается инструмент символьной проверки мультиагентных моделей MCMAS, и в качестве поясняющего примера взята общая задача передачи данных, специальным случаем которой является протокол скользящего окна. Преимущество метода проверки моделей в данном случае, помимо полной автоматизации, заключается в простоте изменения в модели параметров протокола, таких как свойства надежности канала (дублирование, потеря и переупорядочивание сообщений), информация, наблюдаемая агентами отправителем и получателем и т.п. Проверка моделей также позволяет проверить рациональность действий агентов, например, то, что отправитель высылает сообщение (новую порцию сообщений), только если получатель получил предыдущие, или если какое-либо сообщение было потеряно каналом передачи.

В данной работе предлагается специальная мультиагентная модель для протокола скользящего окна — так называемая *интерпретированная система* [7], которая позволяет строго сформулировать работу протокола и выразить его эпистемические и темпоральные свойства, используя подходящую логику. Интерпретированная система — это, в сущности, помеченная система переходов, снабженная средствами определения *поведения и знаний агентов*. Таким образом, отличие предлагаемой мультиагентной модели от многочисленных моделей протокола скользящего окна, использовавшихся при проверке моделей (model checking), заключается в том, что формально заданы действия агентов, и информация, на основе которой они действуют, что позволяет проверить не только свойства живости и безопасности, но и свойства рациональности агентов. Построенную таким образом модель протокола несложно перевести на язык какого-либо инструмента проверки мультиагентных моделей и проверить интересующие свойства. В силу конечности самой модели возникает необходимость прибегнуть к технике независимости от данных [17] и индукции, пользуясь подходом из [13]. Темпоральные и эпистемические свойства протокола сформулированы с помощью логики знаний и времени. Далее в разд. 2 описаны особенности протокола скользящего окна и связанные с ним задачи. В разд. 3 дается понятие интерпретированной мультиагентной системы и логики спецификации. Разд. 4 содержит интерпретированную мультиагентную систему для протокола скользящего окна и формулировку основных свойств. Разд. 5 — заключение.

Работа выполнена при финансовой поддержке Интеграционного гранта Сибирского Отделения Российской Академии Наук № 15/10 “Математические и методологические аспекты интеллектуальных информационных систем”.

2. Протокол скользящего окна. Протокол скользящего окна [14, 16] — это комму-

никационный протокол, который должен гарантировать надежную передачу данных через ненадежную среду коммуникации. Можно рассматривать вариант протокола, в котором получатель и отправитель являются одним процессом, и среда передает сообщения в двух направлениях, доставляя сообщения от одного отправителя-получателя к другому, как в [16], но здесь удобнее считать отправителя и получателя разными компонентами системы, связанными одним двунаправленным каналом, хотя часто рассматривают два канала: канал для передаваемых данных и канал для передачи подтверждения получения.

Отправитель получает данные из входного потока. Эти данные должны быть переправлены через получателя в выходной поток. Данные отправляются получателю через канал, который является ненадежным, т.е. он может изменять пересылаемые данные. Отправитель помечает каждую порцию данных номером, чтобы получатель мог определить правильность получаемых данных. Если с данными все в порядке, получатель подтверждает получение, высылая номер полученных данных либо ожидаемых данных. Также существуют варианты протокола, в которых отправитель высылает номер испорченных данных [16]. Отправитель может заново отослать неподтвержденные данные, которые сохраняются в так называемом *окне передачи*. Получатель хранит полученные данные в *окне приема*, чтобы иметь возможность передать их в правильном порядке в выходной поток.

Таким образом, протокол имеет следующие особенности (список неполон), изменение которых порождает его различные варианты:

- отправитель, получатель и канал действуют синхронно или асинхронно;
- канал может быть двунаправленным, либо могут использоваться два канала в различных направлениях: от отправителя к получателю и обратно;
- размеры окон передачи и приема, как и максимальный номер, используемый в нумерации данных, могут варьироваться;
- ненадежный канал может изменять данные следующим образом: терять, дублировать, переупорядочивать или портить сообщения, а также изменять их по-разному для отправителя и получателя;
- время жизни отправленного сообщения в канале может быть ограничено или неограничено.

Более общим понятием является стандартная задача передачи данных, сформулированная в работе [8] следующим образом:

Рассматриваются два процесса: *отправитель* и *получатель*. У отправителя S имеется входная лента с бесконечной последовательностью элементов данных $X = \langle x_0, x_1, \ldots \rangle$. S читает эти элементы данных и пытается отправить их получателю R, который должен записать их на выходную ленту. Требуется, чтобы выполнялись следующие условия: (1) свойство безопасности — в любой момент времени последовательность данных, записанная R на выходную ленту, являлась префиксом последовательности X; (2) свойство живости — если среда передачи данных удовлетворяет условиям справедливости, то каждый элемент x_i последовательности X будет записан отправителем R.

Протоколы, решающие задачу передачи данных, имеют высокоуровневое представление в виде *протоколов, основанных на знаниях* [8]. Основной особенностью таких протоколов является то, что действия агентов отправителя и получателя по отправке сообщений регулируются их знаниями. А именно, агент-отправитель посылает сообщение, только если он знает, что предыдущее его сообщение было доставлено, или агент-получатель отправляет сообщение об ошибке, только если он "не знает" полученные данные (получил не те данные, которые ожидал). Согласно определению понятия знания в мультиагентных системах [7] в протоколах, основанных на знаниях, выбор следующего действия агента зависит от самой системы, в отличие от стандартных протоколов, где реализация условий передачи сообщений может быть различной для разных версий. Поэтому проверка знаний агентов в стандартных протоколах полезна, чтобы убедиться, что агенты делают то, что необходимо, тогда и только тогда, когда это необходимо. Также можно исследовать зависимость знаний агентов от информации, которая доступна для их наблюдения в системе.

Оставшаяся часть работы посвящена моделированию протокола скользящего окна как мультиагентной интерпретированной системы и формулированию свойств, подлежащих проверке на получившейся модели.

3. Интерпретированная система и логика спецификации. Дадим краткое определение логики спецификации CTL-K$_n$, которая является комбинацией пропозициональной логики знаний PLK [7] и логики деревьев вычислений CTL [5]. Семантика CTL-K$_n$ определена в терминах отношения выполнимости $\models$ в интерпретированных системах, которые являются специальным видом помеченных систем переходов.

Впервые понятие интерпретированных систем было введено в работе [6], однако мы будем пользоваться модифицированным более детальным определением из [9]. Интерпретированные системы могут быть определены следующим образом. Пусть $\{true, false\}$ — булевские константы, $Prop$ — конечный алфавит пропозициональных переменных. Обо-

значим множество агентов как $A = \{1, ..., n\}$, множество локальных состояний каждого агента $i \in A$ как L_i, множество его возможных действий как Act_i, множество локальных состояний и действий среды как L_e и Act_e соответственно. Пусть множество глобальных состояний системы — это $G = L_1 \times ... \times L_n \times L_e$, где каждый элемент $(l_1, ..., l_n, l_e) \in G$ представляет некоторое состояние всей системы. Считаем, что множество протоколов $P_i : L_i \to 2^{Act_i}$ для $i \in A$ представляет поведение каждого агента, и протокол $P_e : L_e \to 2^{Act_e}$ — поведение среды. Пусть $Act = Act_1 \times ... \times Act_n \times Act_e$ — множество совместных действий. Функция переходов $t : G \times Act \to G$ моделирует вычисления в системе. Интуитивно, имея начальное состояние ι, множество протоколов и функцию переходов, можно построить (возможно, бесконечную) структуру, которая представляет все возможные вычисления системы. Для таких структур существуют различные формализмы, однако, поскольку нас интересуют темпорально-эпистемические свойства, то мы будем пользоваться следующим формальным определением интерпретированной системы.

Определение 1. *(интерпретированной системы)*
Для данного множества агентов $A = \{1, ..., n\}$ и множества пропозициональных переменных $Prop$ *интерпретированная система* — это пара $M = (\mathcal{K}, \mathcal{V})$, причем $\mathcal{K} = (G, T, \sim_1, ..., \sim_n, \iota)$, где

- G — множество глобальных состояний системы;
- $\iota \in G$ — начальное состояние;
- $T \subseteq G \times G$ — всюду определенное бинарное отношение на G такое, что $(w, w') \in T$ тогда и только тогда, когда $t(w, act) = w'$ для некоторого $act \in Act$;
- $\sim_i \subseteq G \times G$ $(i \in A)$ — эпистемическое отношение неразличимости для каждого агента $i \in A$ такого, что $w \sim_i w'$ тогда и только тогда, когда $l_i(w') = l_i(w)$, где функция $l_i : G \to L_i$ возвращает локальное состояние агента i из глобального состояния w;
- $\mathcal{V} : G \longrightarrow 2^{Prop \cup \{true, false\}}$ — функция означивания для пропозициональных переменных $Prop$ такая, что $true \in \mathcal{V}(w)$ для всех $w \in G$. $\mathcal{V}$ сопоставляет каждому состоянию множество пропозициональных переменных, истинных в этом состоянии.

Синтаксис логики спецификации CTL-K$_n$ определяется следующим образом.

Определение 2. *(синтаксиса CTL-K$_n$)*
Синтаксис логики CTL-K$_n$ состоит из формул, построенных из булевских констант, пропозициональных переменных, связок $\neg$, $\wedge$, $\vee$, и следующих модальностей. Пусть $i \in \{1, ..., n\}$, φ и ψ будут формулами. Тогда формулами с модальностями являются

- модальности знаний: $K_i\varphi$ и $S_i\varphi$ (читается как ‘агент i знает’ и ‘агент i предполагает’);
- модальности времени: $\mathbf{AX}\varphi$, $\mathbf{EX}\varphi$, $\mathbf{AG}\varphi$, $\mathbf{EG}\varphi$, $\mathbf{AF}\varphi$, $\mathbf{EF}\varphi$, $\mathbf{A}\varphi\mathbf{U}\psi$, и $\mathbf{E}\varphi\mathbf{U}\psi$ (читается как **A** — ‘для всякого будущего’, **E** — ‘для некоторого будущего’, **X** — ‘на следующем шаге’, **G** — ‘всегда’, **F** — ‘когда-нибудь’, **U** — ‘пока’).

Семантика CTL-K$_n$ следует семантике этих логик.

Определение 3. *(семантики CTL-K_n)*
Отношение выполнимости $\models_M$ между интерпретированными системами, состояниями и формулами задается индуктивно согласно структуре формулы. Для булевских констант, пропозициональных переменных, связок и модальностей времени отношение выполнимости стандартно. Для модальностей знаний выполнимость определяется следующим образом. Пусть $w \in G$, $i \in \{1, ..., n\}$, φ — формула, тогда

- $w \models_M (K_i\varphi)$ если и только если для каждого w': $w \sim_i w'$ влечет $w' \models_M \varphi$;
- $w \models_M (S_i\varphi)$ если и только если для некоторого w': $w \sim_i w'$ и $w' \models_M \varphi$.

Семантика формулы φ в интерпретированной системе M — это множество всех состояний M, в котором выполняется данная формула: $M(\varphi) = \{w \mid w \models_M \varphi\}$.

Семантика конструкций CTL стандартна и интуитивно понятна из пояснений к описанию их синтаксиса. Неформально семантика конструкции $K_i\varphi$ определяется следующим образом: агент знает какой-либо факт φ, если этот факт верен во всех состояниях системы, в которых агент имеет одну и ту же информацию, т.е. его локальные состояния одинаковы.

4. Интерпретированная система для протокола скользящего окна. Вообще говоря, в задаче передачи данных значения транспортируемых элементов данных могут принадлежать бесконечной области определения. Поэтому, как и в работе [13], мы будем использовать то, что протокол скользящего окна обладает свойством независимости от данных [17], и, следовательно, далее будем считать, что данные в нашей модели совпадают с присвоенными им номерами. Нужно заметить, что в таком случае невозможна “мутация” данных в канале передачи, то есть что элемент данных прибывает под чужим номером.

Кроме того, наша модель является параметрической относительно размеров окон приема и передачи, а также времени жизни сообщений в канале. Можно показать с помощью индукции, что проверямые свойства предлагаемой модели сохраняются с соответствующим изменением этих параметров, однако это выходит за рамки данной работы.

Вариант протокола скользящего окна, исследуемый в данной работе, имеет следующие характеристики:

- отправитель, получатель и канал действуют асинхронно;
- канал является двунаправленным;
- передаваемые данные нумеруются числами от 0 до 3;
- размер окон передачи и приема равен 2;
- время жизни отправленного сообщения в канале ограничено;
- канал может изменять данные следующим образом: терять, дублировать, переупорядочивать сообщения отправителя и терять и дублировать сообщения получателя.

Чтобы задать интерпретированную систему переходов, нам нужно определить (1) множество агентов и среду; (2) локальные состояния агентов и среды; (3) действия агентов и среды; (4) протоколы агентов и среды; (5) функцию переходов для агентов и среды; (6) начальные состояния и (7) означивание пропозициональных переменных. Для краткости изложения мы будем пользоваться следующими соглашениями. Пусть i — номер элемента данных из диапазона порядковых номеров данных $i \in [0..3]$, а операции $\oplus, \ominus$ — сложение и вычитание по модулю 4.

(1) Множество агентов и среда

Множество агентов A равно $\{S, R\}$, где S — отправитель и R — получатель. Средой E будет являться канал передачи данных.

(2) Локальные состояния

Определяются значениями соответствующих локальных переменных. Дополнительно выделим *наблюдаемые переменные агента*, которые влияют на его действия, но недоступны для изменения данным агентом.

Получатель R. $L_R = \prod_{i\in[0..3]} r_i \times win_r \times ack$, где

• $r_i \in \{0, 1\}$ — массив записываемых данных ($r_i = 1$ — элемент данных с номером i записан, $r_i = 0$ — элемент данных с номером i не записан), $i \in [0..3]$,

• $win_r \in [0..3]$ — указатель на верхнюю границу окна приема,

• $ack \in [0..3] \cup \{all, err\}$ — подтверждение для элемента данных с номером i или для всех данных из окна приема, либо ошибка.

Отправитель S. $L_S = \prod_{i\in[0..3]}(s_i \times timer_i) \times win_s$, где

• $s_i \in \{-1, 0, 1\}$ — массив передаваемых данных ($s_i = -1$ — элемент данных с номером i не передан, $s_i = 0$ — элемент данных с номером i передан, $s_i = 1$ — получено подтверждение для элемента данных с номером i), $i \in [0..3]$,

• $timer_i \in [0..9]$ — время ожидания подтверждения элемента данных с номером i, $i \in [0..3]$,

• $win_s \in [0..3]$ — указатель на верхнюю границу окна передачи.

Среда (канал передачи) E. $L_E = ack_e \times data_e \times event$, где

• $ack_e \in [0..3] \cup \{all, err\}$ — подтверждение от получателя в канале,

• $data_e \in [0..3] \cup \{err\}$ — номер элемента данных от отправителя в канале,

• $event \in \{arrival, none\}$ — события в системе.

Наблюдаемые переменные. $Obs_R = \{data_e, event\}$, $Obs_S = \{ack_e, event\}$.

Таким образом, множество глобальных состояний определяется как $G = L_R \times L_S \times L_E$ и определяется значениями 18-ти локальных переменных.

(3) Действия

Получатель R. $Act_R = \{write_i, clearR_i, moveWin_r, ackn\}$, где

• $write_i$ — записать элемент данных с номером i, $i \in [0..3]$,

• $clearR_i$ — удалить элемент данных с номером i, $i \in [0..3]$,

• $moveWin_r$ — передвинуть окно,

• $ackn$ — подтвердить получение элемента данных.

Отправитель S. $Act_S = \{send_i, ack_i, clearS_i, moveWin_s, startTime_i, Time_i, stopTime_i\}$, где

• $send_i$ — послать элемент данных с номером i, $i \in [0..3]$,

• ack_i — обработать подтверждение для элемента данных с номером i, $i \in [0..3]$,

• $clearS_i$ — удалить элемент данных с номером i, $i \in [0..3]$,

• $moveWin_s$ — передвинуть окно,

• $startTime_i, Time_i, stopTime_i$ — запустить, обновить и остановить таймер ожидания подтверждения для элемента данных с номером i, $i \in [0..3]$.

Среда (канал передачи) E. $Act_E = \{dlvAck, dlvData, doEvent\}$, где

• $dlvAck$ — доставить подтверждение элемента данных от получателя,

• $dlvData$ — доставить элемент данных от отправителя,

• $doEvent$ — породить событие системы.

(4) Протоколы

В нашей мультиагентной интерпретированной системе протокола скользящего окна протоколы действий агентов и среды определяются тривиально следующим образом: в своем локальном состоянии агент выполняет те действия, которые могут быть выполнены. Действия, которые агент может выполнить, задаются с помощью следующей функции переходов.

Определим функцию переходов локальным образом: для действий каждого агента и среды по отдельности. Определяются пред- и постусловия: $(pre_1, ..., pre_{18}) \times acts \longrightarrow (post_1, ..., post_{18})$, где для каждого $j \in [1..18]$ (1) pre_j — предусловие для соответствующей переменной; если предусловие отсутствует, то это означает, что переменная может принимать любое значение; (2) $acts$ — действия, которые соответствующий агент может выполнить в этом состоянии (3) $post_j$ — постусловие для соответствующей переменной;

если постусловие отсутствует, то это означает, что переменная имеет то же значение, что и в предусловии. Отметим, что агенты и среда в предусловии явно имеют значения только своих локальных и наблюдаемых переменных и свои действия, а в постусловии могут меняться только значения их локальных переменных. Для краткости примем следующие сокращения для обозначения множеств состояний в предусловиях для всех $i \in [0..3]$:

$arriv_i = (event = arrival \wedge r_i = 0 \wedge data_e = i \wedge (win_r = i \vee win_r = i \oplus 1))$ —

прибытие элемента данных с номером i в диапазоне окна;

$start_i = (s_i = -1 \wedge (win_s = i \vee win_s = i \oplus 1))$ —

элемент данных с номером i может быть отправлен;

$arrivAck_i = (event = arrival \wedge s_i = 0 \wedge ack_e = i)$ —

прибыло подтверждение для элемента данных с номером i.

(5) Функции переходов

Получатель R.

• $(arriv_i) \times write_i \times ackn \longrightarrow (r_i = 1, ack = data_e)$ — отметить, что прибыл элемент данных с номером i, и отправить подтверждение;

• $(r_i = 1 \wedge r_{i\oplus 1} = 0 \wedge win_r = i \oplus 1) \times clearR_i \times moveWin_r \longrightarrow (r_i = 0, win_r = win_r \oplus 1)$ — отметить, что элемент данных с номером i отправлен в выходной поток, и сдвинуть окно приема;

• $(r_i = 1 \wedge r_{i\oplus 1} = 1 \wedge win_r = i \oplus 1) \times clearR_i \times moveWin_r \times ackn \longrightarrow (r_i = 0, r_{i\oplus 1} = 0, win_r = win_r \oplus 2, ack = all)$ — отметить, что элементы данных с номерами i и $i \oplus 1$ отправлены в выходной поток, и сдвинуть окно приема.

Отправитель S.

• $(start_i) \times send_i \times startTime_i \longrightarrow (s_i = 0, timer_i = 1)$ — если элемент данных не отправлен из окна передачи, то отметить, что элемент данных с номером i отправлен получателю, и запустить его таймер;

• $(arrivAck_i) \times ack_i \longrightarrow (s_i = 1)$ — отметить, что пришло подтверждение для элемента данных с номером i;

• $(timer_i = 9) \times clearS_i \times stopTime_i \longrightarrow (s_i = -1, timer_i = 0)$ — когда время ожидания подтверждение для элемента данных с номером i вышло, то отметить его как неотправленное, и обнулить его таймер;

• $(s_i = 1 \wedge s_{i\ominus 1} = 0 \wedge win_s = i) \times stopTime_i \longrightarrow (timer_i = 0)$ — если получено подтверждение для элемента данных с номером i из старшего элемента окна передачи, то остановить его таймер;

• $(s_i = 1 \wedge s_{i\oplus 1} = 0 \wedge win_s = i \oplus 1) \times moveWin_s \times stopTime_i \longrightarrow (s_i = -1, win_s = win_s \oplus 1, timer_i = 0)$ — если получено подтверждение для элемента данных с номером i, то сдвинуть окно передачи на 1;

• $((s_i = 1 \wedge s_{i\oplus 1} = 1 \vee ack_e = all) \wedge win_s = i \oplus 1) \times moveWin_s \times stopTime_i \times stopTime_{i\oplus 1} \longrightarrow (s_i = -1, s_{i\oplus 1} = -1, win_s = win_s \oplus 2, timer_i = 0, timer_{i\oplus 1} = 0)$ — если получены подтверждения для элементов данных с номерами i и $i \oplus 1$, то сдвинуть окно передачи на 2;

• $(timer_i > 0 \wedge timer_i < 9) \times Time_i \longrightarrow (timer_i = timer_i + 1)$ — обновить время ожидания подтверждения для элемента данных с номером i, если таймер для него запущен.

Среда (канал передачи) E.

• $(true) \times dlvAck \times dlvData \times doEvent \longrightarrow (ack_e = \{ack_e, ack, err\}$[1]$, data_e = \{0..3, err\}, event = \{arrival, none\})$ — дублировать, доставить и испортить подтверждение; доставить, дублировать, переставить и испортить данные; породить событие доставки либо пустое действие. Константа $true$ в предусловии действий среды означает, что она выполняет эти действия в любом состоянии нашей модели.

Пусть множество локальных функций переходов — это $T_l = \{t_1, ..., t_n\}$, где $t_j = Pre_j \times acts_j \longrightarrow Post_j$. Тогда тотальная функция переходов t определяется множеством глобальных функций $t_{\theta(kn)} = \bigcap_{j\in\theta(kn)} Pre_j \times \prod_{j\in\theta(kn)} acts_j \longrightarrow \bigcap_{j\in\theta(kn)} Post_j$, где $\theta(kn)$ — сочетание из n по k. Заметим, что некоторые глобальные предусловия $\bigcap_{j\in\theta(kn)} Pre_j$ пусты, и в этом случае никаких действий не выполняется. По определению локальных функций t_j, если глобальное действие $\prod_{j\in\theta(kn)} acts_j$ выполняется, то глобальное постусловие $\bigcap_{j\in\theta(kn)} Post_j$ всегда непусто.

(6) Начальные состояния

• $\iota = (r_0 = 0, r_1 = 0, r_2 = 0, r_3 = 0, win_r = 1, ack = err, s_0 = -1, timer_0 = 0, s_1 = -1, timer_1 = 0, s_2 = -1, timer_2 = 0, s_3 = -1, timer_3 = 0, win_s = 1, ack_e = err, data_e = err, event = none)$ — в начальном состоянии нет ни принятых, ни отправленных данных, отправитель готов отправить элементы данных с номерами 0 и 1, получатель готов их принять, данные в канале и подтверждения пока ошибочны, и никаких событий не происходит.

(7) Означивание пропозициональных переменных

Определим означивание следующих пропозициональных переменных следующим образом. Каждой переменной сопоставляется множество состояний, в которых выполняются условия на значения соответствующих локальных переменных, в то время как значения остальных локальных переменных произвольны.

• $DifWindow = \neg(win_r = win_s \wedge win_r = win_s \oplus 1)$ — окна приема и передачи не пересекаются, то есть номера ожидаемых элементов данных не совпадают с номерами посылаемых элементов данных.

• $RecEmpty = (r_{win_r} = 0 \wedge r_{win_r\oplus 1} = 0)$ — окно приема пусто.

[1]Запись $var = \{val_1, ..., val_n\}$ означает недетерминированный выбор значения переменной var из множества $\{val_1, ..., val_n\}$.

• $GoodData_i = (event = arrival \wedge s_i = 0 \wedge data_e = i)$ — доставлен высылавшееся элемент данных с номером i.

• $GoodAck_i = (event = arrival \wedge s_i = 0 \wedge (ack_e = i \vee ack_e = all))$ — получено подтверждение, что доставлено высылавшийся элемент данных с номером i.

Используя вышеизложенное, можно построить интерпретированную систему протокола скользящего окна M_{SW}. Имея данную строгую формализацию протокола можно проверять его свойства с помощью подходящих инструментов проверки мультиагентных моделей. Выразим интересующие нас свойства с помощью логики CTL-K$_n$.

Свойство безопасности заключается в том, чтобы получатель передавал данные в выходной поток в том порядке, в котором он их получил. В случае нашей модели для этого необходимо и достаточно, чтобы в его окно приема не попадали неправильные данные. Поскольку по определению локальных функций перехода получателя ошибочные данные вида err он просто никогда не записывает, то остается лишь проверить, что он не записывает элементы данных с номерами, не попадающими в его окно приема. В таком случае формула безопасности выглядит следующим образом: $Safety = \mathbf{AG}DifWindow \rightarrow \mathbf{AX}RecEmpty$. Заметим, что канал не может переупорядочивать подтверждения, в отличие от данных.

Свойство живости, т.е., что при условии справедливой работы канала получатель рано или поздно получит то, что отправлял отправитель, можно выразить формулой $Live_R = \bigwedge_{i\in[0..3]} \mathbf{AG}(s_i = 0 \rightarrow \mathbf{AXAF}r_i = 1)$. Условие справедливости в данном случае заключается в том, что канал не может портить сообщения отправителя бесконечно долго: $Fair_S = \bigwedge_{i\in[0..3]} \mathbf{AGAF}GoodData_i$. Отметим, что в случае протокола скользящего окна необходимо также условие живости для отправителя, т.е. то, что он рано или поздно получит подтверждение, что отправленные им данные получены, иначе он никогда не сдвинет окно и не начнет отправлять очередную порцию данных. В нашей модели это формула $Live_S = \bigwedge_{i\in[0..3]} \mathbf{AG}(r_i = 1 \rightarrow \mathbf{AXAF}s_i = -1)$, и она должна выполняться при аналогичном условии справедливости для подтверждений получателя $Fair_R = \bigwedge_{i\in[0..3]} \mathbf{AGAF}GoodAck_i$.

Для предложенной модели свойство рациональности действий отправителя можно записать формулой $Rational_S = K_S(K_R(r_i = 1) \vee win_s = win_r \ominus 1) \rightarrow \mathbf{AX}(timer_i = 0)$, которая выражает, что, если отправитель знает, что получатель получил данные либо уже сдвинул окно приема, то он на следующем шаге останавливает таймер этих данных и не пытается послать их снова. Более естественно это свойство могло быть определено формулой логики знаний и времени с модальностями прошлого CTL$_{\mathrm{P}}$K [9]: $Rational_S = K_S\mathbf{AH}K_R(r_i = 1) \rightarrow \mathbf{AX}(timer_i = 0)$, т.е., что, если получатель знает, что отправитель когда-то в прошлом получил данные, то он останавливает таймер, однако эта логика здесь

не рассматривается.

5. Заключение. В данной работе была предложена мультиагентная интерпретированная система протокола скользящего окна, и сформулированы основные свойства этого протокола в логике знаний и времени CTL-K_n, а именно, свойства безопасности, живости и рациональности отправителя. Выполнимость этих свойств в ближайшем будущем проверить с помощью инструментов проверки мультиагентных моделей VerICS[10], MCMAS[12] и Examiner[1]. Также планируется построить мультиагентные модели и проверить эпистемические свойства других коммуникационных протоколов.

Список литературы

1. Гаранина Н.О. Проверка моделей распределенных систем с помощью аффинного представления данных. // Моделирование и анализ информационных систем. 2010. Т. 17, №4. С. 52-59.
2. Aho A. V., Ullman J. D., Yannakakis M. Modeling communication protocols by automata. // 20th Symp. on Foundations of Computer Science: Proc. San Juan, Puerto Rico: IEEE Computer Society, 1979. P. 267–273.
3. Bartlett K. A., Scantlebury R. A., and Wilkinson P. T. A note on reliable full-duplex transmission over half-duplex links. // Communications of the ACM. 1969. V. 12. P. 260–261.
4. Chkliaev D., Hooman J., de Vink E. P. Verification and Improvement of the Sliding Window Protocol. // Lecture Notes in Computer Science. 2003. V. 2619. P. 113-127.
5. Clarke E.M., Grumberg O., Peled D. Model Checking. MIT Press, 1999. 324 p.
6. Fagin R., Halpern J.Y. Modelling knowledge and action in distributed systems // Distributed Computing. 1989. V. 3, I. 4. P. 159-177.
7. Fagin R., Halpern J.Y., Moses Y., Vardi M.Y. Reasoning about Knowledge. MIT Press, 1995. 519 p.
8. Halpern J. Y., Zuck L. D. A little knowledge goes a long way: knowledge-based derivations and correctness proofs for a family of protocols // Journal of the ACM. 1992. V. 39(3). P. 449–478.
9. Kacprzak M., Lomuscio A., Penczek W. Verification of Multiagent Systems via Unbounded Model Checking // The Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '04): Proc. IEEE Computer Society Washington, DC, USA, 2004. V. 2. P. 638-645.
10. Kacprzak M., Nabialek W., Niewiadomski A., Penczek W., Pólrola A., Szreter M., Wozna

B., Zbrzezny A. VerICS 2007 — a Model Checker for Knowledge and Real-Time // Fundamenta Informaticae. 2008. V. 85, Num. 1-4. P. 313-328.

11. Knuth D.E. Verification of link-level protocols // BIT. 1981. V. 21. P. 31–36.
12. Lomuscio A., Raimondi F. MCMAS: A Model Checker for Multi-agent Systems // Lecture Notes in Computer Science. 2006. V. 3920. P. 450-454.
13. Stahl K., Baukus K., Lakhnech Y., Steffen M. Divide, abstract, and model-check // Lecture Notes in Computer Science. 1999. V. 1680. P. 57–76.
14. Stenning N.V. A data transfer protocol // Computer Networks. 1976. V.1. P. 99–110.
15. Stulp F. and Verbrugge R. A knowledge-based algorithm for the Internet protocol (TCP) // Bulletin of Economic Research. 2002. V. 54(1). P. 69–94.
16. Tanenbaum A.S. Computer Networks (Third Edition). Prentice-Hall International, 1996. 933 p.
17. Wolper P. Expressing interesting properties of programs in propositional temporal logic // The 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages: Proc. ACM New York, NY, USA, 1986. P. 184-193.
18. Zhao Y., Yang Z., Xie J., Liu Q. Formal Model and Analysis of Sliding Window Protocol Based on NuSMV // Journal of Computers. 2009. V. 4, N. 6. P. 519-526.

UDK: 004.052, 510.643

Title: Model Checking Multi-agent Sliding Window Protocol

Author: Garanina N.O. (A.P. Ershov Institute of Informatics Systems SB RAS)

Abstract: Many variants of the communication sliding window protocol were specified and verified using various techniques like theorem proving, model checking and their combinations. In this paper we consider a specification of the Sliding window protocol as a multi-agent affine model. Temporal and epistemic properties of the protocol are expressed in Logic of Knowledge and Time.

Keywords: multiagent systems, communication protocols, logic of knowledge and time

УДК: 004.434

Название: Языковые средства организации вычислений в области биоинформатики

Автор(ы):

Грехов Г.А. (ООО «НЦИТ Унипро»),

Скопин И.Н. (Институт вычислительной математики и математической геофизики СО РАН, Новосибирский государственный университет)

Аннотация: В статье описан подход к созданию вычислительных платформ на примере UGENE Workflow Designer. Эта платформа, основанная на внутреннем языке программирования UWL, демонстрирует, как, следуя принципам внутренней модели платформы и синтаксиса языка программирования, сохранять систему целостной и удобной для развития и поддержки.

Система UGENE Workflow Designer предназначена для решения задач биоинформатики, а язык UWL является DSL языком этой области. В статье описаны и обоснованы возможности использования этой вычислительной платформы в качестве основы переиспользования ее средств в некоторых других прикладных областях.

Ключевые слова: языки программирования, DSL, конвейерные вычисления, *UGENE*, биоинформатика

1. Введение. Современная прикладная наука богата различного рода вычислительными задачами, и ученым ежедневно требуется вести те или иные расчеты. Кроме того, работа во многих областях знаний подразумевает не просто ведение сложных расчетов, но и многообразие решаемых вычислительных задач. Примером такой области является биоинформатика.

Исследователю этой области в своей работе требуется использовать множество программных инструментов, каждый из которых нацелен на решение конкретной задачи. То есть нужно заниматься поиском этих инструментов и осваивать их интерфейс. Зачастую найденная программа может работать в операционной системе, отличной от системы пользователя, что создает дополнительные трудности для исследователя прикладной области. Во многих случаях возникает необходимость решать последовательность вычислительных задач, решение каждой из которых использует результаты, полученные в ходе работы предыдущих. Другими словами, требуется создавать вычислительные конвейеры для серий сложных однотипных задач над различными входными данными.

Решением указанных проблем могут стать системы, представляющие собой вычислительные платформы, которые предлагают необходимый набор инструментов с единым интерфейсом, а также допускают адаптивное расширение самим пользователем.

В данной статье обсуждается такая система, предназначенная для исследователей в области биоинформатики, называемая UGENE Workflow Designer [1]. Она является частью большей системы Unipro UGENE [6], свободного программного обеспечения для работы молекулярного биолога. Workflow Designer является подсистемой, отвечающей именно за вычислительную часть вопроса, в то время как обработку результатов, полученных в ходе вычислений, можно совершать при помощи средств визуализации и редактирования UGENE. Вся система разрабатывается на языке C++ с использованием фреймворка Qt [7], что позволяет ей функционировать в операционных системах семейства Windows, UNIX и Mac OS.

Workflow Designer развивается во многом по мере поступления запросов от пользователей, и изначально эта система подвергалась экстенсивному развитию, то есть простому наращиванию функциональности в ходе удовлетворения этих запросов. Вычислительные схемы (конвейеры) сохранялись в бинарных форматах файлов, а задача разработки средств управления потоками данных в конвейерах, просмотра временных результатов не ставилась. Впоследствии стало очевидным, что без решения этой задачи система превратилась бы в чрезмерно тяжелую как для использования, так для сопровождения и дальнейшего развития.

Было принято решение рассматривать UGENE Workflow Designer как среду разработки на dataflow-языке программирования [4], а вычислительные схемы — как программы на этом языке. Предусматривается, что такое развитие должно удовлетворять следующему требованию: прежде чем вносить какое-либо улучшение в Workflow Designer, нужно проанализировать, как это улучшение будет отражено в языке, не нарушая его особенностей и избегая дублирования средств. Выполняется и другое требование, повышающее гибкость системы: построенные схемы сохраняются в файлы не в бинарном, а в текстовом виде. Текст схемы — это тоже программа на предлагаемом языке программирования, но написанная с помощью его нового текстового синтаксиса. Таким образом, появились два эквивалентных конкретных синтаксиса (графический и текстовый) одного языка программирования, который стал называться UGENE Workflow Language или, сокращенно, UWL.

UWL является DSL [5] языком. Область его использования определяется требованием решения задач биоинформатики и молекулярной биологии. Специфика этих областей выражается в основном в наборе вычислительных инструментов (стандартной библиотеке), который предоставлен пользователю. Другой особенностью языка, пришедшей из предметной области, является отсутствие циклов в вычислительных схемах, т.к. в решаемых

задачах они не требуются. С учетом сказанного выше платформу языка UWL можно переиспользовать и в некоторых других областях.

2. Вычислительная модель и абстрактный синтаксис. Язык UWL, как и любой другой язык программирования, имеет абстрактный синтаксис, который описывает его вычислительные возможности, т.е. модель вычислений, и конкретный синтаксис, определяющий, как эта модель представляется пользователям. Как уже упоминалось, UWL предлагает два варианта конкретных синтаксисов: текстовый, обеспечивающий традиционные возможности записи программ, и графический, конструкции которого изображаются как элементы схем.

Абстрактный синтаксис [5] предназначен для внутреннего представления программ и характеризуется, в основном, вычислительной моделью языка программирования. Конкретные синтаксисы [5] предназначены для программистов, то есть пользователей UGENE Workflow Designer, и непосредственно позволяют конструировать (программировать) конвейеры вычислений.

2.1. Вычислительные элементы и их порты. Описываемый язык UWL является dataflow-языком [4], поэтому основную часть его вычислительной модели составляют вычислительные элементы и соединяющие их каналы связей. Вычислительный элемент (worker или actor) — это некоторая многократно исполняющаяся программа. Это может быть реализация какого-либо биологического алгоритма, или программа, считывающая или записывающая данные на жесткий диск или удаленную базу данных, или какой-либо служебный элемент, управляющий потоками данных.

Используются вычислительные элементы трех типов:

1) чистые генераторы данных (Data readers);
2) вычислители или генераторы данных в контексте других данных (Computers);
3) писатели данных (Data writers);
4) служебные (Dataflow elements).

В UWL наиболее распространенными элементами, естественно, являются вычислители, так как они предоставляют всю вычислительную функциональность.

Каждый вычислительный элемент имеет один или более портов. Порт (Port) — это средство, с помощью которого вычислительный элемент обменивается данными с другими элементами. Соответственно, не имея портов, вычислительный элемент не имеет и смысла, так как его невозможно включить в какой-либо конвейер вычислений. Порты бывают входные, в которые приходят данные из других вычислительных элементов; и выходные, куда отправляет данные вычислительный элемент, хозяин этого порта.

Чистые генераторы данных (Data readers) могут иметь только выходные порты. Эти элементы генерируют данные (например, считывают их из файлов) и отправляют их в выходные порты остальным элементам.

Вычислители (Computers) имеют и входные, и выходные порты. Эти элементы получают данные из входных портов и на основе них (в их контексте) производят новые данные, которые отправляют в свои выходные порты. Поэтому второе название вычислителей — это генераторы данных в контексте других данных. Примером вычислителя может являться элемент поиска каких-либо интересных исследователю участков в биологических последовательностях, например, элемент поиска открытых рамок считывания (или ORF). Он принимает биологическую последовательность из входного порта, ищет в ней ORF, представляющие собой обычные биологические аннотации, и отправляет найденные аннотации в выходной порт. Эти аннотации произведены и существуют в контексте исходной биологической последовательности.

Писатели данных (Data writers) имеют только входные порты. Эти элементы получают данные из входных портов и записывают их, например, в файл, получая конечный продукт вычислительного конвейера.

Служебные элементы (Dataflow elements), как и вычислители, имеют и входные, и выходные порты, но они не имеют вычислительной семантики. Они используются для управления потоками данных. Эти элементы отвечают за такие процессы, как маркировка и фильтрация данных, группировка (merging), мультиплексирование двух потоков данных в один.

Соединяя между собой порты вычислительных элементов, пользователь строит вычислительную схему (конвейер, workflow). Схема представляет собой ориентированный граф, вершинами которого являются вычислительные элементы, а ребрами — соединения портов. Этот граф также называется графом соединения портов или графом потоков данных.

Исходя из специфики dataflow [4], вычислительные элементы могут исполняться только тогда, когда в их входных портах появились все необходимые входные данные. Соответственно, первыми в схеме исполняются чистые генераторы данных, которые не имеют входных портов. Чистые генераторы данных необходимы для работы схемы, иначе ни один из вычислительных элементов ни разу не сможет исполниться, так как не будет иметь входных данных.

2.2. Параметры. Помимо портов, вычислительные элементы могут иметь параметры. Они необходимы для настройки работы схемы. Например, у читателей данных есть такой параметр, как входные файлы, а у писателей — выходные файлы. Различные

вычислительные алгоритмы могут иметь множество параметров, таких как пороги тех или иных величин, процент точности поиска, выбор реализаций алгоритмов и многое другое.

Параметры бывают обязательными и опциональными. Первые необходимо указывать для управления работой элемента. Например, нужно обязательно задать имя входного файла для считывания данных. Если пользователь не выставил какой-либо обязательный параметр в схеме, то такая схема не пройдет валидацию и не запустится.

Опциональные параметры могут быть не заданы, и в этом случае во время запуска схемы будут использованы их значения по умолчанию. Выбор того, какие из параметров назначить обязательными, а какие опциональными, остается за программистом.

2.3. Шина данных. Соединяя между собой выходной и входной порты каких-либо двух вычислительных элементов, пользователь создает шину данных [4]. Шины данных (Bus) служит для передачи данных между вычислительными элементами. Они также образуют ребра графа потоков данных.

Выходной порт может иметь одну или более шин данных (т.е. можно передавать данные из одного вычислительного элемента нескольким другим). А входной порт может иметь ровно одну шину (т.е. его вычислительный элемент принимает данные ровно от одного элемента). Последнее ограничение служит для упрощения вычислительной модели. Отказ от него ведет к внедрению мультиплексирования потоков данных внутри одного порта, а это сложно как для программирования, так и для понимания пользователями. Вместо этого следует использовать служебный элемент мультиплексор, который соединит два потока данных (две шины) в один, и затем по новой шине направить этот поток в порт нужного элемента.

2.4. Сообщения. Данные в шине передаются в сообщениях. Сообщение — это форма упаковки данных для передачи по шине. Одно сообщение может содержать несколько единиц данных, и каждая единица имеет свой идентификатор. Другими словами, сообщение — это структура, представляющая собой ассоциативный массив, в котором ключом является идентификатор, а значением — единица передаваемых данных.

В описываемой вычислительной модели сообщения, входящие в порт некоторого вычислителя, автоматически передаются на выходные порты этого элемента и ждут отправления. Когда вычислитель отправляет новые данные в свой выходной порт, то эти новые данные и старое входное сообщение (автоматически помещенное на выход и ждущее отправления) соединяются в новое сообщение, которое отправляется на все выходные шины. Таким образом, данные, произведенные первым генератором данных, в конечном итоге достигнут последнего писателя данных.

Описанный механизм говорит о том, что данные передаются в шинах в контексте своего существования. Вычислительная модель поддерживает некоторые способы генерации новых контекстов (используя служебные элементы), но, в любом случае, во время работы схемы всегда выполняются следующие свойства:

1. Данные в шине передаются всегда или в полном контексте своего существования в схеме, или вообще без своего контекста.
2. Данные в шине нельзя модифицировать. Можно лишь воспроизвести новые данные в контексте других.

Эти два пункта являются основным требованием к расширению системы, на котором базируется вся вычислительная модель, и который следует учитывать при развитии языка и разработке новых элементов.

2.5. Слоты. Из сказанного выше понятно, что сообщение может содержать несколько единиц данных. Это достигается тем, что данные из сообщений не удаляются, а только добавляются. И, в конце концов, в порт какого-либо вычислителя может прийти сообщение, содержащее различные единицы данных, предназначенные для разных вычислителей. Некоторых из них передаются для этого вычислителя, а другие нужно просто передать в выходной порт. Для разграничения таких единиц данных предназначены слоты.

Слот — это параметр порта, описывающий данные, приходящие во входной порт или выходящие из выходного порта. Таким образом, порт — это не просто сущность, принимающая или отправляющая сообщения в шину, а набор слотов, описывающих данные, которые этот порт принимает или отдает. Соответственно, как и порты, слоты бывают входные и выходные.

Выходные слоты предназначены лишь для того, чтобы идентифицировать отправляемые в порт данные сообщения. Они задают тот самый ключ для ассоциативного массива сообщений.

Входные слоты позволяют задавать, какие из единиц данных пришедшего сообщения предназначены для этого порта. Таким образом, решается проблема разграничения данных в сообщениях.

Пользователь при конструировании схемы, помимо соединения портов, настраивает, как соединены и слоты. Это значит, что пользователь указывает, какие именно данные в шинах предназначены для того или иного вычислительного элемента. Соединение портов создает так называемые соседние вычислительные элементы в графе потоков данных. И между этими соседними элементами создается шина передачи данных. Соединение слотов

описывает передачу данных не только между соседними элементами, а между двумя произвольными элементами, между которыми есть путь в ориентированном графе соединения портов.

2.6. Идентификация данных. Каждый вычислительный элемент, порт и слот имеют свои идентификаторы: element_id, port_id и slot_id, соответственно. Причем каждый из них уникален в своем контексте. Например, идентификатор слота уникален в контексте своего порта.

Каждой единице данных в сообщении можно присвоить идентификатор вида

element_id.port_id.slot_id

Так как в одном сообщении может содержаться не более одной единицы данных с каждого слота, то такой идентификатор является уникальным в контексте одного сообщения, а значит, это позволяет полноценно функционировать всем описанным механизмам.

3. Модель данных.

3.1. Типизация. В описываемой вычислительной модели поддерживается строгая статическая типизация. Все данные, передаваемые по шинам, имеют конкретный тип. И каждый слот описывается типом передаваемых/принимаемых данных. Соединять два слота можно только в том случае, если их типы совпадают. Таким образом, единица данных, помещенная в какое-либо сообщение, идентифицируется типизированным слотом, а значит, может быть передана только слоту с таким же типом.

Соответствие типов проверяется статически, т.е. перед стартом вычислений на стадии валидации. При несовпадении типов пользователь будет об этом извещен, а вычисления не будут начаты.

В настоящее время UGENE Workflow Designer позволяет оперировать следующими типами данных:

1) Text — любые текстовые данные;

2) Sequence — биологическая последовательность. Содержит символы, описывающие нуклеотиды ДНК, РНК или белковых последовательностей, и некоторые метаданные;

3) MSA — множественное выравнивание. Содержит список последовательностей и информацию о расположении их друг относительно друга;

4) Assembly — данные ассемблирования. Содержит результаты работы ассемблеров;

5) Variation track — набор вариаций. Содержит данные о вариациях: SNP, инсерции, делеции;

6) Annotation table — набор аннотаций. Описывает участки в последовательностях, выравниваниях и данных ассемблирования дополнительными сведениями.

3.2. Интерфейс базы данных. Архитектура всей системы UGENE (не только Workflow Designer) предоставляет способ взаимодействия с базой данных. В ней описан интерфейс работы с базой данных (database interface, dbi), реализуя который, программист может быстрым и легким способом начать использовать ту или иную СУБД.

UGENE Workflow Designer предназначен прежде всего для вычислений над данными очень большого объема, которые не помещаются в оперативной памяти. Поэтому использование базы как временного хранилища данных, является очень эффективным средством для достижения этой цели.

В настоящее время происходит постепенное внедрение dbi в работу вычислительных схем. Цель этого внедрения — сделать так, чтобы в шинах передавались не сами данные, а их идентификаторы во временной базе данных. Применение этого механизма уже произведено для самых ресурсоемких типов данных в UGENE Workflow Designer: sequence, msa, assembly, variation track.

Использование описанного механизма очень полезно для вычислений над данными большого объема, ведь он позволяет, например, не держать в оперативной памяти все 3.2 Гб генома человека, а доставать из базы данных только нужную в данный момент его часть.

В настоящее время UGENE предоставляет возможность работы с довольно небольшой и быстрой базой SQLite [3].

4. Конкретный синтаксис. Язык программирования UWL имеет два конкретных синтаксиса: графический и текстовый.

Графический конкретный синтаксис с помощью визуальных средств UGENE Workflow Designer позволяет конструировать вычислительную схему. Пользователю предоставляется графическая сцена и набор вычислительных элементов (палитра), которые он может с помощью мыши добавлять на сцену, перемещать для удобного визуального расположения и соединять порты элементов стрелками, создавая, таким образом, шины данных. Также, существуют графические элементы для регулирования параметров и соединения слотов.

Текстовый конкретный синтаксис предоставляет пользователю все возможности конструирования схем, используя текстовый редактор. Он будет подробно описан ниже.

Графический и текстовый синтаксисы являются эквивалентными. То есть любая вычислительная схема, сконструированная с помощью одного синтаксиса, может быть сконструирована и с помощью другого синтаксиса. Более того, можно создать какую-либо схему с помощью графического синтаксиса, сохранить ее в файл, и этот файл будет семантическим эквивалентом графической схемы в текстовом синтаксисе. Изменив

текстовый файл, можно переоткрыть его в UGENE Workflow Designer и наблюдать изменения.

5. Текстовый синтаксис. В данной статье проведен поверхностный обзор основных конструкций текстового синтаксиса UWL. Формальное описание этого синтаксиса доступно в документации UGENE Workflow Designer [1].

Программа, написанная с помощью текстового синтаксиса, состоит из нескольких частей:

1. Заголовок файла.
2. Описание схемы:
    1) описание вычислительных элементов;
    2) описание соединения портов;
    3) описание соединения слотов;
    4) метаданные.

Заголовок файла схемы на языке UWL состоит из первой обязательной строки и нескольких строк-комментариев, описывающих предназначение схемы.

Пример заголовка:

```
#@UGENE_WORKFLOW
# Описание предназначения
# схемы
```

Описание схемы — это один блок, содержащий в себе все внутренние описания. Блок озаглавлен ключевым словом workflow и опционально может содержать имя схемы.

Пример блока схемы:

```
workflow “Имя схемы” {
}
```

Описание каждого вычислительного элемента схемы — это отдельный блок, содержащий значения параметров. Заголовок блока — это уникальный идентификатор элемента, о котором было сказано в описании вычислительной модели. Обязательными параметрами каждого элемента являются имя и тип элемента. Тип идентифицирует функциональное предназначение элемента (что это за элемент).

Пример вычислительного элемента:

```
read-file {
   type : read-text;
   name : “Прочитать текстовые данные”;
   url-in : D:/file.txt;
}
```

Как видно из примера, вычислительный элемент имеет тип read-text, т.е. этот элемент считывает входной файл в строку (а именно, D:/file.txt, указанный в параметре url-in) и отправляет считанные данные в выходной порт.

Описание соединения портов также содержится в отдельном блоке, который называется служебным словом «.actor-bindings». Идентификаторы вычислительных элементов не могут содержать символ '.', поэтому никаких конфликтов имен не может быть. Внутренний синтаксис этого блока — это перечисление соединения портов вида:

element-id-1.port-id-1 -> element-id-2.port-id-2

Граф потоков данных не должен содержать циклов (это проверяется при открытии схемы).

Описание соединения слотов не содержится ни в каких блоках, а состоит из простого перечисления вида:

element-id-1.port-id-1.slot-id-1 -> element-id-2.port-id-2.slot-id-2

Очень важно, чтобы левая часть соединения портов/слотов описывала именно идентификатор выходного порта/слота, а правая — входного. Это проверяется при открытии схемы.

Пример описания соединений:

```
 .actor-bindings {
   read-text.out-text->write-text.in-text
}
read-text.text->write-text.in-text.text
read-text.url->write-text.in-text.url
```

Последний блок — это метаданные (.meta), который описывает визуальное изображение схемы (позиции элементов на сцене, угол наклона стрелок соединения портов, цвет, шрифт и прочее) и короткие названия параметров. В UGENE разработан механизм запуска любой схемы из командной строки. Для того чтобы передавать какие-либо параметры схемы во время запуска, в метаданных схемы можно назначить каждому параметру уникальное короткое имя, которое и будет именем параметра. Также можно задать описание этого параметра. Например:

```
.meta {
   parameter-aliases {
      write-text.url-out {
         alias : out;
         help : «Путь до выходного файла»
      }
   }
   visual {
      # описание визуальной информации вида
      # ключ : значение;
   }
}
```

Если схема пользователя содержит этот фрагмент кода, и если он работает с интерфейсом командной строки UGENE для запуска выполнения схем, то он может регулировать путь до выходного файла во время запуска с помощью созданного параметра «out»:

```
./ugene --task=schema.uwl --out=/tmp/file.txt
```

Здесь --task=schema.uwl — это параметр, говорящий о том, что нужно запустить workflow-схему из файла schema.uwl.

Блок visual при создании схемы, используя текстовый синтаксис, обычно описывать не требуется, так как проще и удобнее сделать это, используя графический интерфейс UGENE Workflow Designer.

6. Создание пользовательских вычислительных элементов. Пользователю предоставлена возможность создания собственных вычислительных элементов, а не только использование стандартной библиотеки UGENE. Есть два способа добиться этого:

1) интеграция стороннего инструмента с интерфейсом командной строки;
2) написание скрипта на языке ECMAScript [2].

В UGENE встроен механизм подключения сторонних инструментов. Для того чтобы инструмент был совместим для работы в Workflow Designer, должны быть выполнены условия:

1. Инструмент имеет интерфейс командной строки.
2. Он оперирует входными и выходными файлами, форматы которых поддерживает UGENE.
3. Данные в файлах этих форматов имеют один из типов, которые поддерживает UGENE Workflow Designer.

Если требуемый инструмент удовлетворяет этим условиям, то пользователь может запустить специальный визард (мастер), в котором, пройдя шаг за шагом, он интегрирует инструмент со всей системой. Этот визард предложит настроить параметры, входные и выходные порты и слоты нового элемента. Последний шаг настройки — указание того, как запускать этот инструмент через интерфейс командной строки. Здесь можно использовать только что созданные имена параметров и слотов.

В результате работы визарда на упомянутой выше палитре появится новый полноценный вычислительный элемент, который можно использовать в любой схеме. Связь между сторонним инструментом и UGENE во время исполнения схемы происходит с помощью временных файлов.

Другой способ создания новых вычислительных элементов — это использование ECMAScript [2]. UGENE разрабатывается на языке C++ с использованием фреймворка Qt [7],

в который встроен интерпретатор языка ECMAScript. Поэтому не составило большого труда интегрировать в Workflow Designer способ написания элементов на ECMAScript. Элемент на ECMAScript — это всегда вычислитель, так как в этом языке нет поддержки ввода и вывода информации.

Пользователь может запустить диалог создания нового элемента, в котором нужно указать параметры и входные и выходные слоты портов вычислителя. Затем нужно создать скрипт нового элемента. Описанные параметры и слоты можно использовать как переменные внутри этого скрипта. Из переменных входных слотов можно считывать пришедшие в порт данные, а в переменные выходных слотов — помещать данные для отправления.

Пользователю предоставлен набор утилитных функций для работы с основными типами данных UGENE Workflow Designer.

7. Планы развития. В настоящее время разработчикам и руководителям проекта UGENE известны несколько направлений, в которых нужно развивать UWL и Workflow Designer в целом. Первое из них — это развитие интеграции с ECMAScript. Есть определенная цель — создать язык с полноценной стандартной библиотекой прототипов и утилит и платформу для вычислений в области биоинформатики. В настоящее время биоинформатикам и молекулярным биологам известны несколько языков, таких как BioPython или BioPerl, которые позволяют программировать на неком подобии DSL. Создание в UGENE нового языка на основе ECMAScript даст серьезное развитие этой области, так как помимо написания обычных скриптов вся система в целом предоставляет мощный механизм конвейеров вычислений.

В рамках развития интеграции с ECMAScript планируется создать объектную модель биоинформатики, то есть создать прототипы для всех типов данных Workflow Designer (Sequence, MSA и т.д.) и снабдить эти прототипы основными методами, актуальными для этих типов данных.

Другим направлением развития UWL и Workflow Designer в целом является разработка возможности отладки выполнения схем. В настоящее время уже ведется работа над возможностью временной остановки (паузы) и возобновления схемы, изменению параметров схемы «на лету», удобного просмотра содержимого шин данных во время паузы и прочих инструментов. Также должна быть возможность отладки и работы скриптов элементов.

Немаловажным и ресурсоемким направлением развития в области отладки схем является перезапуск вычислений с незначительными изменениями параметров. В этом случае должны повторяться заново не все вычисления, а лишь те, которых касаются совершенные изменения параметров.

Третье направление развития относится к распределенным вычислениям. В UGENE Workflow Designer уже есть некоторые наработки по удаленному запуску схем и мониторингу их исполнения. Планируется развить эту инфраструктуру с целью организации не просто облачных, а распределенных вычислений, то есть использовать несколько компьютеров в процессе выполнения схемы. Первый шаг к этому уже совершается — перевод всех типов данных Worflow Designer для работы с интерфейсом базы данных. Когда вся система будет работать через единый интерфейс, то нетрудно будет изменить его реализацию для работы с какой-либо другой сетевой СУБД, какой как MySQL или PostgreSQL, необходимой для распределенных вычислений.

8. Заключение и выводы. UGENE Workflow Designer является вычислительной платформой, которая может быть расширена в различных направлениях: как в наборе вычислительных элементов, так и в функционально-семантическом плане. Последнее достигается именно за счет поддержки внутреннего языка программирования UWL. Язык программирования всегда имеет какие-либо особенности, договоренности, законы, и следуя им, получается создавать гибкую, целостную, легко поддерживаемую систему.

UGENE предназначен для решения задач биоинформатики. Но Workflow Designer может быть довольно просто переделан для решения задач из некоторых других областей. Язык UWL никак не связан с биоинформатикой, кроме как типами данных и стандартной библиотекой вычислительных элементов.

Система распространяется под свободной лицензией, потому если есть необходимость использовать ее в новой области знаний, то это вполне достижимая цель. Для этого необходимо:

1. Выяснить, какими данными оперирует эта область знаний, и выявить необходимые новые типы.
2. Интегрировать новые типы данных и их вычислительные элементы: читатели и писатели нужных форматов файлов.
3. Доработать служебные элементы с учетом новых типов.
4. Разработать набор вычислителей, необходимых в этой области знаний. Для экономии ресурсов приветствуется использование уже готовых сторонних инструментов и механизм их интеграции с Workflow Designer.
5. При необходимости использования ECMAScript нужно разработать утилитные функции и/или прототипы для новых типов данных.

Все перечисленные шаги имеют подготовленную основу в UGENE, поэтому действия, необходимые для совершения каждого шага, в некоторой степени минимизированы.

Список литературы

1. Документация о UGENE Workflow Designer // UGENE v.1.11.4 documentation. 2012. [Электронный ресурс]. URL: http://ugene.unipro.ru/documentation/wd_manual/index.html (дата обращения: 14.01.2013).
2. Спецификация языка ECMAScript-262 // ECMAScript Language Specification. 2011. [Электронный ресурс]. URL: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (дата обращения: 14.01.2013).
3. Haldar S. Inside sqlite. O'Reilly, 2007. 76 p.: ISBN: 9780596550066.
4. Johnston W.M.,Hanna J.R.P., Millar R.J. Advances in dataflow programming languages. // ACM Computing Surveys (CSUR). 2004. №36 (1). P. 1-34.
5. Kleppe A. Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Professional, 2008. 207 p.: ISBN:0321553454 9780321553454.
6. Okonechnikov K., Golosova, O., Fursov, M., the UGENE team. Unipro UGENE: a unified bioinformatics toolkit // Bioinformatics. 2012. №28 (8). P. 1166-1167.
7. Qt // Qt [Электронный ресурс]. URL: http://qt.digia.com/ (дата обращения: 14.01.2013).

UDK: 004.434

Title: Programming language approach to organization of computations in bioinformatics

Author(s):

German Andreevich Grekhov (LLC "NCIT Unipro"),

Igor Nikolaevich Skopin (Institute of Computational Mathematics and Mathematical Geophysics SB RAS, Novosibirsk State University)

Abstract: This paper presents the approach to creation of computational platforms in the basics of UGENE Workflow Designer. This platform is based on the internal programming language. It demonstrates a method of keeping the system consistent and handy to maintain and develop if you follow the concepts of the platform internal model and the concepts of the programming language syntax.

UGENE Workflow Designer is designed to solve the bioinformatics tasks; and the UWL language is the DSL of bioinformatics. The paper describes the feature of how to reuse the basics of the platform in some other application areas.

Keywords: programming languages, DSL, computational pipelines, workflow, UGENE, bioinformatics

УДК: 168.2 + 025.4 + 519.682 + 004.43 + 811.93

Название: Онтологический подход к проблеме классификации компьютерных языков: состояние и перспективы

Автор(ы):

Идрисов Р.И. (Институт систем информатики им. А.П. Ершова СО РАН),

Одинцов С.П. (Институт математики им. С.Л. Соболева СО РАН)

Шилов Н.В. (Институт систем информатики А.П. Ершова СО РАН, Новосибирский государственный университет, Новосибирский государственный технический университет)

Аннотация: За полувековую историю развития программирования и информационных технологий возникло несколько тысяч компьютерных языков. Виртуальный мир компьютерных языков включает языки программирования, спецификаций, моделирования и другие языки. В каждой из этих ветвей можно выделить разные подходы (например, императивный, декларативный, объектно-ориентированный), дисциплины обработки (например, последовательная, недетерминированная, распределённая) и формализованных моделей (от машин Тьюринга до машин логического вывода). Поэтому актуальной становится проблема классификации компьютерных языков. Для решения этой проблемы предлагается подход, основанный на парадигмах компьютерных языков и выделении общих атрибутов, присущих разным языкам. При этом особую роль призван сыграть разрабатываемый портал знаний о компьютерных языках. Этот портал предназначен для поиска сведений о компьютерных языках в открытых структурированных источниках в сети Интернет, организации хранения и доступа к собранной информации по логическим запросам с целью выявить законы мира компьютерных языков и помощи в выборе компьютерных языков для формирования новых программных проектов. Существующая в настоящее время версия портала далека от совершенства как с точки зрения представленной информации (количественно и качественно), так и с точки зрения выбора логического языка запросов. Поэтому в настоящей статье представлено общее описание проблемы классификации компьютерных языков, описание принципиального подхода к этой проблеме, текущее состояние портала, а также обсуждаются перспективы развития логического языка запросов.

Ключевые слова: компьютерные языки, компьютерные парадигмы, классификация, портал знаний, логика описаний понятий, паранепротиворечивость, устойчивость к противоречиям, верификация моделей.

1. Введение. Под *компьютерным языком* мы понимаем любой искусственный язык, разработанный для автоматической обработки информации, как то: представления,

преобразования и управления данными и процессами. Под *классификацией* какого-либо универсума (универсума компьютерных языков в частности) мы понимаем подход и инструментарий для выделения сущностей этого универсума (объектов, классов и ролей по отношению друг к другу), а также для навигации между этими сущностями.

Первый вопрос, на который надо дать ответ, почему мы считаем актуальной проблему классификации компьютерных языков. Простой ответ состоит в том, что необходимость классификации (и систематизации) следует уже из числа известных компьютерных языков, которых уже несколько тысяч. Так, например, на плакате *The History of Programming Languages* (http://oreilly.com/news/graphics/prog_lang_poster.pdf) издательства O'REILLY, специализирующегося на публикации ученой литературы по компьютерным языкам, представлены сведения о 2500 языках. Эти сведения включают названия, год рождения, авторство, влияние друг на друга и развитие версий компьютерных языков, появившихся в период с 1956 по 2006 гг.

Однако значение классификации компьютерных языков не исчерпывается желанием «навести порядок и систематику». Классификация уже существующих и новых компьютерных языков призвана помочь в выборе подходящих языков для разработки новых программных и информационных систем на основе требований к этим новым системам.

Классификацию компьютерных языков можно пытаться строить по аналогии с естественными науками, например «по Линнею»: царство – тип (у растений отдел) – класс – отряд (у растений порядок) – семейство – род – вид. Так, например, была устроена *Taxonomic system for computer languages* (которая была доступна с 2006 по 2011 г. на http://hopl.murdoch.edu.au/taxonomy.html). Эта таксономия включала множество сведений о 8512 языках, но выделяла очень странные классы компьютерных языков.

На наш взгляд, сама идея таксономии как основы классификации компьютерных языков представляется довольно-таки сомнительной хотя бы уже в силу большого отличия предметных областей в естественных науках и областью компьютерных языков: если предметные области биологии, химии, физики элементарных частиц сравнительно статичны, то область компьютерных языков высоко динамична.

Так, например, только за последнее десятилетие XX века мы наблюдали как быстрый рост существовавших классов компьютерных языков, так и образование новых классов (например, языков представления знаний, языков кластерного/многоядерного программирования, проблемно-ориентированных языков программирования). Некоторые из этих новых компьютерных языков имеют свой специфический синтаксис (например,

графический), свою семантику (т.е. модель обработки информации или виртуальную машину), свою прагматику (т.е. сферу применения и распространения).

Некоторые из давно существующих классов компьютерных языков сравнительно малочисленны (например, языки проектирования СБИС), некоторые – «густонаселенны» (например, языки спецификаций), а некоторые пережили или переживают сейчас период роста и миграций (например, языки разметки). В тоже время разработка новых компьютерных языков (а, следовательно, и образование новых классов компьютерных языков) будет продолжаться по мере компьютеризации новых отраслей человеческой деятельности.

Кроме того, часто специалисты по компьютерным языкам затрудняются отнести тот или иной язык к одному определённому классу. Например, функциональные языки программирования ML и Рефал появились ещё в 1960-ые годы как языки спецификаций вывода (логического и продукционного), но ещё в 1970-ые годы они стали языками программирования для задач искусственного интеллекта. В то же время они (в силу их декларативности) по-прежнему могут служить языками спецификаций для вычислительных программ (см., например, проект VeriFun на http://www.verifun.org/, [21]). А некоторые языки изначально по замыслу их разработчиков не могут быть отнесены к одному конкретному классу. Вот, например, как позиционируют язык Ruby его активные пропагандисты [23]: *Its creator, Yukihiro "matz", blended parts of his favorite languages (Perl, Smalltalk, Eiffel, Ada, and Lisp) to form a new language that balanced functional programming with imperative programming*.

Мы полагаем, что современная классификация универсума компьютерных языков может быть основана на гибком понимании *компьютерных парадигм*. В современной методологии науки парадигмы – это разные подходы к постановке и решению задачи или проблемы. Современным пониманием этого термина мы обязаны широко известной диссертации Томаса Куна [9], впервые опубликованной в 1970 г.

Роберт Флойд стал первым в информатике, кто ввёл понятие *парадигм программирования* в научный оборот в его тьюринговской лекции *Paradigms of Programming* в 1978 г. [Floyd, 1979]. К сожалению, он не дал явного определения этого понятия.

В 2009 г. Питер ванн Рой опубликовал в Интернете таксономию *The principal programming paradigms* (http://www.info.ucl.ac.be/~pvr/paradigms.html), в которой выделил 27 различных парадигм, и предпринял попытку обосновать её в статье [20]. Однако ни эта таксономия, ни соответствующая статья не дают развёрнутого определения понятия парадигм программирования. Можно процитировать только следующее краткое (и неполное, на наш взгляд) определение [20]: *A programming paradigm is an approach to programming a computer*

based on a mathematical theory or a coherent set of principles. Each paradigm supports a set of concepts that makes it the best for a certain kind of problem.

На основе краткого определения понятия парадигмы программирования, предложенного Питером ванн Роем [20], в 2011 г. нами было дано следующее более полное понятие компьютерных парадигм [3, 17]:

- Компьютерные парадигмы – это альтернативные подходы (образцы) к формализации постановок, представления, инструментов и средств решения задач и проблем информатики.
- Компьютерные парадигмы обычно поддержаны строгими математическими теориями/моделями и аккумулированы в соответствующих компьютерных языках.
- Компьютерные парадигмы соответствуют классам компьютерных языков и наоборот:
    - каждый естественный класс компьютерных языков представляет компьютерную парадигму, которую языки этого класса реализуют;
    - каждая компьютерная парадигма естественно характеризуется классом компьютерных языков, реализующих эту парадигму.
- Онтологическое значение компьютерной парадигмы характеризуется классом задач, для решения которых она подходит более всего, или для решения которого она была разработана.
- Образовательное значение компьютерных парадигм состоит в том, что парадигмы учат мыслить по-разному, видеть под разным углом зрения как конкретные задачи информатики, так и мировоззренческие проблемы информатики.

2. Роль синтаксиса, семантики и прагматики. Категории *синтаксиса*, *семантики* и *прагматики* используются для описания как естественных, так и искусственных языков (компьютерных языков в том числе). Синтаксис – это правописание (орфография) языка. Смысл правильно написанных фраз (слов) языка определяет его семантика. Прагматика – это практика использования синтаксически корректных осмысленных фраз языка.

Разумно предположить, что все эти три категории (синтаксис, семантика и прагматика) должны использоваться для выделения классов компьютерных языков и, опосредовано, для выделения компьютерных парадигм.

Использование синтаксиса для классификации компьютерных языков должно отражать как особенности формального синтаксиса, так и человеко-ориентированный аспект восприятия синтаксиса языков. Для разработки синтаксического анализатора очень важно знать, является ли данный язык регулярным, имеет или нет контекстно-свободный

синтаксис, порождается ли он LR(k) грамматикой и тому подобное. Следовательно, эти и другие формально-синтаксические свойства компьютерного языка могут и должны быть атрибутами, учитывающимися при классификации компьютерных языков.

Однако неформальные (или прагматические) характеристики синтаксиса компьютерного языка также могут и должны учитываться при классификации компьютерных языков. Примерами могут служить неопределяемые атрибуты *гибкость*, *естественность*, *ясность*, или определяемые атрибуты такие как, например, *стиль*, определяемый через библиотеку (набор) примеров хорошего стиля. При этом эти неформализуемые характеристики синтаксиса компьютерных языков приобретают всё более и более важное значение, в то время как значение формальных характеристик синтаксиса (в силу развития методов и технологий синтаксического анализа) меняется сравнительно медленно.

Роль и значение семантики для компьютерных языков общепризнанны. Косвенным подтверждением признания этого является количество лауреатов премии имени А. Тьюринга (самой престижной международной премии в области информатики), заслуживших эту премию за вклад в развитие семантики компьютерных языков: Эдсгер Дейкстра (1972), Джон Бэкус (1977), Роберт Флойд (1978), Ч. Энтони Р. Хоар (1980), Никлаус Вирт (1984), Робин Милнер (1991) и Питер Наур (2005). Таким образом, за 46-летнюю историю присуждения этой ежегодной премии 7 раз она присуждалась именно за вклад в развитие языков программирования и их семантики.

Однако, неформальная семантика мало пригодна для использования при классификации в силу нечёткости терминов и понятий, а использование формальной семантики для классификации компьютерных языков до сих пор наталкивалось на следующие препятствия:

- слабое знание формальной семантики среди пользователей компьютерных языков (разработчиков, руководителей и менеджеров проектов);
- распространённое предубеждение, что формальная семантика малополезна и неэффективна на практике;
- широкий спектр индивидуальных нотаций для формальной семантики с разным уровнем абстракции.

Может показаться, что все эти препятствия можно преодолеть путём унификации разных формальных семантик, развития алгоритмических инструментов для унифицированной семантики, преподавания унифицированной семантики в университетах и популяризации унифицированной семантики среди разработчиков.

Этот путь, по нашему мнению, требует чрезвычайно высокой централизации принятия решений о стандартизации формальных семантик, больших капиталовложений и, кроме того,

этот путь наверняка будет препятствовать появлению и внедрению новых вариантов формальных семантик.

Мы считаем, что затруднения, подобные перечисленным выше, могут быть преодолены путём *многомерной стратификации* семантики наиболее типических компьютерных языков. Подчеркнём, что мы ведём речь именно о многомерной стратификации семантики, когда в качестве одного из измерений может выступать, например, *образовательная* семантика, в качестве другого – *реализационная* и так далее.

Так, образовательная семантика компьютерного языка может делить язык на 2-3 образовательных *уровня*. Эти уровни можно условно назвать *элементарным*, *основным* (или базовым) и *экспертным* (или мастерским), причём элементарный и основной уровни или основной и экспертный уровни могут иногда совпадать. Элементарный образовательный уровень компьютерного языка – это учебный диалект для изучения основ языка человеком (возможно, даже без использования компьютера), с очень простой и прозрачной формальной семантикой в виде *виртуальной машины* языка. Основной образовательный уровень компьютерного языка – это диалект языка для регулярного использования, предполагающий понимание со стороны пользователя, как основные конструкции языка могут быть определены (в принципе) в терминах виртуальной машины языка. Экспертный образовательный уровень компьютерного языка – это язык в полном объеме с пониманием того, какие конструкции языка соответствуют формальной семантике, а какие нет, но имеют семантику обусловленную особенностями и эффективностью реализации (т.е. практикой использования). Обращаем внимание, что здесь речь идёт не о машинных языках, автокодах (или ассемблерах) и языках высокого уровня, а об образовательном аспекте языка.

Реализационная семантика компьютерного языка может делить язык на 2-3 или более *слоя*. Обычно можно условно выделить *ядро*, несколько *промежуточных* слоёв и *полный* язык. Ядерный слой имеет эффективную реализационную семантику для класса *платформ* и достаточные средства для реализации *методом раскрутки*, *интерпретации* или *трансформаций* конструкций промежуточных слоёв. Полный язык может иметь только частичную трансформационную семантику в более низкие (промежуточные или ядерный) слои, часть конструкций полного языка может иметь свою индивидуальную реализационную семантику для каждой конкретной платформы. В качестве примера слоёв языка можно сослаться на «уровни» языка C# [1].

Прагматика компьютерных языков (в отличие от синтаксиса и формальной семантики) – это неформализованные представления людей, вовлечённых в жизненный цикл языка (авторов и реализаторов, «экспертов» и простых пользователей) о происхождении,

назначении и использовании этого языка. Другими словами, прагматика компьютерных языков – это человеческие «знания» о компьютерных языках.

Здесь, однако, уместно напомнить, что хотя мы и ведём речь о «знаниях», но на самом деле надо говорить только о представлениях или мнениях (beliefs), т.к. (согласно традиции, восходящей к Платону) знания – это представления о реальности, соответствующие действительности (т.е. истинные представления); «знания» разных людей, вовлечённых в жизненный цикл компьютерного языка, могут быть просто противоречивыми (и, следовательно, не могут соответствовать действительности одновременно).

Тем не менее, описанная интерпретация прагматики компьютерных языков естественно приводит к идее *явного* использования *формальных онтологий* для представления прагматики компьютерных языков.

Формальная «*онтология – это теория объектов и их связей. Онтология предусматривает критерии для выделения различных типов объектов (конкретных и абстрактных, существующих и несуществующих, реальных и идеальных, зависимых и независимых) и связей между ними (отношений, зависимостей и предшествования)*»[1] [22]. Поэтому под онтологией предметной области (прагматики компьютерных языков в том числе) мы будем понимать формальную онтологию, представляющую мнения «экспертов» об объектах этой области, их классах и отношениях между ними. Будем говорить, что онтология задана явно, если она явно представлена в виде графа, вершины которого – объекты онтологии, а дуги – отношения между объектами; в противном случае будем говорить о неявной онтологии. Наиболее известная неявная онтология – это Wikipedia (www.wikipedia.org). Наиболее популярный инструмент создания явных онтологий – это платформа Protégé (http://protege.stanford.edu/) [19].

Здесь уместно сформулировать ещё несколько требований к онтологическому представлению прагматики компьютерных языков: это должна быть *открытая эволюционирующая темпоральная* формальная онтология. Под открытостью мы понимаем доступ для получения данных и внесения изменений без ограничений и привилегий пользователей: любой участник жизненного цикла компьютерного языка может обратиться к онтологии (на правах анонимности или добровольной самоидентификации) с любым допустимым запросом для получения данных, удовлетворяющих запросу, и может добавить любые данные, соответствующие поддерживаемым форматам. Под эволюционностью мы

[1] “ontology is the theory of objects and their ties. Ontology provides criteria for distinguishing various types of objects (concrete and abstract, existent and non-existent, real and ideal, independent and dependent) and their ties (relations, dependencies and predication)” [22]

понимаем не только развитие онтологии во времени, но и поддержку историй правок, хронику её развития, так что в любой момент можно получить справку о её состоянии в любой момент в прошлом. И, наконец, темпоральность означает не только наличие штампов времени у правок, но и возможность формулировки запросов с привязкой ко времени и темпоральных конструкций (например «*до тех пор пока*», «*в следующей правке*» и так далее). Wikipedia может служить хорошим примером открытой эволюционирующей онтологии, язык запросов к этой неявной онтологии не поддерживает темпоральных конструкций.

3. На пути к онтологии компьютерных языков. Идея использовать аппарат формальных онтологий для представления «знаний» о прагматике компьютерных языков естественно обобщается на онтологию «знаний» о компьютерных языках. Объектами этой онтологии могут быть все компьютерные языки, причём слои и уровни каждого языка следует считать отдельными объектами (т.е. компьютерными языками), диалектами друг друга. Отношения между языками в этой онтологии – это отношения, типичные для данной предметной области, например, «*быть диалектом*», «*быть версией*», «*наследует синтаксис*». Атрибутами объектов и отношений могут выступать (формальные и неформальные) характеристики синтаксиса, семантики и (неформальные) характеристики прагматики компьютерных языков и связей между ними. Подробнее наш подход к разработке этой онтологии и её использованию для классификации компьютерных языков мы опишем ниже, а пока остановимся на существующих онтологиях компьютерных языков.

3.1 Существующие онтологии компьютерных языков. В первую очередь следует вспомнить уже упоминавшийся плакат *The History of Programming Languages* (http://oreilly.com/news/graphics/prog_lang_poster.pdf) издательства O'REILLY. Это явная онтология языков программирования не является ни открытой, ни эволюционирующей. Объектами этой онтологии являются языки программирования, но, к сожалению, в этой онтологии не выделены никакие классы языков программирования. Отношения между языками – это «*повлиял на*» (серые стрелки на плакате) и «*стал развитием*» (стрелка своего цвета для каждой линии развития). Языки аннотированы атрибутами «*год рождения*» и «*авторство*». Средства навигации по этой онтологии – по линии времени, по линиям развития или линиям влияния языков.

History of Programming Languages (HOPL, http://www.scribd.com/doc/6915615/An-interactive-Roster-of-Programming-Languages). Значительно более проработанная интерактивная явная онтология языков программирования. Она насчитывает 8512 объектов (т.е. языков программирования), 17837 библиографических ссылки (в качестве атрибутов для

языков), 5445 связей между языками (влияние, диалекты, версии и реализации, связи по авторам, по году и месту рождения и так далее), поддержана (уже упоминавшейся нами довольно-таки спорной) таксономией классов языков программирования. Средства навигации по этой онтологии представлены связями языков и таксономией классов. Недостатком этой онтологии является её закрытость для редактирования и фиксированное состояние (на 2006 г.).

Progopedia (http://progopedia.ru/) – это двуязычная (поддерживает русский и английский языки) открытая эволюционирующая wiki-подобная неявная онтология языков программирования. Она отслеживает три вида связей между языками (диалект, версия, реализация), предлагает две простых таксономии языков (из 31 парадигмы программирования и 13 вариантов типизации в языках программирования), но значительно «беднее» по числу объектов, чем постер от O'REILLY или HOPL. В Progopedia представлены сведения о 160 языках программирования, 76 их диалектов, 332 реализациях и 700 версиях (то есть 1258 объектах). Средства навигации между объектами – алфавитный порядок, диалекты, версии и реализации, таксономии по парадигмам и вариантам типизации.

3.2 Основы развиваемого подхода. Мы разрабатываем открытую эволюционирующую темпоральную онтологию на основе логики описаний понятий (Description Logic – DL) [2, 4, 18] с использованием технологий языка OWL (Web Ontology Language) [18, 8]. Поэтому мы в описании нашего подхода будем использовать терминологию DL и OWL через слеш DL/OWL.

Объектами разрабатываемой онтологии являются компьютерные языки, включая слои и уровни языков как отдельные объекты. Например, Pascal, LISP, PROLOG, SDL, LOTOS, UMLT, а также C, C-light and C-kernel, OWL-Lite, OWL-DL и OWL-full – эти все языки могут быть объектами. Связи между объектами задаются ролями/свойствами, которые могут быть специфицированы ролевыми термами DL, построенными из явно заданных элементарных ролей/свойств. Например, «*наследовать синтаксис*» или «*быть диалектом*» могут быть элементарными ролями/свойствами, которые должны быть заданы явным перечислением пар объектов (компьютерных языков), связанных соответствующим отношением. Понятиями/классами являются множества объектов, которые могут быть специфицированы (выделены) посредством понятийных термов DL, построенных из явно заданных элементарных понятий. Например, элементарными понятиями могут быть «*является функциональным языком*» или «*является языком спецификаций*»; их содержание должно быть задано явным перечислением объектов (компьютерных языков), входящих в соответствующий класс.

Т.к. мы считаем, что компьютерные парадигмы соответствуют классам компьютерных языков и наоборот, то сказанное выше о выделении классов компьютерных языков посредством понятийных термов DL означает, что каждый понятийный терм определяет компьютерную парадигму посредством выделения понятия/класса компьютерных языков. В такой трактовке парадигмы компьютерных языков перестают быть древовидной таксономией и становятся подрешёткой решётки множеств компьютерных языков.

Объекты (компьютерные языки) могут быть аннотированы разнообразными формальными атрибутами (например, характеристиками формального синтаксиса языка) и неформальными атрибутами (например, библиотеками примеров хорошего стиля для языка). Список возможных атрибутов не фиксирован, но мы полагаем, что некоторые атрибуты должны присутствовать обязательно (т.е. автоматически связываться с любым объектом при его инициализации), например:

- дата рождения языка (с разной степенью гранулярности – от точной даты до интервала в несколько лет);
- авторство языка;
- рекомендуемые библиографические ссылки (заданные URI или другим образом);
- ссылка на доступную пробную версию (простую и компактную в установке или сетевую реализацию языка, распространяемую свободно или условно-свободно).

(При этом мы не запрещаем, что значения этих атрибутов могут быть неопределенны, т.е. соответствующие поля незаполнены.)

Мы также полагаем, что некоторые элементарные понятия/классы должны автоматически присутствовать в нашей онтологии, например: языки с контекстно-свободным синтаксисом, функциональные языки, языки спецификаций, исполняемые языки, языки со статическими типами, языки с динамическим связыванием и другие. Кроме того, по-видимому имеет смысл предусмотреть элементарный класс *парадигмальных* языков, который должен содержать хотя бы по одному языку (но немного) из каждого другого элементарного понятия/класса. Мы также предполагаем заимствовать другие идеи для элементарных понятий/классов, представленные, например, в The Language List (http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm).

Наполнение элементарных понятий/классов должно происходить на основе явно указанных атрибутов объектов (компьютерных языков), например, язык попадает в языки спецификаций, если у этого языка значение соответствующего атрибута установлено явно.

Как уже было сказано, неэлементарные понятия/классы определяются через элементарные посредством понятийных термов DL. Например, *исполнимые языки спецификации* – это пересечение понятий/классов *языков спецификаций* и *исполнимых языков*.

Т.к. наша онтология строится в модели *открытого мира* (т.е. неполной информации), то, естественно, возникают трудности в определении операции дополнения для понятий/классов: например, если для какого-либо языка не сказано, что он имеет контекстно-свободный синтаксис, это еще не означает, что этот язык не имеет контекстно-свободного синтаксиса, но может означать, что значение соответствующего атрибута пока не проставлено. Поэтому мы предполагаем, что в нашей онтологии одновременно с введением какого-либо нового *позитивного* атрибута будет автоматически порождаться его *негативный* напарник. Например, при создании позитивного атрибута «*имеет контекстно-свободный синтаксис*» будет автоматически порождён негативный атрибут «*не имеет контекстно-свободный синтаксис*». Введение негативных атрибутов (которые также могут иметь неопределённые значения) позволяет решить проблему дополнения для понятий/классов: в дополнение до какого-либо элементарного класса, определённого позитивным атрибутом, попадают только те языки, у которых явно проставлено значение соответствующего негативного атрибута. Например, язык программирования C не является языком спецификаций в нашей онтологии, если только явно не проставлено значение его соответствующего негативного атрибута (иначе мы ничего не можем сказать на этот счёт, исходя из представленных в онтологии «знаний»).

Элементарные роли/свойства – это естественные отношения между компьютерными языками: «*быть диалектом*», «*наследовать синтаксис*» и т.д. Например: «C-light *является промежуточным слоем* C», «OWL *наследует синтаксис* XML» и так далее. Для построения сложных ролей/свойств из элементарных мы разрешаем использовать стандартный набор монотонных операций алгебры бинарных отношений: объединение, пересечение, инверсию (взятие обратного отношения), транзитивное замыкание. Например, роль/свойство «*использует синтаксис диалекта*» – это просто композиция элементарных ролей/свойств «*использует синтаксис*» и «*является диалектом*».

Однако с операцией дополнения ролей/свойств возникают те же трудности, что и с операцией дополнения понятий/классов, которые обусловлены неполнотой информации в модели открытого мира. Для преодоления этого мы намерены использовать тот же подход, основанный на создании негативного напарника для каждой позитивной элементарной роли/свойства.

Здесь стоит заметить, что в мире компьютерных языков есть три роли/свойства, специфические для этой предметной области: «*быть диалектом*», «*быть версией*», «*быть реализацией*». (Об этих ролях/свойствах мы упоминали при обсуждении онтологий HOPL и Progopedia.) Разумеется, эти роли/свойства являются элементарными в этой онтологии и должны задаваться явным перечислением пар языков. Но для пользователя онтологией надо предусмотреть правила, которыми следует руководствоваться при причислении пар языков к одному из этих отношений. К сожалению, нет консенсуса по определению этих отношений между языками. Так, например, Progopedia считает, что реализация имеет несколько версий, а The Language List, наоборот, считает, что версия может иметь несколько реализаций. Поэтому для определённости мы закрепляем для нашей онтологии компьютерных языков следующие определения, которыми следует руководствоваться:

- диалекты – это языки у которых общий элементарный уровень. Например, Common LISP и Home LISP – это диалекты друг друга;
- версии (или варианты) – это языки с общим основным уровнем. Например, Visual C и Borland C – это версии друг друга;
- реализация – это платформа-специфическая версия языка. Например, Visual C – это версия языка C Карнигана и Риччи для платформы Windows.

Универсальные и экзистенциальные (кванторные) ограничения также могут использоваться для конструирования новых понятий/классов. Например, если пользоваться нотацией DL, то понятийный терм

$$(\textit{markup language}) \sqcap \exists(\textit{uses syntax of}): (\neg\{\text{XML}\})$$

охватывает класс языков разметки (*markup language*), которые используют синтаксис не XML. Т.к. дополнение понятий в нашей онтологии реализуется при помощи негативных атрибутов, то, если предположить, что у языка разметки LATEX этот негативный атрибут был кем-то явно указан, то LATEX будет примером языка из этого понятия. Примером использования универсального ограничения может служить следующее предложение (из T-бокса): $\{\text{XML}\} \sqsubseteq \forall(\textit{is dialect of}): (\neg\{\text{ML}\})$; оно выражает тот факт, что XML является диалектом какого угодно языка, но не ML.

3.3 Программная реализация. Мы приступили к реализации портала знаний о компьютерных языках (который со временем должен развиться в явную открытую эволюционирующую темпоральную онтологию) около двух лет назад [17]. В 2011–12 гг. он был доступен для публичного тестирования в сети Интернет по адресу http://complang.somee.com/Default.aspx. Он был реализован как веб-приложение, поэтому

познакомиться с ним можно было с использованием браузера. Интерфейс позволял пользователю просматривать и редактировать любые данные портала. Однако этот прототип не поддерживал эволюционность и темпоральность. Для внутреннего представления данных использовался RDF-репозиторий.

Прототип портала предоставлял два основных сервиса: это решатель DL-запросов как основное средство выделения классов, навигации по онтологии и проверки совместности, и визуализация онтологии в виде графа. Решатель – это верификатор моделей с явным представлением состояний, но для паранепротиворечивого варианта DL [10, 11, 12], обогащённого алгебраическими конструкциями из анализа формальных понятий (Formal Concept Analysis – FCA) [7, 18. 2]. В основе этого варианта DL лежит четырёхзначная пропозициональная логика Белнапа [5, 13, 14, 15]. Конструкции. Заимствованные из FCA – это верхние и нижние производные. Такой выбор варианта DL обусловлен открытостью онтологии и неполнотой информации в модели открытого мира. Использование алгоритма верификации модели в качестве основы решателя запросов (вместо логического вывода) обусловлено тем, что мы намерены создать эволюционирующую онтологию для очень динамичного универсума компьютерных языков, а не инструмент для вывода следствий из заранее сформулированных законов, сложенных в базе знаний портала.

Благодарности. Работа подготовлена в рамках междисциплинарного проекта СО РАН №3 «Принципы построения онтологии на основе концептуализаций средствами логических дескриптивных языков» на 2012–2014 гг.

Список литературы

1. Непомнящий В.А., Ануреев И.С., Дубрановский И.В., Промский А.В. На пути к верификации C#-программ: трехуровневый подход // Программирование. 2006. №4. С.4-20.
2. Шилов Н.В. Формализмы и средства создания и поддержания онтологий. // Модели и методы построения информационных систем, основанных на формальных, логических и лингвистических подхода: монография / под ред. Марчука А.Г.. Новосибирск: из-во СО РАН, 2009. С.10-48.
3. Akinin A.A., Zubkov A.V., Shilov N.V. New Developments of the Computer Language Classification Knowledge Portal // Proceedings of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2012), May 30-31, 2012, Perm, Russia. P. 54-58.

4. Baader F., Calvanese D., Nardi D., McGuinness D., and Patel-Schneider P. (editors) The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press, 2003.
5. Belnap N.D. How a computer should think. Contemporary Aspects of Philosophy: Proceedings of the Oxford International Symposium, 1977. P. 30-56.
6. Floyd R.W. The paradigms of Programming // Communications of ACM. 1979. V. 22. P. 455-460. (Русский перевод: Флойд Р. О парадигмах программирования. // В кн.: Лекции лауреатов премии Тьюринга. М: Мир, 1993.)
7. Ganter B. and Wille R. Formal Concept Analysis. Mathematical Foundations. Springer Verlag, 1996.
8. Hitzler P., Krötzsch M., Rudolph S. Foundations of Semantic Web Technologies. Chapman & Hall/CRC, 2009.
9. Kuhn T.S. The structure of Scientific Revolutions. Univ. of Chicago Press, 3rd Ed., 1996. (Русский перевод: Кун Т. Структура научных революций. Издательство АСТ, 2003.)
10. Ma Y., Hitzler P., and Lin Z. Algorithms for paraconsistent reasoning with owl. Proc. of European Semantic Web Conference 2007, Springer Lecture Notes in Computer Science, V. 4519. 2007. P. 399-413.
11. Ma Y., Hitzler P., and Lin Z. Paraconsistent reasoning for expressive and tractable description logics. Proc. of the 21st International Workshop on Description Logic, CEUR Electronic Workshop Proceedings. V. 353, 2008.
12. Ma Y. and Hitzler P. Paraconsistent Reasoning for OWL 2. Proc. of Web Reasoning and Rule Systems, Springer Lecture Notes in Computer Science. V. 5837. 2009. P. 197-211.
13. Odintsov S.P., Wansing H. Inconsistency-Tolerant Description Logic. Motivation and Basic Systems, in: V.F. Hendricks, J. Malinowski (eds.) “Trends in Logic: 50 Years of Studia Logica”, Kluwer Academic Publishers, Dordrecht, 2003. P. 287-321.
14. Odintsov S.P, Wansing H. Inconsistency-Tolerant Description Logic. Part II: A Tableau Algorithm for $CALC^{C}$, Journal of Applied Logic. 2008. V. 6. P. 343-360.
15. S.P. Odintsov, H. Wansing. Modal logics with Belnapian truth values. Journal of Applied Non-Classical Logics. 2010. V. 20. P. 279-301.
16. Ritchie D.M. The Development of the C Language. The second ACM SIGPLAN History of Programming Languages Conference (HOPL-II), ACM, 1993. P. 201-208.
17. Shilov N.V., Akinin A.A., Zubkov A.V., and Idrisov R.I., Development of the Computer Language Classification Portal // Lecture Notes in Computer Science. 2012. Vol.7162. P.340-348.

18. Staab S. and Studer R. (editors) Handbook on Ontologies. International Handbooks on Information Systems. Springer, 2nd edition, 2009.
19. Tudorache T., Nyulas C. I., Musen M. A., and Noy N. F. WebProtégé: A Collaborative Ontology Editor and Knowledge Acquisition Tool for the Web // Semantic Web Journal. 2011. V. 11, n.16. P. 1-11.
20. van Roy P. Programming Paradigms for Dummies: What Every Programmer Should Know // In: New Computational Paradigms for Computer Music. IRCAM/Delatour, France. 2009. P. 9-38.
21. Walther Ch., Schweitzer St. About VeriFun // Lect. Not. in Comp. Sci. 2003. V. 2741. P. 322-327.
22. Ontology. A Resource Guide for Philosophers // [Электронный ресурс]. URL: http://www.formalontology.it/ (дата обращения: 19.11.2012).
23. Ruby. A Programmer's best friend // [Электронный ресурс]. URL: http://www.ruby-lang.org/en/about/ (дата обращения: 19.11.2012).

UDK: 168.2 + 025.4 + 519.682 + 004.43 + 811.93

Title: Ontology approach to classification of Computer Languages: current state and challenges

Author(s):

Renat I. Idrisov (A.P. Ershov Institute of Informatics Systems),

Sergey P. Odintsov (S.L. Sobolev Institute of Mathematics),

Nikolay V. Shilov (A.P. Ershov Institute of Informatics Systems, Novosibirsk State University, Novosibirsk State Technical University)

Abstract: During the semicentennial history of Computer Science and Information Technologies, several thousands of computer languages have been created. The computer language universe includes languages for different purposes (programming, specification, modeling, etc.). In each of these branches of computer languages it is possible to track several approaches (imperative, declarative, object-oriented, etc.), disciplines of processing (sequential, non-deterministic, distributed, etc.), and formalized models, such as Turing machines or logic inference machines. The listed arguments justify the importance of of an adequate classification for computer languages. Computer language paradigms are the basis for the classification of the computer languages. They are based on joint attributes which allow us to differentiate branches in the computer language universe. We present our computer-aided approach to the problem of computer language classification and paradigm identification. The basic idea consists in the development of a specialized knowledge portal for automatic search and updating, providing free access to

information about computer languages. The primary aims of our project are the research of the ontology of computer languages and assistance in the search for appropriate languages for computer system designers and developers. The paper presents our vision of the classification problem, basic ideas of our approach to the problem, current state and challenges of the project, and design of query language.

Keywords: computer languages, computer paradigm, classification, knowledge portal, description logic, paraconsistency, inconsistency-tolerance, model checking

УДК: 519.681

Название: Комплексный подход к локализации ошибок при верификации Си-программ

Автор(ы):

Кондратьев Д.А. (Новосибирский государственный университет),

Промский А.В. (Институт систем информатики им. А.П. Ершова СО РАН)

Аннотация: Отличительной чертой проекта C-light является максимальное использование формальных непротиворечивых методов. Это касается не только фундаментальных базисов для верификации, таких как операционная семантика Плоткина или логика Хоара, но и реализационных аспектов. Одним из таких моментов является локализация потенциальных ошибок. В большинстве известных систем верификации локализация ошибок просто реализуется программистами как часть функционала системы без подведения какой-либо формальной основы под этот процесс. В свете недавнего расширения проекта C-light техникой семантической разметки и выбора инструментария LLVM/Clang для входного блока системы мы приводим обзор наших наработок для решения этой задачи.

Ключевые слова: семантика, спецификация, верификация, трансляция, локализация ошибок, разметка, Си, C-light, LLVM, Clang

1. Введение. В области разработки систем верификации программ[1] можно выделить два основных направления. К первому относятся системы, ориентированные на максимальную производительность или на поиск как можно более многочисленных типов ошибок[2]. Однако зачастую эффективность поиска ошибок является следствием применения менее строгих методов и алгоритмов. Особенно показателен пример системы ESC/Java, где применялись не только неполные, но в некоторых случаях и противоречивые методы.

Противоположный лагерь представляют академические исследования, ориентированные в основном на обучение методам верификации, или системы, поддерживающие более узкие классы свойств программ и/или ошибок в них. Для таких систем характерно использование более строгих и надежных подходов. Так, если сравнивать их с упомянутой ESC/Java, то можно заметить, что для ограниченных диалектов Java Card и Java^{light} при помощи системы Isabelle/HOL были разработаны полные и непротиворечивые семантики.

В проекте C-light, развиваемом в Лаборатории теоретического программирования ИСИ СО РАН, мы, конечно же, стремимся к широкому охвату языка Си и верифицируемых свойств в рамках дедуктивного подхода Хоара. Однако приоритетной для нас является максимальная корректность используемых методов.

Для того чтобы сформулировать основную тему данной работы, нам понадобится крат-

[1]Мы намеренно не указываем язык, так как это верно не только для Си, но и для, скажем, Java, C# и т.д.

[2]В зарубежной литературе используется термин "industrial-strength verification".

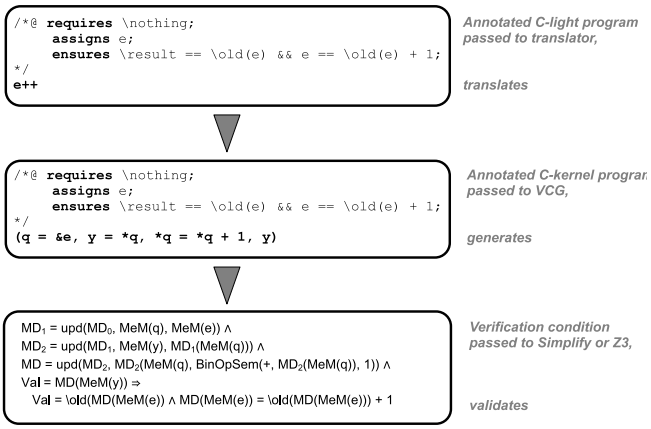


Рис. 1: Схема верификации в проекте C-light

кий обзор проекта C-light. Итак, язык C-light является представительным подмножеством стандарта C99. Детальный обзор его допустимых конструкций можно найти в [3]. Здесь же отметим следующее. В ходе эволюции языка Си во многом преследовались цели эффективности и максимальной переносимости программ. Это привело к тому, что стандарт для некоторых конструкций не дает детальной или вообще детерминированной семантики, оставляя ее на усмотрение разработчиков компилятора. Поэтому семантика некоторых низкоуровневых конструкций C-light обладает высоким уровнем абстракции (например, адресация или раскладка значений в памяти), а такой тип данных, как объединение (union), запрещен. Также в C-light фиксирован порядок вычисления выражений, а побочные эффекты срабатывают сразу. Для C-light была разработана структурированная операционная семантика, фактически являющаяся его формальным определением. Несмотря на ограничения, язык C-light все же слишком широк для классической логики Хоара. Поэтому в нем было выделено ограниченное ядро C-kernel, и предложены формальные правила перевода из C-light в C-kernel. Была доказана формальная корректность такого перевода. Это позволило предложить простую аксиоматическую семантику C-kernel в классическом стиле Хоара, что, в свою очередь, позволило доказать ее непротиворечивость относительно операционной семантики. Таким образом, для C-light используется двухэтапная схема верификации, представленная на рис. 1. Отметим, что в качестве языка спецификаций выбран язык ACSL.

Как видно из обзора, корректность каждого из этапов в нашем подходе была формально обоснована. Вместе с тем схема на рис. 1 не описывает весь процесс верификации. В частности, в ней не представлена интерпретация результатов верификации пользователем. Особый интерес представляет случай, когда доказатель теорем (prover) обнаружил ложные или недоказанные условия корректности, и необходимо найти источник проблем

в программе. В подавляющем большинстве случаев трассировку условий обратно в исходную программу реализуют как часть системы, но не предлагают никакой формализации для нее. Для решения этой задачи в нашем проекте было разработано расширение аксиоматической семантики языка семантическими метками, которые позволяют формализовать соотнесение условий корректности и фрагментов программ, а также позволяют автоматически порождать объяснения для условий корректности на естественном языке.

Наконец, разработка формализма для локализации ошибок снова показала, что корректность должна быть установлена не только для теоретических методов, но и для практических аспектов, связанных с реализацией системы верификации. Идеальным решением была бы реализация системы на языке C-light и ее (само)верификация. Эта амбициозная задача еще только формулируется в планах. Пока же в контексте задачи локализации ошибок при верификации разрабатывается входной анализатор/транслятор с помощью нового инструментария LLVM/Clang. При этом для той его части, что вкладывается в C-light, будут предложены формальные спецификации на языке ACSL. Это позволит провести его частичную верификацию.

Итак, в данной работе мы представляем обзор теоретических и практических результатов, полученных нами в контексте решения задачи локализации ошибок при верификации Си-программ. В разд. 2 обсуждаются причины выбора нового инструментария, и описана архитектура анализатора/транслятора. Разд. 3 посвящен расширению логики Хоара для языка C-kernel семантическими метками и их использованию для локализации ошибок и интерпретации условий корректности. Обзор родственных работ дается в разд. 4.

Эта работа частично поддержана грантом РФФИ 11-01-00028-а.

2. Трансляция из C-light в C-kernel. В ходе развития проекта был выбран новый подход для реализации транслятора из C-light в C-kernel. На смену использовавшейся ранее классической связке Flex/Yacc(Bison), пришла относительно новая связка LLVM/Clang. О причинах такого выбора и архитектуре разработанного прототипа транслятора говорится в первой и второй частях данного раздела, соответственно. Вопрос сопоставления фрагментов C-kernel программы и фрагментов исходной C-light программы затронут при рассмотрении одного из классов API Clang. Это сопоставление важно для задачи локализации ошибок.

2.1. Выбор инструментария. Было принято решение не реализовывать данный транслятор «с нуля», а использовать уже существующие инструменты для лексического анализа и построения внутреннего представления аннотированных C-light программ. Данное решение было принято по следующим причинам:

1. Многие из этих инструментов имеют очень хорошо продуманную структуру, и поэто-

му, предоставляют довольно удобное API для работы со внутренним представлением программ. Если бы была предпринята попытка своими силами разработать собственное API для тех же целей, то оно вероятнее всего не было бы удобней API, предоставляемых этими инструментами, так как эти API совершенствовались в течение многих лет.

2. Многие из этих инструментов гарантируют, что они поддерживают все возможности языка Си, описанные в стандарте ISO/IEC 9899:1999. Для реализации нашего транслятора этот факт очень важен, так как язык C-light является очень широким подмножеством языка Си, и поэтому очень схож с ним. Попытка реализовать поддержку всех возможностей языка C-light получилась бы бесполезной тратой времени ввиду того, что уже существуют инструменты, поддерживающие все возможности языка Си.

3. Очень многие из этих инструментов можно использовать не просто для анализа C-light программ, а для анализа C-light программ, снабженных ACSL спецификациями. Это возможно благодаря тому, что с точки зрения этих инструментов ACSL спецификации являются самыми обычными комментариями в Си-программе и никак не нарушают синтаксис и семантику Си-программ. Более того, многие из этих инструментов не вырезают комментарии, и поэтому позволяют создавать анализаторы ASCL спецификаций.

4. Многие из этих инструментов хорошо протестированы на наличие ошибок. Это подтверждается большими тестовыми базами, на которых проверялись эти инструменты. Было бы сложно создать такие большие тестовые базы только собственными силами.

В качестве такого инструмента для лексического анализа и построения внутреннего представления C-light программ было выбрано API на языке программирования C++, предоставляемое компилятором Clang и виртуальной машиной LLVM. Этот инструментарий в полной мере обладает всеми перечисленными выше преимуществами и, кроме того:

- API на языке программирования C++ позволяет использовать возможности объектно-ориентированного анализа и дизайна при разработке транслятора. Использование объектно-ориентированного анализа и дизайна позволяет значительно облегчить разработку практически любых программных систем. Также использование

объектно-ориентированного анализа и дизайна при разработке транслятора дает возможность довольно легко вносить изменения в уже реализованный транслятор. Задача внесения изменений в уже реализованный транслятор является актуальной в связи с тем, что в настоящее время активно ведутся работы по расширению языка C-light.

- API на языке программирования C++, предоставляемое компилятором Clang, позволяет очень легко работать с внутренним представлением Си-программ. Например, это API предоставляет возможность достаточно легко обойти всю программу, пользуясь ее внутренним представлением.

- Задача трансляции исходного кода C-light программ в исходный код C-kernel программ предполагает много работы со строковыми данными. Язык программирования C++, на котором предоставляет свой API компилятор Clang, дает возможность легко работать со строковыми данными по сравнению со многими другими языками программирования (например, по сравнению с языком программирования Си).

- Сейчас компилятор Clang активно поддерживается, и нет оснований полагать, что его поддержка в скором времени прекратится. Значит, с большой уверенностью можно предполагать, что API компилятора Clang является надежным инструментом для лексического анализа и построения внутреннего представления Си-программ.

- Язык программирования C++, на котором предоставляет свой API компилятор Clang, дает возможность снабдить спецификациями код методов классов, определенных при реализации транслятора по аналогии с ACSL-спецификациями аннотированных C-light программм. Благодаря этому появляется возможность провести формальную верификацию транслятора.

Так как API Clang имеет встроенный лексический анализатор и парсер Си-программ, то использование API Clang позволило не заниматься реализацией этой функциональности. В тексте программы, являющейся реализацией транслятора, лексический анализ и парсинг C-light программ занимает всего 35 строк. Если бы в тексте этой программы не использовалось API Clang, то, по проведенным оценкам, лексический анализ и парсинг C-light программ занимал бы в этом тексте приблизительно 7000 строк. Получается, что при использовании API Clang удалось добиться сокращения части кода, отвечающей за рассматриваемую функциональность, приблизительно в 200 раз.

Отметим, что использование API Clang позволяет добиться существенного уменьшения текста программы, реализующей транслятор. Текст этой программы занимает приблизительно 11000 строк. Если бы в этом тексте не использовалось бы API Clang, то, по

проведенным оценкам, текст этой программы занимал бы приблизительно 22000 строк. Итак, с помощью API Clang удалось сократить количество строк кода транслятора приблизительно в 2 раза. Все эти данные подтверждают правильность принятых решений.

2.2. Архитектура транслятора. Транслятор аннотированных C-light программ в аннотированные C-kernel программы был реализован на языке программирования C++, так как при реализации этого транслятора активно использовалось API на языке программирования C++, предоставляемое компилятором Clang. При создании транслятора активно использовались принципы объектно-ориентированного анализа и дизайна. Поэтому при рассмотрении текста программы, являющейся реализацией транслятора, интересно рассмотреть классы из API компилятора Clang, объекты которых использовались в этой программе, а также другие классы, на которых строится реализация правил трансляции. Особенно важно рассмотреть классы, отвечающие за внутреннее представление программы и работу с ним.

В API, предоставляемым компилятором Clang на языке программирования C++, внутреннее представление программы называется `AST`. Для работы с этим внутренним представлением API компилятора Clang предлагает довольно большой набор классов. Задачи трансляции удобнее всего решать, используя те из этих классов, которые отвечают за реализацию шаблона проектирования `Visitor`. В классической книге назначение [1] паттерна `Vistor` описывается так: “Описывает операцию, выполняемую с каждым объектом из некоторой структуры” т.е. паттерн `Visitor` позволяет осуществить обход `AST` и выполнить при этом обходе ряд операций для реализации правил трансляции. При использовании паттерна `Visitor` можно не опасаться появления зависимостей от внутреннего устройства `AST`. Использование в нашем трансляторе классов, которые отвечают за реализацию шаблона проектирования `Visitor`, позволяло при его разработке концентрироваться не на обходе дерева, а на реализации правил трансляции.

Из довольно большого числа классов, которые отвечают за реализацию паттерна `Visitor` в API, предоставляемом компилятором Clang, при разработке нашего транслятора использовались следующие:

`ASTConsumer` — это интерфейс, который должны реализовывать те, кто использует `AST`. `ASTConsumer` позволяет создать уровень абстракции, на котором можно быть независимым от того, кто предоставляет `AST`. Таким образом, используя `ASTConsumer`, можно не заботиться о том, кто предоставил `AST` — парсер или класс, десериализующий `AST` из какого-либо файла. Для того чтобы начать обход `AST`, удобнее всего наследоваться от класса `ASTConsumer` и переопределить его методы. При реализации транслятора был создан класс `MyASTConsumer`, наследник класса `ASTConsumer`. В классе `MyASTConsumer` был переопределен метод `HandleTopLevelDecl` класса `ASTConsumer`. После переопределения этого

метода, была получена возможность обойти все декларации самого высокого уровня. В нашем трансляторе обход всех деклараций самого высокого уровня является началом обхода `AST`.

`RecursiveASTVisitor` — это класс, который позволяет в прямом порядке поиском в глубину обойти все `AST` и посетить каждый узел. При реализации транслятора была использована в том числе и такая функциональность этого класса: если выбрать декларацию, то этот класс позволяет обойти ту часть `AST`, которая имеет корнем эту декларацию. Именно эта возможность была использована при реализации транслятора в том месте, где вызывается метод `TraverseDecl`, и аргументом ему передается декларация, полученная в начале обхода `AST`. Также этот класс предоставляет важнейшую возможность наследоваться от него и переопределить методы, вызывающиеся при обработке нужных узлов `AST`. Например, можно переопределить метод VisitDecl, который вызывается при обходе каждого узла `AST`, являющегося декларацией. Таким образом, использование класса `RecursiveASTVisitor` заметно облегчило обход `AST` при реализации правил трансляции.

`SourceLocation` — это класс, который отвечает за местоположение того или иного объекта в исходном коде программы. Например, объект этого класса можно получить у заданной деларации и оператора. При нахождении в определенном узле AST во время обхода AST с помощью наследника класса `RecursiveASTVisitor` можно понять, какому месту в исходном коде программы соответствует этот узел. Таким образом, с помощью класса `SourceLocation` можно установить соответствие между исходным кодом и синтаксическими конструкциями в программе. Также очень важно, что при реализации правил трансляции с помощью класса `SourceLocation` можно запоминать участки кода, в которых были произведены изменения, что дает возможность создать протокол, по которому можно от конструкций в тексте C-kernel программы вернуться к конструкциям текста C-light программы. Такой протокол позволяет сообщить не только о том, что программа не корректна, но и указать в исходном тексте C-light программы на причины.

`Rewriter` — это класс, который позволяет изменять исходный код программы, загруженный в специальные буферы. При реализации правил трансляции все необходимые изменения в исходном коде программы осуществляются с помощью класса `Rewriter`. Класс `Rewriter` использует объекты классов `SourceLocation`, чтобы узнать, в каких местах исходного кода производить изменения. `Rewriter` позволяет осуществлять операции вставки, копирования, замены, т.е. все те операции, которые производятся во время работы с текстом программы при применении правил трансляции. После применения правил трансляции у объекта класса `Rewriter` можно запросить объект класса `RewriterBuffer`. Из объекта класса `RewriterBuffer` можно получить уже модифицированный исходный код.

Приведем пример реализации такого правила трансляции из [2]: всякая декларация

вида "*type_spec declarator_list*; в которой отсутствует явная спецификация класса памяти, заменяется на декларацию вида "*storage type_spec declaratory_list*; где *storage* — это, в зависимости от места самой декларации, либо ключевое слово `static`, либо ключевое слово `auto`. Часть исходного кода транслятора, отвечающая за реализацию данного правила трансляции, приведена в листинге №1:

```
...

class MyASTVisitor : public RecursiveASTVisitor<MyASTVisitor>
{
 public:
    MyASTVisitor(Rewriter &R): TheRewriter(R){}

    bool VisitStmt(Stmt *s)
    {
       if (isa<DeclStmt>(s))
       {
          DeclStmt *declStmt = cast<DeclStmt>(s);
          Decl *d = *(declStmt->decl_begin());

          if (isa<VarDecl>(d))
          {
             VarDecl *varDecl = cast<VarDecl>(d);

             if (varDecl->hasLocalStorage())
             {
                TheRewriter.InsertText(varDecl->getLocStart(),
                                       "auto ", true, true);
             }
          }
       }

       return true;
    }

private:
```

```
    void AddBraces(Stmt *s);
    Rewriter &TheRewriter;
};

...

class MyASTConsumer : public ASTConsumer
{
 public:
    MyASTConsumer(Rewriter &R): Visitor(R){}

    virtual bool HandleTopLevelDecl(DeclGroupRef DR)
    {
       for (DeclGroupRef::iterator b = DR.begin(), e = DR.end();
            b != e; ++b)
            Visitor.TraverseDecl(*b);
       return true;
    }

 private:
    MyASTVisitor Visitor;
};

...
```

В примере можно увидеть, что классы `ASTConsumer`, `RecursiveASTVisitor`, `SourceLocation` и `Rewriter` действительно очень удобно использовать для реализации правил трансляции.

3. Использование семантической разметки. Верификация методом Хоара хорошо работает только в идеальном случае, когда программа и ее спецификации корректны, теория проблемной области полна, и автоматический доказатель теорем обладает достаточной мощью. Тогда программа может быть верифицирована автоматически с минимальным участием пользователя. Но на практике некоторое из перечисленных условий (а зачастую, все) бывает не выполнено. В этом случае пользователь получает набор недоказанных/ложных условий корректности, но, как правило, не получает информации о причинах неудачи. Он должен проанализировать условия, проинтерпретировать их составные части и соотнести их с исходными местами в программе.

Концепции	**Примеры меток**	**Аспекты условий корректности**
Гипотезы • Утверждения • Управляющие предикаты	asm_pre, asm_inv, then, while_t	*Гипотезы* отражают предположения о том, что некоторые логические утверждения выполнены в тех или иных точках программы. Это могут быть как исходные предусловия, так и управляющие выражения операторов наподобие `while` и `if`.
Заключения	ens_post, ens_inv_iter	*Заключения* отражают основное предназначение условий корректности, состоящее в *гарантировании* выполнения тех или иных утверждений в заданных местах программы.
Уточнители • Подстановки • Присваивания	sub, upd, alloc, init	*Уточнители* вводят более детальную характеризацию для гипотез и заключений, отражая способ получения под-формулы. Как правило, они соответствуют выражениям и операциям в программе.
Индуктивные уточнители	call, pres_inv	*Индуктивные уточнители* отражают второстепенное предназначение условия корректности. Например, условия корректности для вложенного цикла концептуально связаны с предназначением условий и для охватывающего цикла.

Таблица 1: Роль под-формул условия корректности для верификации

На базе идеи Денни и Фишера [7] предлагается расширить правила Хоара с помощью «семантической разметки» так, чтобы само исчисление порождало объяснения для условий корректности. Эти структурированные метки прикрепляются к формулам в правилах Хоара. Метки, пройдя различные этапы обработки, извлекаются из окончательных условий корректности и преобразуются в объяснения на естественном языке (в данной работе — на английском).

Концепции и метки. При определении меток стоит разделить две задачи. Задача локализации ошибок достаточно проста. Метка, очевидно, должна содержать информацию о месте срабатывания правила Хоара (имя файла, строку и, возможно, колонку). Более сложная задача объяснения условия корректности требует понимания *роли* условия в процессе верификации.

В первом приближении можно воспользоваться тем фактом, что условия корректности, как правило, имеют вид Хорновских клозов: $H_1 \wedge \ldots \wedge H_n \Rightarrow C$. Тогда заключение C можно охарактеризовать как *предназначение* этого условия. Далее, для осмысленного объяснения структуры условия корректности необходимо предложить более детальную характеризацию его подформул. Базисная информация при генерации объяснения пред-

ставляет собой множество *концепций*, зависящее от того, какой аспект условий корректности необходимо объяснить. В табл. 1 дается краткий обзор концепций, предложенных к настоящему времени. Более подробный обзор можно найти в [15].

Мы используем нотацию из [7] для вывода помеченных термов вида $\ulcorner t \urcorner^l$, где каждый терм t может быть помечен некоторой меткой l. Сами метки имеют вид $c(o, n)$. Концепция c (см. примеры из второй колонки в табл. 1) описывает роль данного помеченного терма. Местоположение o отражает происхождение терма (номер строки или диапазон строк). Список меток (возможно, пустой) n записывает некоторую уточняющую информацию.

Примеры правил Хоара для C-kernel с метками. Основным достоинством данного метода является то, что семантические метки не меняют структуру исходных правил для C-kernel. Мы просто «украшаем» термы метками. Метка может быть добавлена к отдельной подформуле, группе формул или тройке Хоара. Обычно, метки над подформулами соответствуют гипотезам, заключениям и уточнителям. Группы формул и тройки помечаются индуктивными уточнителями.

Исходная логика Хоара для C-kernel была представлена в [3], а ее вариант с метками в [15]. Рассмотрим для примера правило для композиции:

$$\frac{\mathcal{E}nv \Vdash \langle\, P \,\rangle\, S_1 \,\langle\, \ulcorner R \urcorner^{\mathsf{ens\_post}} \,\rangle \qquad \mathcal{E}nv \Vdash \langle\, \ulcorner R \urcorner^{\mathsf{asm\_pre}} \,\rangle\, S_2 \,\langle\, Q \,\rangle}{\mathcal{E}nv \Vdash \langle\, P \,\rangle\, S_1\, S_2 \,\langle\, Q \,\rangle}$$

Таким образом, метки $\mathsf{ens\_post}$ and $\mathsf{asm\_pre}$ отражают роли промежуточного утверждения R. Необходимо гарантировать (ensure), что оно является постусловием в первой тройке, а также необходимо предположить (assume), что оно является предусловием во второй.

Переписывание помеченных термов и преобразование меток в объяснения. Условия корректности (как обычные, так и помеченные) становятся громоздкими при выводе и должны быть упрощены до этапа доказательства. Цель этапа переписывания состоит в удалении избыточных меток и минимизации области действия оставшихся; при этом необходимо сохранить достаточное количество меток для объяснения неудач с доказательством.

Помеченные правила переписывания базируются на обычных непомеченных правилах (например, см. [14]), но не переиспользуют их в неизменном виде, поскольку метки требуют более аккуратной работы. Например, с одной стороны, мы можем безопасно удалить метки из тождественно истинных подформул, поскольку они не требуют объяснения. С другой стороны, метки из тождественно ложных подформул удаляются избирательно, чтобы сохранить информацию для объяснения неудачи.

Окончательной стадией работы с метками является преобразование их в объяснения доступные человеку. Составные шаги излагаемого подхода представлены на рис. 2 и включают в себя: (1) нормализацию условий корректности с использованием помеченной си-

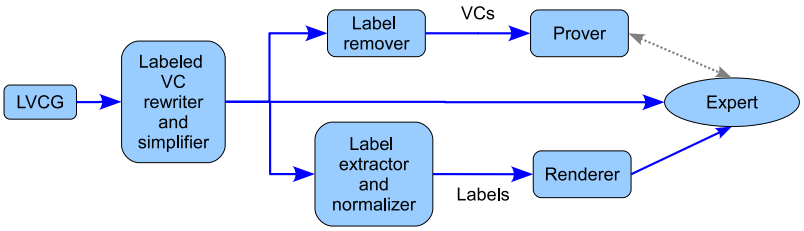


Рис. 2: Метки и помеченные условия корректности

стемы перевода; (2) извлечение меток; (3) нормализацию самих меток с целью их соответствия шаблонам порождения объяснения; (4) генерацию текстовых объяснений с помощью набора шаблонов (реализованы на языке ML).

3.1. Пример. Рассмотрим следующую программу с внесенной ошибкой:

```
void NegateFirst(int ia[], int Length)
{
    //@ pre ...
    int i;
    //@ inv ...
    for (i = 0; i < Length; i++) {
        if (ia[i] < 0) {
            ia[i] = ia[i];
            break;
        }
    }
    //@ post ...
}
```

Оригинальная программа [14] должна была найти первый отрицательный элемент массива, изменить его знак и завершить работу. Как видно, в присваивании `ia[i] = -ia[i]` потерян знак "минус".

Спецификации, отражающие требуемое правильное поведение программы, выглядят как

$$\text{pre}: \quad \exists old : \texttt{int[]}.\ \mathrm{MD}(\mathrm{MeM}(\texttt{ia})) \neq \mathsf{null} \wedge \mathrm{MD}(\mathrm{MeM}(\texttt{ia})) = \mathrm{MD}(\mathrm{MeM}(old))$$

$$\begin{aligned}\text{post}: \quad & \forall i.\ (0 \leq i \leq \mathrm{MD}(\texttt{Length}) \Longrightarrow \\ & ((\mathrm{MD}(\mathrm{MeM}(old, i)) < 0 \wedge \\ & (\forall j.\ 0 \leq j < i \Rightarrow \mathrm{MD}(\mathrm{MeM}(old, j)) \geq 0)) \Rightarrow\end{aligned}$$

$$\mathrm{MD}(\mathtt{MeM}(\mathtt{ia}, i)) = -\mathrm{MD}(\mathrm{MeM}(old, i)) \wedge$$
$$old[i] \geq 0 \Rightarrow \mathrm{MD}(\mathtt{MeM}(\mathtt{ia}, i)) = \mathrm{MD}(\mathrm{MeM}(old, i)))$$

$$\begin{aligned}\mathrm{inv}\ :\ & 0 \leq \mathrm{MD}(\mathtt{i}) \leq \mathrm{MD}(\mathtt{Length}) \wedge \\ & (\forall j.\ 0 \leq j < \mathrm{MD}(\mathtt{i}) \Rightarrow \\ & \quad (\mathrm{MD}(\mathrm{MeM}(\mathtt{ia}, j)) \geq 0 \wedge \mathrm{MD}(\mathrm{MeM}(\mathtt{ia}, j)) = \mathrm{MD}(\mathrm{MeM}(old, j))) \ .\end{aligned}$$

Транслятор, описанный в разд. 2.2, порождает эквивалентную программу на C-kernel:

```
1   void NegateFirst(int ia[], int Length) {
2       //@ pre ...
3       auto int i;
4       i=0;
5       while(i < Length){
6           //@ inv ...
7           if (ia[i]<0){
8               ia[i] = ia[i];
9               goto L;
10          }
11          else {}
12          auto int* q1;
13          q1 = &i;
14          *q1 = *q1 + 1;
15      }
16      L:;
17      //@ post ...
18  }
```

Номера строк не порождаются при трансляции, а добавлены для удобства.

Генератор условий корректности порождает 5 помеченых условий и одну тождественно истинную тройку Хоара. Четыре из условий автоматически доказываются в программе Simplify. Однако условие, соответствующее ветви *then* условного оператора, оказывается

ложным. Оно имеет вид

$$\left[\begin{array}{ll} \ulcorner \mathrm{inv}(\mathrm{MD} \leftarrow \mathrm{MD}_1) \urcorner^{\mathsf{asm\_inv}(6)} & \wedge \\ \ulcorner \mathrm{MD}_1(\mathrm{MeM}(\mathtt{i})) < \mathrm{MD}_1(\mathrm{MeM}(\mathtt{Length})) \urcorner^{\mathsf{while\_t}(6)} & \wedge \\ \ulcorner \mathrm{MD}_1(\mathrm{MeM}(\mathtt{ia}), \mathrm{MD}_1(\mathrm{MeM}(\mathtt{i}))) < 0 \urcorner^{\mathsf{then}(7)} & \wedge \\ \ulcorner \mathrm{MD} = \mathit{upd}(\mathrm{MD}_1, (\mathrm{MeM}(\mathtt{ia}), \mathrm{MD}_1(\mathrm{MeM}(\mathtt{i}))), & \\ \qquad \mathrm{MD}_1(\mathrm{MeM}(\mathtt{ia}), \mathrm{MD}_1(\mathrm{MeM}(\mathtt{i})))) \urcorner^{\mathsf{upd}(8)} & \\ \Rightarrow & \\ \ulcorner \mathrm{post} \urcorner^{\mathsf{ens\_inv}(9,\mathrm{L})} & \end{array}\right]^{\mathsf{pres\_inv}(6..10)}$$

В попытке его доказать мы заметим, что конъюнкт

$$\ulcorner \mathrm{MD} = \mathit{upd}(\mathrm{MD}_1, (\mathrm{MeM}(\mathtt{ia}), \mathrm{MD}_1(\mathrm{MeM}(\mathtt{i}))), \mathrm{MD}_1(\mathrm{MeM}(\mathtt{ia}), \mathrm{MD}_1(\mathrm{MeM}(\mathtt{i})))) \urcorner^{\mathsf{upd}(8)}$$

противоречит заключению. Метка `upd(8)` указывает, что проблема связана с обновлением элемента массива в строке 8. В свою очередь, связи, хранимые в объекте класса `SourceLocation` для данного узла `AST` (см. разд. 2.2), позволят указать и исходное присваивание в C-light программе.

Конечно, для этой тривиальной программы условие корректности и ложный конъюнкт оказываются достаточно обозримыми. Для реальных программ формулы значительно разрастаются. И здесь снова на выручку приходят семантические метки. Вместо анализа структуры условия корректности или его доказательства мы можем просто извлечь метки, нормализовать их и применить трансляционные шаблоны для порождения объяснения на естественном языке. В данном примере получается следующее:

```
This VC corresponds to lines 6-10 in the function "NegateFirst". Its
purpose is to contribute the loop invariant preservation on each
iteration.
Hence, given
  - assumption that loop invariant holds at line 6,
  - assumption that the loop condition holds at line 6,
  - assumption that "then"-branch is chosen at line 7,
  - substitution for MD
    originating in array update at line 8,
show that label invariant holds at line 9.
```

Пояснение для «подозрительной» подформулы выделено рамкой.

4. Родственные работы. Среди большого числа проектов по верификации Си-программ (подробный обзор в [14]) отметим два проекта, в которых также используется

трансляция в промежуточный язык. Во-первых, проект WHY/Frama-C [8], развиваемый в INRIA. В нем исходные программы транслируются в языки WHY и CIL. Основная цель такой трансляции — генерация условий корректности независимо от доказателя теорем.

Во-вторых, проект VCC (A Verifier for Concurrent C), развиваемый в Microsoft Research [6]. Программы транслируются в логические формулы с помощью инструмента Boogie, который сочетает в себе промежуточный язык Boogie PL и генератор условий корректности. В качестве недостатка обоих проектов в сравнении с нашим отметим отсутствие формального доказательства корректности трансляции.

В сравнении с проектами по верификации работы по объяснению условий корректности и формальной локализации ошибок менее многочисленны. Наибольший интерес представляют следующие три. Во-первых, проект Centaur [9]. В нем генерируемые условия корректности анализировались для поиска условных выражений, которые использовались в исходных условных операторах и циклах. Для поиска использовались некоторые алгоритмы из области отладки программ.

Более позднее исследование [7] во многом повлияло на данную работу. Денни и Фишер предложили расширить правила Хоара с помощью меток с целью автоматического порождения объяснений для условий корректности. Объяснения можно легко настраивать для отражения различных аспектов условий корректности. Подход полностью декларативный и для порождения объяснения анализируются только метки, а не логический смысл самих условий или их доказательства.

Наконец, Лейно и др. [12] также расширили базовую логику за счет меток, представляющих семантическую информацию, пригодную для объяснения. В их случае метки вводятся на этапе трансляции в промежуточный язык, который в дальнейшем обрабатывается стандартным безметочным генратором.

В качестве недостатка всех трех перечисленных проектов отметим использование модельных входных языков, более простых, чем Си.

Также отметим некоторые работы по трансляции Си-программ. В [5] описана система `ctt` (Code Transformation Tool), предназначенная для трансформации Си-программ с целью распараллеливания и оптимизации. В [11] описана программа Linsert, транслирующая Си-программу в эквивалентную программу без побочных эффектов. Заметим, что хотя сами алгоритмы перевода имеют формальную основу, доказательства эквивалентности перевода отсутствуют (либо просто базируются на интуитивной трактовке стандарта). Более формальное обоснование предложено в [10], но оно связано не с формальной семантикой Си, а с сохранением тестовых покрытий.

5. Заключение. Дедуктивная верификация программ была задумана как способ достоверного доказательства корректности программы в отличие от тестирования, которое

имеет более вероятностный характер. Закономерно возникает вопрос ее надежности. И если теоретические основания верификации давно обоснованы в классических работах, то на практике все упирается в корректность реализации системы верификации. Очевидно, что идеальным решением было бы написать систему на целевом языке и проверить ее своими же средствами, т.е. получить «верифицированный верификатор». Для такого сложного языка, как Си, это нетривиальная задача. Основных препятствий здесь два.

Во-первых, любая осмысленная Си-программа использует стандартную библиотеку, которая не только не верифицирована, но даже не обладает полной формальной спецификацией. Шаги в этом направлении стали предприниматься только в последние годы. Помимо экспериментов в проекте WHY/Frama-C, не представленных в литературе, работа ведется и в рамках проекта C-light [16].

Во-вторых, формальная база дедуктивной верификации относится только к порождению условий корректности и их доказательству. Процессы же интерпретации результатов верификации пользователем (в особенности отрицательных результатов) и локализации ошибок формализованы слабо. Данная статья представляет обзор наших разработок на пути решения этой задачи. В соответствии с двухуровневой схемой верификации в проекте C-light работа ведется по двум направлениям.

Первое направление касается реализации транслятора из C-light в C-kernel. В частности, используется новый инструментарий. При этом для той части транслятора, которая выразима на языке C-light, будут предложены ACSL-спецификации, что позволит в будущем перейти к его верификации. Также для задачи локализации ошибок реализуется возможность обратной трансляции из C-kernel в C-light, что отсутствовало в предыдущих прототипах транслятора. Решение об использование инструментария Clang для этой цели оказалось оправданным. Оно позволило существенно сократить усилия при разработке транслятора. Поэтому можно рекомендовать при решении сходных задач трансляции применять инструментарий Clang.

Что касается второго этапа, то в аксиоматическую семантику языка C-kernel введено расширение в виде семантических меток. Структурированные метки отражают роли подтермов условий корректности в контексте верификации, а также аккумулируют в себе информацию об исходных фрагментах программы. Метки играют важную роль в процессе упрощения условий корректности, но основное их предназначение — это сохранение информации об исходных фрагментах программы в автоматическом доказателе и генерация объяснений для условий корректности на естественном языке.

Список литературы

1. Влиссидес Д., Гамма Э., Джонсон Р., Хелм Р. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2012. — 368 с.
2. Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В. На пути к верификации C программ. Часть 3. Перевод из языка C-light в язык C-light-kernel и его формальное обоснование. — Новосибирск, 2002. — 82 с. — (Препр. / РАН. Сиб. Отдние. ИСИ; N 97).
3. Промский А.В. Формальная семантика C-light программ и их верификация методом Хоара // Диссертация на соискание ученой степени кандидата физико-математических наук. — Новосибирск, 2004. — 157 с.
4. Baudin P., Filliâtre J.C., Marché C., Monate B., Moy Y., Prevosto V. ACSL: ANSI/ISO C Specification Language. [Электронный ресурс]. URL: `http://www.frama-c.cea.fr/download/acsl_1.4.pdf`
5. Boekhold M., Karkowski I., Corporaal H., Cilio A. A programmable ANSI C code transformation engine. — Delft, 1998. — (Tech. Rep. / Delft University of Technology; N 1-68340-44(1998)-08).
6. Cohen E., Dahlweid M., Hillebrand M.A., Leinenbach D., Moskal M., Santen T., Schulte W., Tobies S. VCC: A Practical System for Verifying Concurrent C // Proc. TPHOLs 2009. — LNCS. — 2009. — Vol. 5674. — P. 23-42.
7. Denney E., Fischer B. Explaining Verification Conditions // Proc. AMAST 2008. — LNCS. — 2008. — Vol. 5140. — P. 145–159.
8. Filliâtre J.C., Marché C. Multi-prover verification of C programs // Proc. ICFEM 2004. — LNCS. — 2004. — Vol. 3308. — P. 15–29.
9. Fraer R. Tracing the origins of verification conditions. — Rocquencourt, 1996. — 17 p. — (Rapp. / INRIA; N 2840).
10. Harman M., Hu L., Hierons R., Wegener J., Sthamer H., Baresel A., Roper M. Testability transformation // IEEE Trans Software Eng. — 2004. — N 30(1). — P. 3–16.
11. Harman M., Hu L., Munro M., Zhang X.Side-effect removal transformation // Proc. IEEE Intern. Workshop Program Comprehension. — IEEE Computer Society Press, 2001. — P. 310–319.
12. Leino K.R.M., Millstein T., Saxe J.B. Generating error traces from verification condition counterexamples // Science of Computer Programming. — 2005. — Vol. 55, N 1–3. — P. 209–226.
13. Programming languages — C: ISO/IEC 9899:1999. — 1999. — 566 p.
14. Promsky A.V. Towards C-light program verification: Overcoming the obstacles // Proc.

PU-2009, 19–23 June, Altai Mountains, Russia, 2009. — P. 53–63.

15. Promsky A.V. Error-tracing semantics for C-kernel // Bull. Nov. Comp. Center, Comp. Science, 31 (2010). P. 123–138.
16. A. Promsky. C-light Program Verification: Error Tracing and Library Specification // Proc. Second Workshop "Program Semantics, Specification and Verification: Theory and Applications". — St. Petersburg, Russia, June 12–13, 2011. — P. 83–92.
17. Promsky A.V. Verification Condition Understanding // Proc. Ershov Informatics Conference (PSI Series, 8-th Edition). — June 27 – July 1, 2011, Akademgorodok, Novosibirsk, Russia. — P. 295–300.
18. Promsky A.V. A formal approach to the error localization // Preprint 169, A. P. Ershov Institute of Informatics Systems, Novosibirsk, Russia, 2012.

UDK: 519.681

Title: An integrated approach to the error localization during C program verification

Author(s):

Dmitry Alexandrovich Kondratyev (Novosibirsk State University),

Alexey Vladimirovich Promsky (A.P. Ershov Institute of Informatics Systems)

Abstract: Maximal employment of formal and sound methods is one of the distinctive features of C-light project. This concerns not only the fundamental verification bases, like Plotkin's operational semantics or Hoare's logics, but also implementation aspects. The localization of possible errors is one of them. In the majority of known verification systems such localization is simply coded by developers as a part of system functionality without any formal basis for it. The recent reinforcement of the C-light project by the semantical labeling technique and the choice of LLVM/Clang for the system input block allowed us to obtain some new results, which are surveyed in this paper.

Keywords: semantics, specification, verification, translation, error localization, labeling, the C language, C-light, LLVM, Clang

УДК: 004.932.4

Название: Исследование стрип-метода обработки сигналов и изображений

Автор(ы):

Рассохина А.А. (Новосибирский государственный университет)

Аннотация: В статье рассмотрен стрип-метод линейного предыскажения сигналов и изображений с использованием нескольких видов матриц преобразования. Определены матрицы, которые достаточно хорошо минимизируют амплитуду помехи. Исследована зависимость качества восстановления данных от местоположения и размера помехи. Выявлена возможность использования стрип-метода в сочетании с компрессией, основанной на разреживании изображения, и разработан алгоритм, реализующий это сочетание.

Ключевые слова: стрип-метод, помеха, обработка изображений, помехоустойчивость, матрица Адамара, восстановление данных

1. Введение. Стрип-метод является одним из методов, используемых для хранения и помехоустойчивой передачи сигналов и изображений [2]. Его суть заключается в преобразовании сигнала или изображения на передающем конце путем «разрезания» его на фрагменты равной длины или площади соответственно, формирования их линейных комбинаций и обратного «склеивания». Преобразованный сигнал (изображение) передается по каналу связи, где подвергается воздействию помехи. Ее действие может приводить к искажению или полной потере отдельных фрагментов передаваемого сообщения. На приемном конце выполняется обратное преобразование, в результате которого происходит восстановление сигнала (изображения). Если обеспечить равномерное распределение помехи по всей длине или площади передаваемых данных, то произойдет значительное ослабление ее амплитуды и будет достигнуто приемлемое качество всех участков восстановленного сообщения.

Преимущество данного метода заключается в том, что на передающем конце осуществляется линейное комбинирование всех фрагментов исходного сигнала или изображения. Это приводит к тому, что каждый фрагмент передаваемого сообщения несет информацию обо всех без исключения фрагментах исходного сообщения, что позволяет в случае потери или повреждения большого числа фрагментов восстановить весь сигнал или все изображение с наименьшими искажениями. В этом смысле стрип-метод напоминает голограмму. Классические алгоритмы фильтрации в случае такого рода помех могут привести к побочным эффектам изменения незашумленных участков изображения.

Стрип-преобразование широко используется при передаче сигналов со спутников, т.к. в ионосфере может происходить кратковременная потеря связи, при которой целые участки

сигнала полностью пропадают. Также стрип-метод может быть использован в базах данных изображений и в других компьютерных системах хранения, обработки и передачи визуальной информации.

Основной целью является исследование вопроса, насколько качественно происходит восстановление сигналов и изображений для различных матриц и различных типов помех. Также рассматривается возможность применения стрип-преобразования изображений в сочетании с компрессией, что уменьшает вычислительные затраты на хранение и передачу данных. Для проведения исследования были разработаны два программных средства, моделирующих стрип-преобразование сигналов и изображений, а также реализована библиотека отдельных модулей, которые могут быть встроены в другие программные системы.

2. Одномерное преобразование. Одномерное стрип-преобразование используется для передачи сигналов [2] и включает в себя три этапа: прямое преобразование, передачу по каналу связи и обратное преобразование.

Первый этап преобразования заключается в разбиении длинного исходного сигнала $x(t),\ 0 \le t \le T$ на n участков равной длительности $h = T/n$ и получению n коротких сигналов вида

$$x_1(t) = x(t),\ x_2(t) = x(t + T/n), \ldots, x_n(t) = x(t + (n-1)T/n), \qquad (1.1)$$

$$0 \le t \le T/n.$$

Из них формируется *n*-мерная вектор-функция

$$X(t) = \begin{pmatrix} x_1(t) \\ \ldots \\ x_n(t) \end{pmatrix},\ 0 \le t \le T/n. \qquad (1.2)$$

При помощи ортогональной матрицы A вектор $X(t)$ преобразуется в вектор $Y(t)$:

$$Y(t) = AX(t) = \begin{pmatrix} y_1(t) \\ \ldots \\ y_n(t) \end{pmatrix},\ 0 \le t \le T/n. \qquad (1.3)$$

Компоненты вектора $Y(t)$ имеют вид: $y_i(t) = A_i X(t);\ i = 1, 2, \ldots, n,$ где A_i – i-я строка A. Заметим, что в результате такого преобразования каждая компонента $y_i(t)$ несет в себе информацию обо всех компонентах $x_i(t)$. Именно это обеспечивает высокую помехоустойчивость передачи и восстановление в случае, если часть сигнала исказилась.

На втором этапе сигнал $Y(t)$ передается по каналу связи, при этом часть сигнала искажается или пропадает.

Далее, на приемном конце вектор $Y(t)+\Delta Y(t)$, где $\Delta Y(t)$ – вектор-функция помехи, подвергается обратному преобразованию с использованием матрицы A^{-1}:

$$A^{-1}(Y(t)+\Delta Y(t))=X(t)+\Delta X(t). \quad (1.4)$$

При этом погрешность восстановления сигнала определяется по формуле

$$\Delta X(t)=A^{-1}\Delta Y(t). \quad (1.5)$$

Если одна из компонент вектора $\Delta Y(t)$ отлична от нуля, то все компоненты вектора погрешности $\Delta X(t)$ будут отличны от нуля, таким образом, происходит распределение помехи по всей длине сигнала.

3. Двумерное преобразование. Аналогично одномерному случаю, первый этап двумерного преобразования, применяемого при передаче изображений, состоит в разбиении исходного изображения на *N* одинаковых по размеру прямоугольных фрагментов [2].

Обозначим число горизонтальных и вертикальных полосок, на которые «разрезается» изображение, через m и n; тогда $N=mn$. Далее осуществляется линейное комбинирование фрагментов по следующей схеме.

Исходное изображение, разбитое на фрагменты, рассматривается как блочная матрица X размера $m\times n$, которая умножается на ортогональную матрицу B размера $m\times m$ слева и на ортогональную матрицу A размера $n\times n$ справа: $Z=BXA$ (двустороннее преобразование).

В этом случае имеет место умножение числовых матриц *A, B* на блочную матрицу *X*, элементами которой являются фрагменты изображений. Умножение осуществляется по следующим правилам:

1. Сложение блоков (фрагментов). Сложение отдельных блоков (фрагментов) матриц изображений производится путем суммирования соответствующих элементов блоков. Эта операция аналогична сложению двух матриц одинакового размера.

2. Умножение фрагмента на число. Операция производится путем умножения каждого пикселя фрагмента на число. При этом изменяется яркость фрагмента в целом. Операция аналогична умножению матрицы на число.

3. Умножение блочной матрицы на числовую матрицу. Такое умножение производится аналогично обычному перемножению числовых матриц по правилу «строка на столбец» с учетом использования первых двух операций.

Рассмотренный подход обеспечивает полное «перемешивание» фрагментов изображения: каждый фрагмент преобразованного изображения содержит информацию обо всех *mn* фрагментах исходного изображения.

Соответственно, обратные преобразования, выполняемые на приемном конце канала связи, описываются формулой: $X = B^{-1}ZA^{-1}$. Рассмотрим их подробнее.

В канал связи передается изображение $Z = BXA$. Далее к нему добавляется импульсная помеха Δ (блочная матрица размера $m \times n$), в результате чего на выходе имеем изображение $\hat{Z} = Z + \Delta$. На приемном конце $\hat{Z}$ подвергается обратному преобразованию для получения матрицы результирующего изображения. Оно описывается формулой

$$\hat{X} = B^{-1}\hat{Z}A^{-1} = B^{-1}(Z + \Delta)A^{-1} = B^{-1}ZA^{-1} + B^{-1}\Delta A^{-1} = X + B^{-1}\Delta A^{-1}. \qquad (2.1)$$

При использовании данного метода удобно взять матрицы *A* и *B* равными, так как это упростит вычисления, а также сэкономит память, если применять метод на практике.

Тогда уравнение прямого преобразования примет вид $Z = AXA$, где A – ортонормированная матрица. Упростится также уравнение обратного преобразования

$$\hat{X} = X + A^T\Delta A^T. \qquad (2.2)$$

Заметим, что все изложенное выше относилось к случаю черно-белых изображений, которые могут быть представлены одной матрицей яркости, состоящей из значений от 0 до

255. Именно эта матрица и подвергается фрагментации при стрип-преобразовании. В случае цветных изображений можно использовать трехслойную матрицу RGB и подвергать стрип-преобразованию каждый из слоев.

4. Выбор матриц преобразования. Матрицы стрип-преобразования выбираются исходя из того, что нужно добиться равномерного распределения помехи по сигналу или изображению в результате обратного декодирования на приемном конце канала связи. Это позволит восстановить информацию об искаженных или потерянных фрагментах. Возникает задача определения вида матрицы преобразования A, которая обеспечит минимизацию амплитуды помехи в восстановленном сообщении.

Легко показать, что амплитуда помехи определяется величиной максимального элемента матрицы A. Таким образом, задача поиска оптимальных матриц преобразования сводится к задаче поиска ортогональных матриц, у которых максимальный элемент минимален. Этому свойству удовлетворяют, в частности, нормированные матрицы Адамара [5], существующие для n, кратных четырем, а также С-матрицы, существующие для n, кратных двум.

5. Результаты преобразований. Рассмотрим изображение размера 1024×1024, представленное на рис. 4.1 (а). Разобьем его на 8 частей по горизонтали и на столько же по вертикали. Полученную блочную матрицу X, содержащую 64 фрагмента, подвергнем двустороннему стрип-преобразованию с помощью матрицы Адамара 8-го порядка:

$$Z = \frac{1}{8} A^T X A \tag{4.1}$$

В результате такого преобразования произошло перемешивание фрагментов.

Рис. 4.1

а) исходное изображение *б) фрагментированное изображение*

На рис. 4.2 приведено изображение, полученное в результате прямого стрип-преобразования, с добавленной к нему помехой размера 50% от размера всего изображения и результат его восстановления.

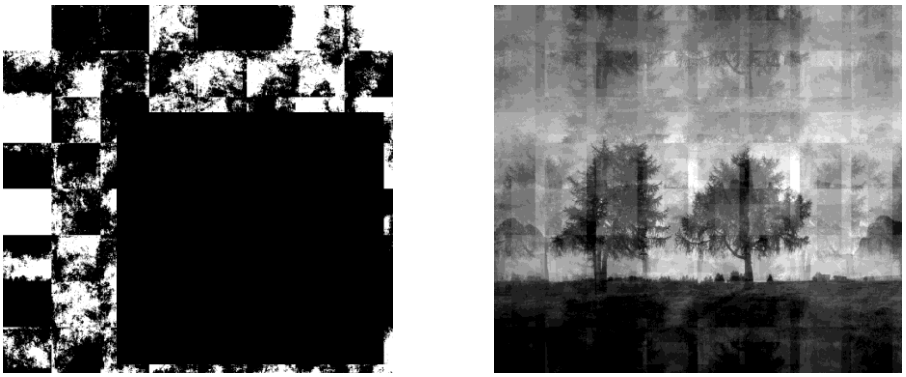


Рис. 4.2

а) фрагментированное изображение с помехой *б) результат преобразования*

Замечательным свойством стрип-метода является то, что даже если размер помехи составляет 80% от всего изображения, его все еще можно восстановить, хотя и с большими искажениями.

В работе рассматриваются три основных критерия оценки погрешности восстановления:

- среднеквадратичное отклонение значений пикселов (L_2 мера, или root mean square – RMS):

$$d(x,y)=\sqrt{\sum_{i,j=1}^{n}\frac{(x_{ij}-y_{ij})^2}{n^2}}; \tag{4.2}$$

- максимальное отклонение (RSDmax):

$$d(x,y)=\max_{i,j}\left|x_{ij}-y_{ij}\right|; \tag{4.3}$$

- отношение сигнала к шуму (peak-to-peak signal-to-noise ratio — PSNR):

$$d(x,y)=10\log_{10}\frac{255^2n^2}{\sum_{i,j=1}^{n}(x_{ij}-y_{ij})^2}. \tag{4.4}$$

Каждый из критериев имеет свои недостатки, и выбор между ними осуществляется на основе требований, предъявляемых к изображениям. Ниже в табл. 4.1 приведены значения

ошибки восстановления изображения на рис. 4.1 для каждого из критериев в зависимости от размера помехи.

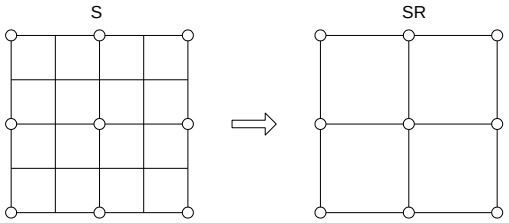
Таблица 4.1 Зависимость погрешности восстановления от размера помехи

	RMS	RSDmax	PSNR
100×100	4.8732	14.9219	34.3746
200×200	8.7232	35.7500	29.3173
300×300	12.1715	52.0625	26.4239
500×500	18.4796	94.5000	22.7969
700×700	35.0431	166.0938	17.2387
900×900	44.3808	252	15.1869

Значение имеет не только размер помехи, но и то, какую часть изображения она охватывает. В силу того, что первые строка и столбец матриц Адамара состоят целиком из единиц, а остальные строки и столбцы имеют равное количество 1 и −1, наибольшие значения элементов блочной матрицы, полученной в результате прямого стрип-преобразования, будут сконцентрированы в блоках, составляющих ее первую строку и первый столбец. При этом левый верхний блок несет в себе максимум информации об исходном изображении. Следовательно, если помеха охватывает этот блок, изображение восстановится с бóльшими искажениями. Если помеха расположена на месте блоков, составляющих первую строку или первый столбец, изображение восстановится с меньшими искажениями и, наконец, если она охватывает любые другие блоки, восстановление будет наилучшим. С помощью переупорядочения строк матрицы Адамара можно добиться концентрации максимума информации об исходном изображении в других блоках: например, таких, где вероятность появления помехи меньше.

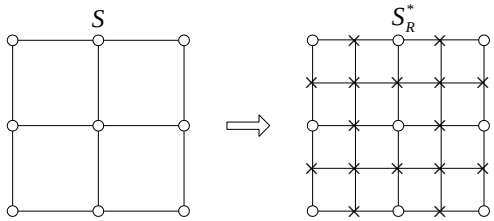
6. Стрип-метод в сочетании с компрессией. Исходя из того, что стрип-метод обладает высокой помехоустойчивостью и позволяет восстанавливать потерянные участки изображения, его можно использовать для компрессии. Для этого после прямого этапа стрип-преобразования применим разреживание изображения. В ходе разреживания размер матрицы яркости уменьшается в n раз, где n кратно четырем. Это позволяет хранить и передавать по каналу меньший объем данных. В канале к уменьшенному изображению добавляется помеха, после чего происходит его интерполяция до исходного размера, и уже к полученному после интерполяции изображению применяется обратное стрип-преобразование.

Пусть S-матрица яркости исходного изображения, S_R – матрица разреженного изображения. В матрицу S_R войдут только элементы нечетных строк и столбцов матрицы S, размер S_R в 4 раза меньше размера S.



При этом разреживание можно производить в несколько этапов, и каждый раз размер полученной матрицы становится в 4 раза меньше по сравнению с предыдущим. Целесообразно применять его не более трех раз.

Далее матрица S_R искажается помехой, после чего к ней применяется интерполяция. Результатом является матрица S_R^*.



Интерполяция производится отдельно для каждого блока, на которые разбивается матрица исходного изображения, т. к. после прямого преобразования происходит перемешивание блоков, и значения на их границах могут сильно отличаться. Следовательно, для восстановления значений матрицы S_R^* внутри или на границе конкретного блока используются значения только из этого блока.

Интерполяция блока размера $M \times M$ производится по следующим формулам:

$$S_R^*(i,j)=\begin{cases}\frac{1}{2}(S_R(i-1,j)+S_R(i+1,j)), \text{ если } i=2k,\ j=2l-1,\\ \qquad\qquad k=1,\ldots\frac{M}{2}-1,\ \ k=1,\ldots\frac{M}{2}\\ \frac{1}{2}(S_R(i,j-1)+S_R(i,j+1)), \text{ если } i=2k-1,\ j=2l,\\ \qquad\qquad k=1,\ldots\frac{M}{2},\ \ k=1,\ldots\frac{M}{2}-1\\ \frac{1}{4}(S_R(i-1,j-1)+S_R(i-1,j+1)+S_R(i+1,j-1)+S_R(i+1,j+1)),\\ \qquad\qquad \text{если } i=2k,\ j=2k, k=1,\ldots\frac{M}{2}-1\end{cases} \tag{5.1}$$

$$S_R^*(i,M)=S_R^*(i,M-1),\ \ i=1,\ldots,M \tag{5.2}$$

$$S_R^*(M,j)=S_R^*(M-1,j),\ \ j=1,\ldots,M\,. \tag{5.3}$$

После интерполяции матрица S_R^* подвергается обратному стрип-преобразованию.

Для тестирования сочетания стрип-метода и компрессии использовалось изображение размера 1024×1024. К нему применялось от одного до трех этапов разреживания с использованием стрип-метода и без него. После разреживания к изображению добавлялась помеха, составляющая 61% от его размера.

В качестве критериев потери качества изображения были выбраны среднеквадратическое отклонение значений пикселов (RMS)

$$RMS = \sqrt{\frac{\sum_{i=1,j=1}^{n,n} (x_{ij} - y_{ij})^2}{n^2}} \tag{5.4}$$

и максимальное отклонение (RSDmax)

$$RSD\max = \max_{i,j} \left| x_{ij} - y_{ij} \right|. \tag{5.5}$$

Результаты тестирования для трех этапов разреживания приведены в табл. 5.1.

Таблица 5.1 Погрешность восстановления изображения после разреживания с использованием стрип-метода и без него

Компрессия	1 этап	2 этап	3 этап
С использованием стрип-метода	RSDmax=165.5313 RMS =39.5621	RSDmax=178.9297 RMS = 41.0271	RSDmax =222.6250 RMS = 39.3739
Без использования стрип-метода	RSDmax = 255 RMS= 108.8013	RSDmax = 255 RMS = 107.7964	RSDmax =255 RMS=112.8662

Исходя из данных таблицы, можно сделать вывод, что использование стрип-метода позволяет уменьшить значение среднеквадратического отклонения пикселов в 2,6 – 2,8 раза.

7. Заключение. В данной статье рассмотрен стрип-метод, предназначенный для хранения и помехоустойчивой передачи сигналов и изображений. Основной задачей было оценить влияние разного рода помех на качество восстановления передаваемых данных и погрешность восстановления. В ходе работы были проведены эксперименты с несколькими видами матриц стрип-преобразования. Определены те из них, которые достаточно хорошо минимизируют амплитуду помехи, равномерно распределяя ее по всему сигналу или изображению, тем самым позволяя восстановить потерянные участки. Была выявлена зависимость степени восстановления от местоположения и размера помехи. Также была

рассмотрена возможность использования стрип-метода в сочетании с компрессией, которая показала хорошие результаты восстановления изображений.

В системе Matlab разработаны два программных средства, реализующих алгоритмы одномерного и двумерного стрип-преобразования с использованием разных типов матриц и видов помех, а также библиотека отдельных модулей, содержащая реализацию стрип-преобразования в сочетании с компрессией.

Список литературы

1. Ахмед Н., Рао К. Р. Ортогональные преобразования при обработке цифровых сигналов. Пер. с англ./ Под ред. И. Б. Фоменко, М.: Связь, 1980, С. 130 – 132.
2. Мироновский Л. А., Слаев В. А. Стрип-метод преобразования изображений и сигналов. СПб ГУАП, 2006.
3. Харкевич А. А. Борьба с помехами. М.: Наука, 1989, С. 19 – 21.
4. Хармут Х. Теория секвентного анализа. М.: Мир, 1980, С. 43 – 70.
5. Холл М. Комбинаторика. М.: Мир, 1970, С.283 – 293.
6. Ярославский Л. П. Введение в цифровую обработку изображений. М.: Сов. Радио, 1979, С. 22 – 27.

UDK: 004.932.4

Title: Investigation of strip-method for signal and image processing

Author(s):

Anastasia Alexandrovna Rassokhina (Novosibirsk State University)

Abstract: The article presents a strip-method of linear pre-distortion of signals and images using multiple types of transformation matrices. Matrices that are good enough to minimize the amplitude of the noise were determined. The dependence of the quality of data recovery on the location and size of the noise were investigated. Also we found the possibility of using the strip-method in combination with image compression and developed an algorithm that implements this approach.

Keywords: strip-method, noise, image processing, noise immunity, Hadamard matrix, data recovery

УДК: 004.89

Название: Подход к созданию исследовательской информационной системы с документально подтверждаемой информацией

Автор(ы):

Сидорова Е.А. (Институт систем информатики СО РАН),

Серый А.С. (Институт систем информатики СО РАН)

Аннотация: Данная статья посвящена проблемам интеллектуального доступа к информационным ресурсам. Приводится анализ информационных систем, поддерживающих работу с данными, либо представленными текстовыми документами и их фрагментами, либо заданными объектной моделью на основе онтологии предметной области. Авторами предложен подход к созданию системы, которая бы поддерживала оба вышеперечисленных способа представления информации, описана ее архитектура и схема хранилища данных.

Ключевые слова: информационная система, интеллектуальный доступ к данным, онтология, текст, аннотирование.

1. Введение. Современные информационные системы предоставляют пользователю информацию в виде «готового знания» — набора определенных фактов о действительности [5]. Однако в некоторых сферах деятельности, таких как юриспруденция, делопроизводство, научные исследования и т.д., требуется документальное подтверждение любой информации или факта. Это означает, что любой факт должен сопровождаться ссылкой на источник. Таким источником может быть как документ, так и отдельно взятый фрагмент документа, подтверждающий данный факт. Важной функцией в вышеупомянутых сферах деятельности становится также и проверка данных при поступлении в базу данных, контроль достоверности информации, ее актуальности во времени.

Работа с документами и их текстовыми фрагментами поддерживается системами класса корпус-менеджер [1,6], которые обеспечивают аннотирование или разметку текста, комплексную возможность просмотра контекстов в виде конкордансов, поиск в корпусе текстов, фильтрацию по различным основаниям, сбор статистики и т.п. Основное назначение таких систем — проведение комплексного исследования на обширном материале, представленном в корпусе документов.

На сегодняшний день существует множество различных систем для работы с текстами и корпусами. Одни из них предоставляют инструменты для разметки текста и поиска по корпусам, но, в то же время, не позволяя, например, выявить в тексте референциальные отношения. Другие создаются для извлечения информации и выделения терминов и кореферентных выражений, являясь узкоспециализированными в фиксированных

предметных областях. В качестве примера первых можно привести системы **Bonito** и **Xaira** (http://projects.oucs.ox.ac.uk/xaira). Первая представляет собой графический пользовательский интерфейс для корпуса-менеджера Manatee, основным назначением которого является обработка различных поисковых запросов и визуализация результатов поиска. Вторая предназначена для поиска в тексте на основе XML-разметки и не зависит от языка, т.к. в ее основе лежит объектная модель, позволяющая описывать сущности, а также задавать методы представления и поиска различных лингвистических ресурсов.

Представителями систем, реализующих другой подход, являются, к примеру, создаваемые с целью разработки и оценки методов извлечения информации и data mining аннотированные корпусы текстов (преимущественно англоязычных) по биомедицине и молекулярной биологии **GENIA** (http://www-tsujii.is.s.u-tokyo.ac.jp/GENIA/) и **BioInfer** (http://www.it.utu.fi/BioInfer). **GENIA** — это информационный ресурс на базе аннотированного биомедицинского корпуса, в котором выделены термины, отношения, ситуации, кореферентные выражения, а **BioInfer** — открытый ресурс для ручной разметки корпуса и связанных ресурсов и для извлечения информации в биомедицинской предметной области; элементами разметки здесь являются термины и связи между ними.

Данные системы эффективны при решении конкретных задач в специализированных (фиксированных) предметных областях, однако сфера их применения не может быть обобщена или расширена. Возможность многослойной лингвистической разметки корпусов текстов, а также визуализация разметки и средства фильтрации и поиска реализованы в системе UAM CorpusTool [13]. Она позволяет задавать проекты в виде набора файлов, к которому может быть применена единая система признаков разметки. Каждому признаку соответствует слой текстовой разметки. UAM CorpusTool позволяет аннотировать фрагменты текста без ограничения на вложенность, пересечение и разрывность. Система позволяет анализировать, фильтровать размеченные фрагменты корпуса и осуществлять поиск по размеченному корпусу. Разметка представлена в формате XML, а также хранится отдельно от текста в виде аннотаций [2,9]. Несмотря на то, что система находится в открытом доступе, она работает только с английскими словарями, что делает невозможным лексический анализ текстов на русском языке.

Наша работа, следуя курсу наиболее актуальных направлений исследований в области обеспечения интеллектуального доступа к информационным ресурсам, направлена на создание системы, которая бы совмещала стандартные способы представления информации на основе модели предметной области и представления этой же информации в виде совокупности текстовых фрагментов из различных документальных источников, в которых

она упоминается. Система будет снабжена инструментами как для информационного поиска, основанного на фактографическом представлении информации, так и для проведения исследований и анализа имеющейся информации. Достоверность информации, представленной в системе, может быть подтверждена ссылками на текстовые источники, в которых она упоминается.

Работа выполняется при финансовой поддержке Российского фонда фундаментальных исследований (грант №12-07-31216).

2. Общее описание подхода. Разрабатываемая система содержит информацию, представленную двумя типами данных: тексты на естественном языке и набор формально описанных фактов, упоминаемых в этих текстах или введенных пользователем вручную (или иным способом). При этом связи между фактом и текстами, являющимися его источниками, должны сохраняться. В этом и состоит идея «документального подтверждения данных», когда для некоторого факта можно подобрать множество текстов и указать конкретные цитаты, где упоминается данный факт. В сравнении с информационно-поисковыми системами, осуществляющими фактографическое индексирование текстов, предлагаемый подход имеет существенное отличие: один и тот же факт может быть представлен в различных документах. Можно рассматривать связь факта и текстов как ссылки на первоисточники. В качестве примера ситуации, в которой необходима ссылка на источник, можно привести наличие факта назначения срока наказания за некое противоправное деяние, подтверждением которого будет цитирование соответствующей статьи уголовного кодекса.

Фактически контент системы имеет двойственную природу и требует разработки методов представления, хранения и анализа знаний в едином информационном пространстве. В рамках предлагаемого подхода представление структурированных ресурсов базируется на описании предметной области в виде онтологии, определяющей основные понятия и отношения рассматриваемой области знаний. Информация, содержащаяся в интегрируемых текстовых ресурсах — документах и корпусах — порождает и дополняет объектную структуру представления данных.

При наличии возможности оценить достоверность того или иного факта на основе анализа его источников, а также возможности удобного и содержательного доступа к данным, такая информационная система будет полезна как аналитикам, так и лицам, принимающим решения. Включение в систему развитых средств анализа и визуализации информации из различных корпусов документов в виде конкорданса, т.е. совокупности контекстов фактов, окажется полезно лингвистам для проведения исследований семантических свойств текста, а также инженерам знаний, которые получат инструментарий для автоматизированного

создания лингвистических ресурсов и систем анализа текстов, в том числе для автоматического наполнения информационных систем. Для долговременного функционирования и развития такой системы необходимо обеспечить поддержание логической целостности и достоверности информации. Целостность контента обеспечивается онтологией, дающей полное и целостное описание предметное области, а также организацией хранилища данных и методов доступа к нему. Поддержание информации в актуальном состоянии обеспечивает специально разработанный механизм, на основе анализа поступающих текстовых данных и их экспертной оценки. Устаревшие и более не считающиеся достоверными факты автоматически исключаются из системы.

Создание единой инструментальной среды, включающей инструменты как хранения и поиска информации, так и поддержки исследований на основе текстовых материалов, позволяет говорить о создаваемом программном продукте как об *исследовательской информационной системе*.

3. Аннотация документа. Как было сказано выше, контент системы неоднороден. С одной стороны он описывается онтологией предметной области, с другой — формируется множеством семантически аннотированных текстовых документов (пример такой семантической аннотации можно посмотреть в [12]). Данные представлены в виде информационных объектов со своими атрибутами, связями и текстовыми источниками — фрагментами документов, в которых упоминаются объекты. Хранение этих двух по сути различных видов контента организовывается по-разному. Оригиналы документов (представляющие собой файлы определенного формата) хранятся в специальной директории файловой системы сервера. Тем не менее, важно, чтобы документы и информационные объекты были связаны друг с другом в единой базе данных ИИС. Структурой, организующей связывание текстовых фрагментов и информационных объектов, является аннотация.

3.1. Требования к хранилищу данных. Основные требования к базе данных ИИС формулируются нами следующим образом:

- база данных должна совмещать хранение онтологии предметной области, лингвистической онтологии (системы дополнительных лингвистических признаков для разметки текста) и контента в виде информационных объектов и аннотированных документов;
- база данных должна обеспечивать хранение настроек визуализации как для отображения информационных объектов, так и для раскраски информации в тексте;
- в базе данных должны быть так или иначе представлены документы, имеющиеся в системе, причем тексты документов должны входить в данное представление (текст

дополняется ссылкой на оригинал, находящийся в файловой системе), а документы — снабжаться метаинформацией, аналогично представлению информационного объекта;

- в базе данных должны быть представлены аннотации, обеспечивающие связывание фрагментов текста с информационными объектами;
- необходимо обеспечить хранение корпусов документов; каждый корпус может иметь свою собственную направленность (по теме и/или жанру) и индивидуальный набор лингвистических признаков для лингвистического аннотирования; корпуса создаются и поддерживаются пользователями-экспертами, обладающими соответствующими правами; права могут раздаваться индивидуально для каждого корпуса;
- хранение данных о зарегистрированных пользователях: логины, пароли, права доступа и права редактирования.

3.2. Хранилище аннотаций. База данных состоит из пяти концептуальных блоков: онтологии, блока данных (информационных объектов), лингвистического блока, блока аннотаций и блока пользовательских настроек. На Рис. 1 изображен фрагмент схемы БД, подробно описывающий блок аннотаций.

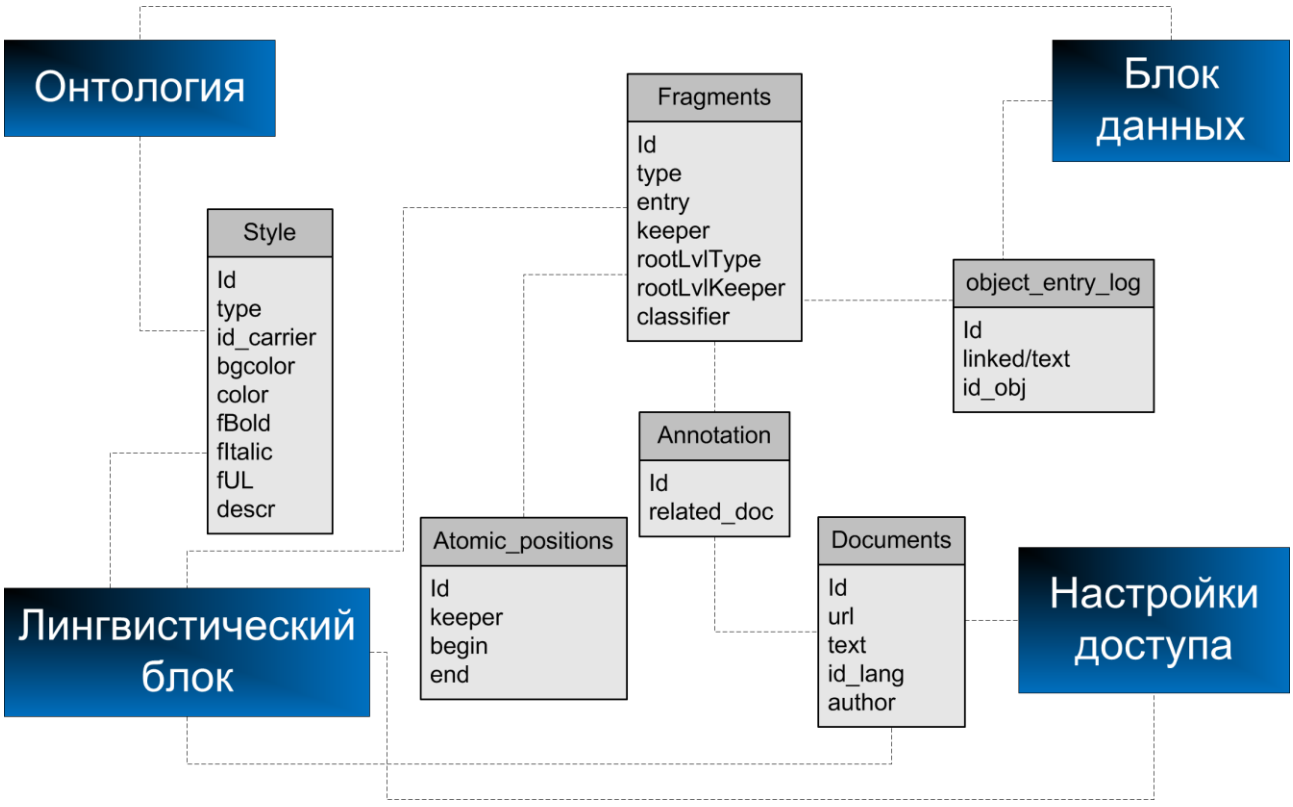


Рис. 1. Блок аннотаций базы данных системы

Рассмотрим таблицы, входящие в блок аннотаций.

3.3. Таблица документов. Таблица *Documents* содержит информацию об имеющихся документах, их тексты и ссылки на оригиналы документов, находящиеся в репозитории. Ниже перечислены столбцы данной таблицы.

id — уникальный идентификатор (ключ) документа,

url — ссылка на оригинал документа,

text — текст документа,

id_lang — язык документа,

author — автор документа (идентификатор пользователя системы, от имени которого был добавлен документ).

3.4. Таблица аннотаций. Таблица *Annotation* необходима для сведения информации обо всех размеченных фрагментах всех имеющихся документов. Она состоит всего из двух столбцов.

id — уникальный идентификатор (ключ),

related_doc — идентификатор документа, аннотированного данной аннотацией.

Аннотация документа может быть многоуровневой, т.е. содержать как лингвистическую разметку, в случае чего она будет связана с лингвистическим блоком, так и семантическую разметку информационных объектов, связь с которыми осуществляется через вхождение объекта в текст.

3.5. Таблица вхождений объектов. Таблица *object_entry_log* описывает все вхождения информационных объектов в текст документов.

id — уникальный идентификатор (ключ),

linked/text — идентификатор документа, содержащего данное вхождение,

id_obj — идентификатор объекта, участвующий в данном вхождении.

Вхождение объекта соответствует каждому упоминанию объекта в тексте, соответственно, в каждом тексте может быть несколько вхождений одного и того же объекта. Одно вхождение объекта может включать несколько текстовых фрагментов, которые соответствуют разным атрибутам объекта.

3.6. Таблица фрагментов. Таблица *Fragments* — таблица размеченных фрагментов. Фрагмент — это либо ссылка на ранее размеченный фрагмент, но уже с новыми свойствами, либо множество атомарных позиций, задающих текстовый интервал. Соответственно, выделяется непосредственный владелец фрагмента, т.е. признак или атрибут объекта, сопоставляемый с данным текстовым фрагментом, и владелец нулевого уровня, содержащий данный фрагмент не как ссылку, а как набор атомарных позиций (см. определение фрагмента). Ниже приведен список столбцов данной таблицы.

id — уникальный идентификатор (ключ),

type— тип непосредственного владельца данного фрагмента,

entry — вхождение (аннотация) объекта, в котором задействован данный фрагмент,

keeper — непосредственный владелец фрагмента,

rootLvlType — тип владельца нулевого уровня,

rootLvlKeeper — владелец нулевого уровня,

classifier — принимает значения *true* или *false* в зависимости от того, является ли данный фрагмент обозначением классификатора — типа сущности или непосредственным ее значением.

Параметры **type** и **rootLvlType** определяют таблицы, в которых задаются элементы лингвистической или предметной онтологии, а **keeper** и **rootLvlKeeper** — индексы строк в этих таблицах. Параметр entry является ссылкой на элемент таблицы *object_entry_log* и может быть не определен, если владельцем фрагмента не является объект. Данное поле определяет свойства объекта, упомянутые в конкретном вхождении, если эти свойства были аннотированы.

3.7. Таблица атомарных позиций. Таблица *Atomic_positions* содержит атомарные позиции или атомы, т.е. неразрывные текстовые фрагменты. Для атомарной позиции можно однозначно определить координаты начала и конца в тексте. Более сложные фрагменты состоят из множества атомарных позиций. Таблица включает четыре столбца.

id— уникальный идентификатор (ключ),

keeper — фрагмент, содержащий данную позицию,

begin — начало атома,

end — конец атома.

3.8. Таблица стилей. Таблица *Style* — это таблица стилей, используемых для визуализации разметки текста. Стиль сопоставляется либо элементам онтологии предметной области, либо лингвистическим понятиям.

id — уникальный идентификатор (ключ),

type — тип элемента, которому соответствует данный стиль,

id_carrier — ссылка на этот элемент,

bgcolor — цвет фона текста,

color — цвет текста,

fBold — полужирный текст,

fItalic — курсив,

fUL — подчеркнутый текст,

descr — описание стиля.

Как видно, значения кортежей таблицы задают различные стили форматирования текста в зависимости от типа размечаемого элемента. Параметр **type** определяет таблицу, в которой задается элемент онтологии, а **id_carrier** — индексы строк в этой таблице.

4. Архитектура системы. Для доступа к контенту предполагается разработка нескольких видов интерфейса: программного, пользовательского и интерфейса редактора. Программный интерфейс (API) позволит использовать содержимое системы другими программами и приложениями для решения задач информационного поиска, обработки текстов документов, построения лингвистических ресурсов и др. Пользовательский интерфейс обеспечит доступ к содержимому системы рядовым пользователям и предоставит удобные средства поиска фактов и их контекста в документах, обеспечит навигацию по системе признаков и в корпусе документов. Интерфейс редактора предоставит удобные средства добавления информации и разметки текстов, формирования корпуса, согласованного с общим содержанием системы, создания системы лингвистических признаков. На рис. 2 показана общая архитектура системы.

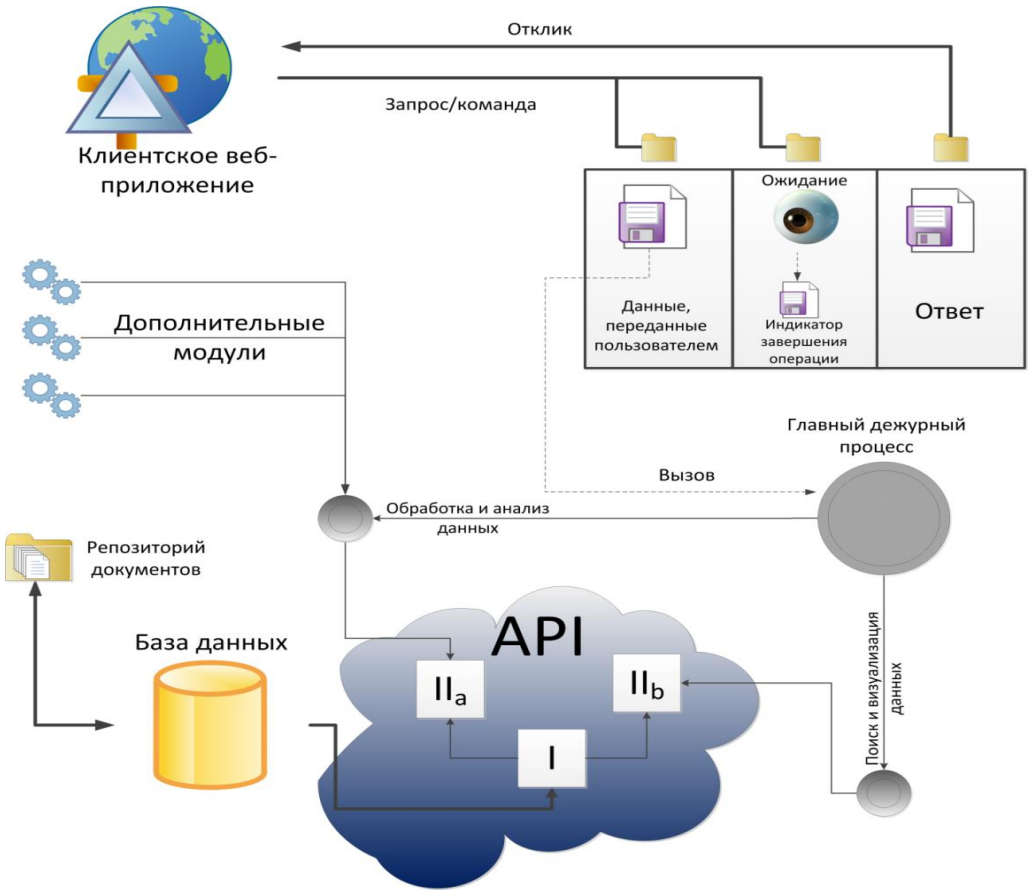


Рис. 2. Архитектура системы

Фактически работа с хранилищем данных в системе включает три контура (API). I — уровень ядра, инкапсулирует операции генерации SQL-запросов и обмена информацией с базой данных, реализует оптимизационные алгоритмы и механизмы обеспечения корректности и целостности данных. II_a — аналитический уровень, предоставляет инструменты для поиска данных и редактирования БД в терминах онтологии. API данного уровня оперирует понятиями онтологии, информационными объектами, документами и их аннотациями. II_b — уровень визуализации, снабжает пользовательский интерфейс информацией, необходимой для визуализации объектов аналитического уровня — информационных объектов и аннотаций. API данного уровня расширяет объекты уровня II_a набором методов для настройки визуализации. Как можно видеть на рис. 2, конечный пользователь системы работает с системой через веб-приложение. Для исполнения пользовательских запросов необходимо дополнительное звено. Таким звеном должен стать дежурный процесс, следящий за поступлениями новых запросов и обеспечивающий обмен информацией между веб-приложением и базой данных, не нарушая при этом принятой концепции. Для обмена информацией было решено использовать язык JSON (Java Script Object Notation). Единичную операцию поискового обращения к БД можно составить из следующих пунктов:

- пользователь средствами веб-интерфейса формулирует запрос и отправляет его на сервер;
- сервер, получив запрос, присваивает ему код, конвертирует его в файл формата JSON и дает сигнал об этом дежурному процессу;
- дежурный процесс извлекает из файла запроса всю необходимую информацию и вызывает соответствующие функции API базы данных;
- получив результат, дежурный процесс формирует файл-ответ в формате JSON, помечая его тем же кодом, которым был помечен запрос;
- веб-сервер после получения запроса и передачи его в дежурный процесс проверяет наличие ответа на него в течение заданного времени с заданной периодичностью; в случае отсутствия ответа по истечению срока пользователю отправляется сообщение вида not responded;
- если во время очередной проверки веб-сервер обнаруживает ответ на присланный запрос, он забирает его и отправляет пользователю.

Кроме того, в системе функционируют инструментальные модули, реализующие средства разметки текстов, проверки поступающего потока данных, контроля достоверности и пр.; эти средства, разрабатываемые по отдельности, интегрируются в единую среду.

5. Контроль целостности контента. Задача интеграции структурной и текстовой информации предполагает не только ее сбор с помощью ручной разметки текста экспертом или автоматического аннотирования документов какой-либо системой анализа, но и разработку методов и средств контроля поступающей информации, отслеживания ее достоверности и актуальности. Контроль достоверности данных предполагает слежение, в автоматическом или автоматизированном режиме, за поступающей информацией и ее оценку на основании упоминающих ее источников-документов. Соответственно, чем большее число источников упоминает некоторый факт, и чем выше авторитет этих источников, тем больше доверия вызывает данный факт. Даже если на момент поступления в систему информация имела статус заслуживающей доверия, со временем она может устареть, потерять актуальность и даже стать ложной. Косвенным признаком утери актуальности факта может послужить длительное отсутствие его упоминаний в новых документах. Накапливаясь в системе, подобные неактуальные данные занимают компьютерные ресурсы, вносят беспорядок и затрудняют поиск. Одним из видов средств, облегчающих работу с базой данных и обеспечивающих рациональное использование компьютерных ресурсов, являются разработанные оригинальные методы корректного пополнения базы данных фактами, полученными из текста. Основная идея подхода заключается в трехуровневой проверке информации, поступающей в базу данных. Рассмотрим каждый из этих уровней более подробно.

1. Установление референциальных связей между информационными объектами. Данный метод базируется на существующих методах разрешения анафоры в документах на русском и английском языках [4, 8, 11], а также на методах сравнения информационных объектов, предлагаемых в [3]. Он включает в себя установление степени сходства объектов, построение множества гипотетических эквивалентов для каждого объекта и объединение объектов, признанных кореферентными. Данный уровень подробно описан в [8]. Основной целью поиска референциально тождественных объектов является сокращение числа ИО, представляющих одну сущность, что, в свою очередь, повышает вероятность их успешной идентификации.
2. Идентификация информационных объектов — разрешения контекстной омонимии, являющейся одним из побочных эффектов обработки текста. Контекстная омонимия проявляется в наличии двух и более вариантов отождествления полученных из текста информационных объектов с объектами базы данных информационной системы. Ключевым для метода идентификации данных является понятие фокусного множества. Фокусное множество включает все экземпляры отношений, с помощью которых текущий

объект непосредственно связан с другими входными объектами. При этом множество отношений разбивается на подмножества связей с идентифицированными и требующими идентификации объектами. Основой метода является сопоставление фокусных множеств найденных в тексте объектов с фокусными множествами объектов, уже содержащихся в базе данных информационной системы.

3. Разрешение противоречий между содержащимися в базе данных и вновь поступившими фактами посредством вычисления специального параметра, количественно выражающего достоверность того или иного атрибута или связи. Фактически данный пункт заключает в себя слежение, как за достоверностью поступающих данных, так и за актуальностью уже имеющихся. Для контроля данных были разработаны подходы, включающие элементы теории вероятности. Были введены вероятностные характеристики документов и содержащихся в них фактов, а в основу модели жизненного цикла факта в информационной системе положен неоднородный марковский случайный процесс.

6. Заключение. Основной отличительной чертой данной работы является интеграция различных методов и подходов, существующих независимо, под одной исследовательской информационной оболочкой. В рамках данной системы знания извлекаются из текстов и визуализируются в виде базирующихся на онтологии структур (объектов, отношений). Процесс добавления новой информации может осуществляться пользователем либо через специализированный редактор, наследуемый от редактора данных из порталов знаний [5], либо при аннотировании текстов через подсистему семантической разметки корпуса текстов.

Технология разметки корпуса текстов в терминах объектов и связей предметной области была апробирована при создании корпуса текстов по катализу [7]. В рамках этого проекта были разработаны методы и средства семантического аннотирования двух типов: терминологическая разметка, которая предназначалась для фиксации в текстах имен понятий предметной области (терминологическая разметка была использована для создания предметного словаря по катализу), и разметка ситуаций (химических реакций), представляющих собой многоместные отношения, в которых размеченные сущности выступают в определенных семантических ролях.

Специфической особенностью исследовательской информационной системы является постоянное накопление документальных ресурсов и информации, корректность и достоверность которых необходимо постоянно отслеживать. С этой точки зрения потребуются дополнительные исследования ситуаций добавления разными экспертами противоречивой информации. Также в дальнейшем планируется подключать модули

автоматического анализа текста, извлечение информации в которых будет опираться на экспертные знания, выраженные с помощью средств аннотирования документов.

Список литературы

1. Апресян Ю. Д., Богуславский И. М. и др. Синтаксически и семантически аннотированный корпус русского языка: современное состояние и перспективы // Национальный корпус русского языка: 2003–2005. М.:Индрик, 2005. С.193–214.
2. Биряльцев Е.В., Елизаров А.М., Жильцов Н.Г., Иванов В.В., Невзорова О.А., Соловьев В.Д. Модель семантического поиска в коллекциях математических документов на основе онтологий // Труды 12-й Всероссийской научной конференции «Электронные библиотеки: перспективные методы и технологии, электронные коллекции» – RCDL'2010. Казань, 2010. С. 296–300.
3. Васильев И.А. Оценка семантической близости объектов с использованием дескриптивной логики // Материалы 5-й научно-практической конференции «Современные средства и системы автоматизации». Томск: ТУСУР, 2004. С. 160–163.
4. Ермаков А.Е. Референция обозначений персон и организаций в русскоязычных текстах СМИ: эмпирические закономерности для компьютерного анализа. // Компьютерная лингвистика и интеллектуальные технологии: Труды международной конференции «Диалог 2005». М.: Наука, 2005. С. 131–135.
5. Загорулько Ю.А., Боровикова, О.И. Подход к построению порталов научных знаний // Автометрия. 2008 № 1, Т. 44, С. 100–110.
6. Захаров В.П., Богданова С.Ю. Корпусная лингвистика // Учебник для студентов гуманитарных вузов. Иркутск: ИГЛУ, 2011. 161 с.
7. Кононенко И.С., Сидорова Е.А. Система семантической разметки корпуса текстов как инструмент извлечения экспертных знаний (на материале текстов по катализу) // Труды международной конференции «Корпусная лингвистика – 2011». СПб, 2011. С. 193–198.
8. Серый А.С., Сидорова Е.А. Поиск референциальных отношений между информационными объектами в процессе автоматического анализа документов // Труды XIV Всероссийской научной конференции RCDL-2012 Электронные библиотеки: перспективные методы и технологии, электронные коллекции. – Переславль-Залесский, 2012. С.206–212
9. Blanco X. Using Noo J for Multipurpose Analysis of Romance Languages Corpora // Труды межд. конф. «Корпусная лингвистика–2008». СПб., 2008. С.40–44.
10. Caroline V. Gasperin Statistical anaphora resolution in biomedical texts. // In Proc. of the 22nd International Conference on Computational Linguistics. Manchester. UK. 2008. P. 257–264.

11. Heeyoung Lee, Yves Peirsman, Angel Chang, Nathanael Chambers, Mihai Surdeanu, and Dan Jurafsky. 2011. Stanford's Multi-Pass Sieve Coreference Resolution System at the CoNLL-2011 Shared Task. // In Proc. of the 15th Conference on Computational Natural Language Learning: Shared Task. Portland. Oregon. USA. 2011. P.28–34.
12. Kim J.D., Ohta T., Tsujii J. Corpus annotation for mining biomedical events from literature // BMC Bioinformatics. 2008. 9:10.
13. O'Donnell M. The UAM CorpusTool: Software for corpus annotation and exploration // In Bretones Callejas, Carmen M. et al. (eds) Applied Linguistics Now: Understanding Language and Mind / La Lingüística Aplicada Hoy: Comprendiendo el Lenguaje y la Mente. Almería: Universidad de Almería. 2008. p. 1433–1447.

UDK: 004.89

Title: An approach to designing an information system with documented information.

Author(s):

Sidorova E.A. (A.P. Ershov Institute of Informatics Systems),

Seryj A.S. (A.P. Ershov Institute of Informatics Systems)

Abstract: The authors discuss an intelligent access to information resources. An analysis of information systems which support textual or ontology-based data representation is provided. The authors propose an approach to develop an information system that would support both of the aforementioned ways to represent knowledge. A possible architecture and database schema for such a system are described.

Keywords: information system, intelligent access to data, ontology, text annotation.