Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

И.С. Ануреев

СИСТЕМЫ ПЕРЕХОДОВ, ОРИЕНТИРОВАННЫЕ НА РАЗРАБОТКУ СРЕДСТВ СПЕЦИФИКАЦИИ И ВЕРИФИКАЦИИ ПРОГРАММНЫХ СИСТЕМ

Препринт
165

Новосибирск 2012

В работе предложен высокоуровневый язык спецификации помеченных систем переходов, ориентированный на разработку средств спецификации и верификации программных систем. Выделены классы таких систем, ориентированные на разработку операционной семантики языков программирования и доказательство свойств безопасности программ. Описана методология применения специализированных систем переходов к разработке операционной семантики и логики безопасности на примере модельного языка программирования.

Siberian Branch of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

I.S. Anureev

TRANSITION SYSTEMS ORIENTED TO DEVELOPMENT OF SOFTWARE SYSTEMS SPECIFICATION AND VERIFICATION TOOLS

Preprint
165

Novosibirsk 2012

A high-level specification language for labeled transition systems oriented to development of software systems specification and verification tools is proposed in this paper. The classes of such systems, focused on development of operational semantics of programmming languages and proving of safety properties of programs are presented. A methodology of application of the specialized transition systems for development of oprational semantics and safety logic by the example of a model programming language is described.

1. ВВЕДЕНИЕ

Современная тенденция в области верификации программ — переход от разработки методов верификации программ, применяемых для небольших программ на модельных языках программирования, к верификации больших программных систем на индустриальных языках программирования. Эта тенденция состоит в выделении практически значимых свойств программ и построении специализированных методов и техник анализа и верификации, ориентированных на эти свойства. Унификация и формализация процессов описания таких свойств и построения для них методов и техник является важной открытой проблемой. Для индустриальной верификации также характерно использование комбинации различных методов верификации. Это приводит к появлению новых гибридных методов верификации программ. Создание средств систематизации, анализа и формализации опыта, накопленного в области интеграции различных методов верификации — еще одна важная открытая проблема.

Для решения этих проблем мы предлагаем использовать ориентированные на программы системы переходов и представляем высокоуровневый язык описания (спецификации) таких систем. Эти системы задают общую базу для описания специализированных систем переходов, формализующих и унифицирующих те или иные аспекты работы с программами.

В этой статье рассмотрены три класса специализированных систем переходов.

Ориентированные на операционную семантику системы переходов используются для быстрой разработки формальных спецификаций языков программирования и прототипирования программных систем.

Ориентированные на логику безопасности системы переходов используются для формализации и унификации дедуктивных методов верификации программ.

Онтологические системы переходов используются для разработки операционно-онтологической семантики языков программирования и программных систем. Они позволяют описывать операционную семантику языков программирования и программных систем на базе их онтологии.

Работа выполнена при финансовой поддержке гранта РФФИ №11-01-00028-а и интеграционного проекта РАН №15/10 «Математические и методологические аспекты интеллектуальных информационных систем».

2. ПРЕДВАРИТЕЛЬНЫЕ ПОНЯТИЯ И ОБОЗНАЧЕНИЯ

Опишем нотацию основных структур данных, используемых в этой статье, как на уровне объектного описания, так и на уровне метаописания, — списков, последовательностей и функций.

Списки имеют вид (a$_1$ ... a$_n$), где элементы a$_i$ списка разделены пробелами. Пусть (lists of x) обозначает множество всех списков из элементов множества x, (lists of x of length n) — множество всех списков из элементов множества x длины n, (length of a) — число элементов в списке a.

Конечные последовательности элементов имеют вид a$_1$ ... a$_n$, где a$_i$ разделены пробелами. Пусть (sequences of x) обозначает множество всех конечных последовательностей элементов множества x, и (sequences of x of length n) — множество всех последовательностей элементов множества x длины n. Пустая последовательность (последовательность, не содержащая элементов) обозначается через empty-sequence.

Пусть bool обозначает множество {true, false} и nat — множество натуральных чисел с нулем. Пусть (union of A$_n$ where (n $\in$ x)) обозначает объединение всех множеств A$_n$ для всех (n $\in$ x).

Пусть (f x) обозначает применение функции f к последовательности аргументов x (вызов функции f на аргументах x).

Пусть undef обозначает тот факт, что некоторая функция не имеет значения для некоторого аргумента и ((domain of f) = {x | ((f x) $\neq$ undef))} — область определения функции f. Пусть f и g — функции такие, что (((domain of f) $\cap$ (domain of g)) = $\emptyset$). Тогда объединением (f $\cup$ g) функций f и g называется функция h такая, что ((domain of h) = (domain of f) $\cup$ (domain of g)), ((h x) = (f x)) для (x $\in$ (domain of f)) и ((h x) = (g x)) для (x $\in$ (domain of g)).

Для функций (как правило, конечных) мы будем также использовать альтернативную атрибутную нотацию. Пусть f — функция. В этом случае функция f называется атрибутной структурой, элементы области определения f — атрибутами, а объект (f a) (обозначаемый f.a) — значением атрибута a структуры f.

Определим операции доступа . и модификации upd на функциях следующим образом:

- (f.x = (f x));
- если (y $\neq$ x), то (((upd f x e) y) = (f y));
- (((upd f x e) x) = e).

Операции `.` и `upd` переносятся на последовательности и списки, если их рассматривать как функции, областью определения которых является целочисленный отрезок.

Будем говорить, что функция `f` может отличаться от функции `g` только на множестве `x` и обозначать это факт `(f can differ from g only on x)`, если `((f a) = (g a))` для любого `(a` $\notin$ `x)`.

Помеченная система переходов `lts` — это тройка `(states labels tr)`, где `states` и `labels` — множества, элементы которых называются состояниями и метками, соответственно, логическая функция `(tr` $\in$ `((states` $\times$ `labels` $\times$ `states)` $\to$ `bool))` называется отношением перехода. Говорят, что из состояния `s` можно перейти в состояние `ss` по метке `lab`, если `(tr s lab ss)`.

3. ОРИЕНТИРОВАННЫЕ НА ПРОГРАММЫ СИСТЕМЫ ПЕРЕХОДОВ

Ориентированные на программы системы переходов (Program Specific Transition Systems, P-STS) — это системы переходов, которые используются для формализации различных аспектов работы с программами (определение семантики, задание стратегий верификации программ и т.п.).

Состояниями в P-STS являются алгебраические системы специального вида.

Пусть `atoms` — некоторое множество объектов, называемых атомами.

Выражение — это либо список выражений, либо атом. Пусть `expressions` обозначает множество всех выражений.

В контексте последующего изложения будем называть элементы множеств `(sequences of expressions)` и `(lists of atoms)` программами и символами, соответственно, и использовать обозначения `programs` и `symbols` для этих множеств.

Пусть `elements` — множество объектов, называемых элементами, такое, что `(expressions` $\subseteq$ `elements)`.

Состояние `s` относительно `(atoms elements)` определяется как тотальная функция из `(symbols` $\to$ `((union of (elements`^{n} $\to$ `elements) where (n` $\in$ `nat))` $\cup$ `{undef}))`. Множество `elements` называется носителем `s`, а его элементы — элементами `s`.

Множество `{(f` $\in$ `symbols)` | `((s f)` $\neq$ `undef)}` называется сигнатурой `s` и обозначается `(symbols of s)`, а элементы этого множе-

ства — символами s. Функция (s f) называется интерпретацией символа f в состоянии s. В отличие от стандартной алгебраической системы, в которой символы сигнатуры — атомы, символы состояния — списки атомов. Использование списков атомов в качестве символов состояния позволяет приблизить описание вызовов функций, обозначаемых этими символами, к описанию на естественном языке.

Другой важной особенностью символов состояния является использование в них специальных атомов _ и __, которые делают такие символы шаблонами для вызовов функций, обозначаемых этими символами. Эти атомы, называемые спецификаторами аргументов, обозначают места для аргументов функций. Пусть (f ∈ (symbols of s)). Атом _, входящий в f, дополнительно указывает на то, что аргумент при вызове функции (s f) нужно сначала вычислить, а атом __, входящий в f, — что вычислять аргумент не нужно. Число вхождений спецификаторов аргументов в f называется местностью f и обозначается (arity of f). Для s должно выполняться свойство: если (n = (arity of f)), то ((s f) ∈ (elements^n → elements)).

Например, для операции равенства = соответствующий символ f будет иметь вид (_ = _) и ((s f) ∈ ((elements × elements) → bool)). В этом случае, (bool ⊆ elements).

Пусть (u, w ∈ (sequences of expressions)). Пусть typed-args — список структур с атрибутами arg и type со значениями из expressions и {value, itself}, называемых типизированными аргументами. Выражение e — пример f относительно typed-args тогда и только тогда, когда выполнено первое подходящее свойство:

- если (e = ()) и (f = ()), то (typed-args = ());
- если (e = (typed-args.1.arg u)) и (f = (_ w)), то (typed-args.1.type = value) и (u) — пример (w) относительно (typed-args.2 ... typed-args.n);
- если (e = (typed-args.1.arg u)) и (f = (__ w)), то (typed-args.1.type = itself) и (u) — пример (w) относительно (typed-args.2 ... typed-args.n);
- если (e = (b u)) и (f = (b w)), то (u) — пример (w) относительно typed-args;
- false.

Выражение e называется примером f, если e — пример f относительно некоторого typed-args.

Одно и то же выражение может быть примером для разных символов состояния, поэтому возможен конфликт при выборе символа состояния для этого выражения. Мы считаем, что определена функция (match ∈ ((expressions × states) → ((symbols × expressions) ∪ {undef}))), разрешающая этот конфликт, такая, что,

- если ((match e s) = (f typed-args)), то (f ∈ (symbols of s)) и e — пример f относительно typed-args;
- если ((match e s) = undef), то не существуют (f ∈ (symbols of s)) и typed-args таких, что e — пример f относительно typed-args.

Значение (value of e in s) выражения e в состоянии s определяется следующим образом (применяется первое подходящее правило):

- если e — атом, то ((value of e in s) = e);
- если e = ((ee)) и (ee ∈ (sequences of expressions)), то ((value of e in s) = ((value of (ee) in s)));
- если (e ∈ (symbols of s)), то ((value of e in s) = (s e));
- если ((match e) = (f typed-args)), то ((value of e in s) = ((s f) arg-values));
- ((value of e in s) = undef).

Список (arg-values ∈ (lists of elements of length n)), где (n = (arity of f)), в вышеприведенном определении задается следующим образом:

- если (typed-args.i.type = value), то (arg-values.i = (value of typed-args.i.arg in s));
- если (typed-args.i.type = itself), то (arg-values.i = typed-args.i.arg).

Функция (σ ∈ (atoms → (sequences of expressions))) называется подстановкой. Если (domain of σ) = {x_1, ..., x_n}, то σ может записываться как ((x_1 (σ x_1)) ... (x_n (σ x_n))). Функция подстановки subst относительно σ определяется следующим образом (применяется первое подходящее правило):

- если (e ∈ (domain of σ)), то ((subst e σ) = (σ e));
- если (e ∈ atoms), то ((subst e σ) = e);
- ((subst (e_1 ... e_n) σ) = ((subst e_1 σ) ... (subst e_n σ))).

Опишем теперь компоненты, из которых строится P-STS (p-sts = (states labels tr)).

Состояние `predefined-interpretation` определяет интерпретацию предопределенных символов. Интерпретация этих символов не меняется, когда `p-sts` переходит из состояния в состояние. Пусть `predefined-symbols` обозначает множество `(symbols of predefined-interpretation)`.

Множество `modifiable-symbols` включает модифицируемые символы. Их интерпретация может меняться, когда `p-sts` переходит из состояния в состояние.

Множества `predefined-symbols` и `modifiable-symbols` могут пересекаться. Состояние `s` системы `p-sts` расширяется на `predefined-symbols` таким образом, что `(((s f) a) = ((predefined-interpretation f) a))` для любого `(a` $\notin$ `(domain of (s f)))`.

Множество `modifiable-symbols` включает специальные символы `(value _)`, `(history length)` и `(_ is new-element)`.

Символ `(value _)` используется для хранения промежуточных значений, появляющихся при функционировании P-STS. Для каждого состояния `s` промежуточные значения суть значения выражений `(value 1)`, ..., `(value k)` в `s`, где `(k = (value of (history length) in s))`.

Пусть `((new elements of s) = {(a` $\in$ `elements) | (((s (_ is new-element)) a) = true)})`. Это множество характеризует элементы из `elements`, которые «не используются» в состоянии `s` и состояниях, которые предшествовали `s` относительно `tr`. Элементы этого множества будем называть новыми элементами в `s`. Для любого `(s` $\in$ `states)` символ `(_ is new-element)` обладает следующими свойствами:

- `(((s (is new-element)) a)` $\in$ `bool)` для любого `(a` $\in$ `elements)`. Это свойство означает тотальность символа `(_ is new-element)`;
- если `(tr s lab ss)`, то `((new elements of ss)` $\subseteq$ `(new elements of s))`. Это свойство означает монотонное невозрастание множества `(new elements of s)` относительно `tr`;
- если `(tr s lab ss)`, то `((new elements of s)` $\setminus$ `(new elements of ss))` конечно. Это свойство означает, что при каждом переходе «используется» только конечное число новых элементов;
- `(new elements of s)` бесконечно. Это свойство означает, что число новых элементов всегда «достаточно» для «использования» в любом числе переходов;
- `((value of e in s)` $\in$ `(new elements of s)` для любого `(e` $\in$ `expressions)` и `(((new elements of s)` $\cap$ `expressions) =` $\emptyset$`)`.

Эти свойства формализуют понятие «не используется».

Помеченная система переходов `p-sts = (states labels tr)` называется P-STS относительно `(atoms elements predefined-interpretation modifiable-symbols)`, если `lab = (p | pp)` для некоторых `(p, pp ∈ programs)`, и для любого `(s ∈ states)` выполнены следующие условия:

- `s` — состояние относительно `(atoms elements)`;
- `((symbols of s) = (predefined-symbols ∪ modifiable-symbols))`. Это свойство означает, что сигнатура любого состояния содержит в точности символы из `predefined-symbols` и `modifiable-symbols`;
- `((s (history length)) ∈ nat)`;
- если `(tr s lab ss)`, то `((s (history length)) ≤ (ss (history length)))` для любого `(ss ∈ states)`. Это свойство означает, что множество промежуточных значений может только пополняться;
- `((value of (value i) in s) = undef)` для любого `(i ≥ (s (history length)))`. Это свойство означает, что множество промежуточных значений ограничено константой `(history length)`.

Если `(tr s (p | pp) ss)`, то говорят, что `p` преобразует `s` в `ss` относительно `pp`. Таким образом, программы можно рассматривать как преобразователи состояний.

Список `(p s)` называется конфигурацией. Конфигурация должна удовлетворять следующему ограничению: если `p` содержит выражение вида `(value a)`, то `(a ∈ nat)`, и `(a ≤ (s (history length)))`.

Если `(tr s (p | pp) ss)`, то говорят, что из `(p s)` можно перейти в `(pp ss)`. Конфигурация `(p s)` называется заключительной, если не существует конфигурации `(pp ss)` такой, что `(tr s (p | pp) ss)`. Состояние `s` называется заключительным, если `(p s)` — заключительная конфигурация для любой программы `p`. Конфигурация называется точкой ветвления, если из нее можно перейти более чем в одну конфигурацию.

Трассой называется конечная или бесконечная последовательность конфигураций $(p_1\ s_1)\ (p_2\ s_2)\ \ldots$ такая, что из $(p_i\ s_i)$ можно перейти в $(p_{i+1}\ s_{i+1})$.

Специальный атом `backtrack` специфицирует фиктивную трассу исполнения программы. Конфигурация `(p s)` называется конфигураци-

ей отката, если (p.1 = (backtrack)). Отношение tr должно удовлетворять следующему свойству: если (p s) — конфигурация отката, то (p s) — заключительная конфигурация. Конечная трасса, последний элемент которой — конфигурация отката, называется фиктивной. Откат заключается в том, что при достижении конфигурации отката (p s) система p-sts возвращается (откатывается) в ближайшую из точек ветвления, из которой p-sts перешла в (p s), и выбирает другую трассу для выполнения. Если таких трасс нет, то p-sts откатывается в предыдущую точку ветвления. Если таких точек нет, то все трассы фиктивны (возможно, за исключением трассы, состоящей из единственного элемента — начальной конфигурации).

Функция (io-sem ∈ (programs → ((states × states) → bool))) называется семантикой входа-выхода относительно p-sts, если ((io-sem p) s ss) тогда и только тогда, когда существует конечная нефиктивная трасса (p_1 s_1) ... (p_n s_n) такая, что ((p_1 s_1) = (p s)), (s_n = ss) и (p_n s_n) — заключительная конфигурация.

Отношение перехода tr определяется как объединение отношений перехода, каждое из которых характеризуется видом выражения p.1. В свою очередь, эти выражения делятся на регулярные выражения и нерегулярные выражения. Нерегулярные выражения изменяют состояние p-sts специфичным образом, и для каждого вида таких выражений дается свое определение семантики. Регулярное выражение p.1 изменяет состояние некоторым унифицированным образом, задаваемым правилами перехода (или правилами операционной семантики) для p.1.

Правило перехода r имеет вид (if sam var x hvar w then c), где (sam ∈ expressions), (x ∈ (sequences of expressions)), (w ∈ symbols), и (c ∈ programs).

В случае ((length of x) = 0) или ((length of w) = 0) соответствующие компоненты var x и hvar w могут опускаться.

Выражение sam называется образцом правила r. Образец sam определяет множество выражений p.1, к которым применимо правило r.

Элементы x называются спецификаторами переменных образца. Если спецификатор u является атомом, то он специфицирует переменную образца u типа exp. Если u имеет вид (seq v), где (v ∈ atoms), то он специфицирует переменную образца v типа seq. В переменных образца сохраняются значения, полученные при сопоставлении с образцом выражения p.1. В переменных типа exp хранятся выражения, а в переменных типа seq — последовательности выражений. Пусть u.var обо-

значает переменную, которую специфицирует u, а u.type — тип u.var.

Элементы w называются переменными истории правила r. Они используются для сохранения промежуточных значений в символе (value _). Значение символа (history length) при переходе с помощью правила r увеличивается на (length of w).

Программа c называется телом правила r. Результат cc модификации тела c в соответствии со значениями переменных образца и переменных истории подставляется в p вместо p.1. В результате получается pp. Тело c не должно включать примеры символов (value _) и (history length).

Для разных видов P-SPS множества модифицируемых символов, виды нерегулярных выражений, их семантика и семантика выполнения правил перехода может отличаться.

4. ОРИЕНТИРОВАННЫЕ НА ОПЕРАЦИОННУЮ СЕМАНТИКУ СИСТЕМЫ ПЕРЕХОДОВ

Ориентированные на операционную семантику системы переходов (Operational Semantics Specific Transition Systems, OS-STS) — это специальный случай P-STS, используемый для описания операционной семантики программ.

Множество modifiable-symbols включает символ (value), который используется, чтобы моделировать возвращение значений при функционировании P-STS, и хранит последнее возвращенное значение. Говорят, что p возвращает значение v относительно pp, если (tr s (p | pp) ss) для некоторого ss и ((ss (value)) = v). Говорят, что p возвращает значение v, если p возвращает значение v относительно некоторой pp.

Нерегулярные выражения в OS-STS бывают пяти видов.

Нерегулярное выражение (stop) называется остановом и имеет следующую семантику: (tr s ((stop) p | pp) ss) тогда и только тогда, когда (ss = s) и pp — пустая последовательность.

Нерегулярное выражение (assume a) называется условием продолжения и имеет следующую семантику: (tr s ((assume a) p | pp) ss) тогда и только тогда, когда (ss = s) и выполнено первое подходящее свойство:

- если ((value of a in s) = true), то (pp = p);
- (pp = (backtrack) p).

Нерегулярное выражение (modify a) называется условием модификации и имеет следующую семантику: (tr s ((modify a) p | pp) ss)

тогда и только тогда, когда выполнено первое подходящее свойство:

- если `((value of a in s wrt ss) = true)`, то `(pp = p)` и `(ss can differ from s only on x)`, где `x` — множество модифицируемых символов, для которых существует пример `(e)` такой, что `(: e)` входит в `a`;
- `(ss = s)` и `(pp = (backtrack) p)`.

Значение `(value of e in s wrt ss)` выражения `e` в состоянии `s` относительно `ss` определяется следующим образом (применяется первое подходящее правило):

- если `(e ∈ atoms)`, то `((value of e in s wrt ss) = e)`;
- если `e = ((ee))` и `(ee ∈ (sequences of expressions))`, то `((value of e in s wrt ss) = ((value of (ee) in s wrt ss) ))`;
- если `(e = (: ee))` и `((ee) ∈ (symbols of s))`, то `((value of e in s wrt ss) = (ss (ee)))`;
- если `(e = (: ee))` и `((match (ee) ss) = (f typed-args))`, то `((value of e in s wrt ss) = ((ss f) arg-values))`;
- если `(e.1 ≠ :)` и `(e ∈ (symbols of s))`, то `((value of e in s wrt ss) = (s e))`;
- если `(e.1 ≠ :)` и `((match e s) = (f typed-args))`, то `((value of e in s wrt ss) = ((s f) arg-values))`;
- `((value of e in s wrt ss) = undef)`.

Список `(arg-values ∈ (lists of elements of length n))` в вышеприведенном определении задается так же, как в определении для `(value of e in s)`. Специальный атом `:` в выражении `(: ee)` означает, что символ, примером которого является `(ee)`, интерпретируется в модифицированном состоянии `ss`.

Нерегулярное выражение `(modify a else b)` называется условием модификации с альтернативой и имеет следующую семантику: `(tr s ((modify a else b) p | pp) ss)` тогда и только тогда, когда выполнено первое подходящее свойство:

- если `((value of a in s wrt ss) = true)`, то `(pp = p)` и `(ss can differ from s only on x)`, где `x` — множество модифицируемых символов, для которых существует пример `(e)` такой, что `(: e)` входит в `a`;
- `(ss = s)` и `(pp = b p)`.

Нерегулярное выражение `(a ::= b)` называется модификацией символа и имеет следующую семантику: `(tr s ((a ::= b) p | pp) ss)`

тогда и только тогда, когда выполнено первое подходящее свойство:

- если ((match a s) = (f typed-args)) и (f ∈ modifiable-symbols), то ss = s и (pp = (modify ((: f) = (updv f args b))) p)
- (ss = s) и (pp = (fail) p).

Предопределенный символ (updv _ _ _) имеет следующую интерпретацию: ((value of (updv g (x_1 ... x_n) y) in s) = (upd (value of g in s) ((value of x_1 in s) ... (value of x_n in s)) (value of y in s))).

Список args ∈ (lists of expressions of length (arity of f)) определяется следующим образом:

- если (typed-args.i.type = itself), то (args.i = (quote typed-args.i.arg));
- если (typed-args.i.type = value), то (args.i = typed-args.i.arg).

Предопределенный символ (quote __) имеет следующую интерпретацию: ((value of (quote a) in s) = a).

Определим семантику правила r системы op-sts.

Пусть σ — подстановка на множестве переменных образца правила r такая, что ((σ x.i.var) ∈ expressions), если (x.i.type = exp) и ((σ x.i.var) ∈ (sequences of expressions)), если (x.i.type = seq), δ — подстановка на переменных истории правила r такая, что (δ w.j) = (value ((value of (history length) in s) + j)) и (γ = (σ ∪ δ)).

Пусть (cc ∈ programs) определяется следующим образом: ((length of cc) = (length of c)) и (cc.k = (del* (subst c.k γ) s)) для всех (1 ≤ k ≤ (length of c)).

Функция del* вычисляет выражения, помеченные специальным атомом * в теле правила r, и определяется следующим образом (применяется первое подходящее правило):

- ((del* (* a) s) = (value of (del* a) in s));
- ((del* (a_1 ... a_n) s) = ((del* a_1 s) ... (del* a_n s)));
- ((del* a s) = a).

Пусть ((length of w) = m), и ((length of p) = n). Отношение перехода tr для регулярного выражения p.1 определяется правилами перехода следующим образом: (tr s (p | pp) ss) тогда и только тогда, когда существует подстановка σ такая, что

- ((subst a σ) = p.1);

- `(ss can differ from s only on {(history length)})` и `((ss (history length)) = ((ss (history length)) + m))`;
- `(pp = cc p.2 ... p.n)`.

В качестве примера применения правил OP-STS определим операционную семантику выражений, часто используемых в OP-STS.

Выражение `(new element)` называется генератором нового элемента и определяется правилом

```
(if (new element)
 then (modify (((: value) is new-element) and
  ((: _ is new-element) =
   (updv (_ is new-element) ((: value)) false)))))
```

Выражение `(fail)` обозначает некорректное завершение программы и называется небезопасным завершением. Конфигурация `(p s)` называется небезопасной, если `(p.1 = (fail))`. В противном случае `(p s)` называется безопасной. Отношение `tr` должно удовлетворять следующему свойству: если `(p s)` — небезопасная конфигурация, то `(p s)` — заключительная конфигурация. Конечная трасса, последний элемент которой — небезопасная конфигурация, называется небезопасной.

Выражение `(assert a)` называется условием безопасности и определяется правилами

```
(if (assert a) var a then (assume a))

(if (assert a) var a then (assume (not a)) (fail) (stop))
```

Выражение `(restart)` называется перезапуском и определяется правилом

```
(if (restart)
 then (modify (((: value) = undef) and
  (forall x ((: value x) = undef)) and
  ((: history length) = 0))))
```

5. ОРИЕНТИРОВАННЫЕ НА ЛОГИКУ БЕЗОПАСНОСТИ СИСТЕМЫ ПЕРЕХОДОВ

Пусть `os-sts` — OS-STS. Определим понятие безопасности программы относительно `os-sts`.

Программа `p` называется безопасной в системе `os-sts` относительно состояния `s`, если не существует конечной трассы $(p_1\ s_1)\ \ldots\ (p_n\ s_n)$

такой, что ((p$_1$ s$_1$) = (p s)) и (p$_n$ s$_n$) — небезопасная конфигурация (см. определение на стр. 16).

Программа p называется безопасной в os-sts относительно предусловия (pre ∈ expressions), если p безопасна в os-sts относительно s для любого состояния s такого, что ((value of pre in s) = true).

Ориентированные на логику безопасности системы переходов (Safety Logic Specific Transition Systems, SL-STS) являются специальным видом P-STS и используются для проверки безопасности программ. Пусть sl-sts — ориентированная на логику безопасности система переходов. Правила перехода sl-sts сводят проверку безопасности программы в os-sts относительно pre к проверке истинности набора выражений, называемых условиями корректности. Если все условия корректности истинны, то программа безопасна в os-sts относительно pre.

P-STS sl-sts называется ориентированной на логику безопасности системой переходов относительно os-sts, если выполнены свойства, сформулированные в следующем абзаце.

Множество modifiable-symbols системы sl-sts включает специальные символы (precondition) и (version of _). Символ (precondition) хранит предусловие. Множество (domain of (s (version of _))) одно и тоже для любого состояния s и совпадает с множеством modifiable-symbols системы os-sts. Обозначим это множество через (domain of version). Элемент ((value of (version of f) in s) ∈ nat) хранит число предыдущих состояний системы os-sts, в которых, возможно, менялась интерпретация символа f. Этот элемент называется версией символа f в состоянии s.

Поскольку символы (value) и (_ is new-element) принадлежит modifiable-symbols системы os-sts, множество modifiable-symbols системы sl-sts использует вместо них символы (value*) и (_ is new-element*).

Нерегулярные выражения в SL-STS бывают семи видов.

Нерегулярное выражение (stop) определяется так же, как для OS-STS.

Нерегулярное выражение (modify a) называется условием модификации и имеет следующую семантику: (tr s ((modify a) p | pp) ss) тогда и только тогда, когда (pp = p) и выполнены следующие свойства:

- (ss can differ from s only on {(precondition), (version of _)}),
- ((ss (precondition)) = ((s (precondition)) and (add

version to a wrt s))),

- если (f ∈ x), где x — множество модифицируемых символов, для которых существует пример (e) такой, что (: e) входит в a, то ((ss (version of f)) = ((s (version of f)) + 1)),
- если (f ∈ (modifiable symbols) \ x), то ((ss (version of f)) = (s (version of f))).

Специальный атом : в выражении (: e) означает, что символ, примером которого является (e), увеличивает свою версию на единицу при переходе в состояние ss.

Пусть символ (concatenate atoms _ and _) интерпретируется как конкатенация атомов, являющихся его аргументами. Пусть (f ∈ (domain of version)). Выражение (add version to a wrt s) расставляет в a версии у всех символов, входящих в (domain of version) и имеет следующее определение (применяется первое подходящее правило):

- если (a ∈ atoms), то ((add version to a wrt s) = a);
- если a = (: b)) и (b) — пример f, то ((add version to a wrt s) = ((concatenate atoms : and ((value of (version of f) in s) + 1)) (add version to b wrt s)));
- если a = (b) и a — пример f, то ((add version to a wrt s) = ((concatenate atoms : and (value of (version of f) in s)) (add version to b wrt s)));
- если a = (b), то ((add version to a wrt s) = ((add version to b wrt s)));
- ((add version to a_1 ... a_n s) = (add version to a_1 wrt s) ... (add version to a_n wrt s)).

Нерегулярное выражение (assume a) называется условием отката и имеет следующую семантику: (tr s ((assume a) p | pp) ss) тогда и только тогда, когда (pp = p), (ss can differ from s only on {(precondition)}) и ((ss (precondition)) = ((s (precondition)) and (add version to a wrt s))).

Нерегулярное выражение (a ::= b) называется модификацией символа и имеет следующую семантику: (tr s ((a ::= b) p | pp) ss) тогда и только тогда, когда выполнено первое подходящее свойство:

- если ((match a s) = (f typed-args)) и (f ∈ modifiable-symbols), то ss = s и (pp = (modify ((: f) = (updv f args b))) p)
- (ss = s) и (pp = (fail) p).

Семантика нерегулярных выражений `(assume* a)`, `(modify* a)` и `(a ::=* b)`, называемых операционным условием продолжения, операционным условием модификации и операционным условием модификации символа, соответственно, совпадает с семантикой выражений `(assume a)`, `(modify a)` и `(a ::= b)` в OS-STS.

Семантика правил в SL-STS совпадает с семантикой правил в OS-STS.

В качестве примера применения правил SL-STS определим операционную семантику выражений, часто используемых в SL-STS.

Выражение `(fail)` называется условием небезопасного завершения и определяется правилом

```
(if (fail)
 then (value* ::=* (quote (not (* precondition)))) (stop))
```

Выражение `(new element)` называется генератором нового элемента и определяется правилом

```
(if (new element)
 then (modify (((: value) is new-element) and
  ((: _ is new-element) =
   (updv (_ is new-element) ((: value)) false)))))
```

Выражение `(new element*)` называется операционным генератором нового элемента и определяется правилом

```
(if (new element*)
 then (modify* (((: value) is new-element*) and
  ((: _ is new-element*) =
   (updv (_ is new-element*) ((: value)) false)))))
```

Выражение `(assert a)` называется условием безопасности и определяется правилами

```
(if (assert a) var a then (assume a))

(if (assert a) var a then (assume (not a)) (fail) (stop))
```

Выражение `(assert* a)` называется операционным условием безопасности и определяется правилами

```
(if (assert* a) var a then (assume* a))

(if (assert* a) var a then (assume* (not a)) (fail) (stop))
```

Выражение `(restart with precondition a)` называется перезапуском и определяется правилом

```
(if (restart with precondition a) var a
 then (modify* (((: value*) = undef) and
  (forall x ((: value x) = undef)) and
  ((: history length) = 0) and
  (forall x ((: version of x) = 0)) and
  ((: precondition) = (add null version to a)))))
```

Выражение `(add null version to a)` расставляет в `a` нулевые версии у всех символов, входящих в `(domain of version)`, и имеет следующее определение (применяется первое подходящее правило):

- если `(a` $\in$ `atoms)`, то `((add null version to a) = a)`;
- если `a = (b)`, и `a` — пример `(f` $\in$ `(domain of version))`, то `((add null version to a) = (:0 (add null version to b)))`;
- если `a = (b)`, то `((add null version to a) = ((add null version to b)))`;
- `((add null version to` a_1 `...` a_n `s) = (add null version to` a_1 `wrt s) ... (add null version to` a_n `wrt s))`.

Множество условий корректности, порождаемых `sl-sts` для программы `p` относительно предусловия `pre`, определяется как множество значений символа `(value*)` во всех состояниях `s` таких, что `((io-sem (reset with precondition pre) p (stop)) undef-state s)`, где `undef-state` — состояние такое, что `((undef-state f) = undef)` для любого `(f` $\in$ `(symbols of undef-state))`.

6. ОПРЕДЕЛЕНИЕ ОНТОЛОГИЧЕСКИХ СИСТЕМ ПЕРЕХОДОВ В ТЕРМИНАХ ОРИЕНТИРОВАННЫХ НА ОПЕРАЦИОННУЮ СЕМАНТИКУ СИСТЕМ ПЕРЕХОДОВ

Операционно-онтологический подход к формальной спецификации языков программирования, основанный на спецификации концептуального окружения с помощью онтологии, был предложен авторами в [2]. В рамках подхода был введен новый вид семантики языков программирования — операционно-онтологическая семантика. В отличие от обычной операционной семантики, которая не накладывает никаких ограничений на состояния, состояния в операционно-онтологической семантике определяются как онтологические модели (множества экземпляров понятий онтологии) языка программирования.

Также был предложен формализм задания операционно-онтологической семантики — онтологические системы переходов (OTS) [4, 13,

21, 22] — специальный класс систем переходов, в котором на состояния накладывается онтологическая структура.

В этом разделе мы определим OTS и операционно-онтологическую семантику с помощью OS-STS. Это позволит объединить преимущества онтологического подхода с выразительностью формализма OS-STS.

Онтологическая система переходов `ots` определяется как ориентированная на операционную семантику система переходов, в которой во множестве `modifiable-symbols` выделяется два подмножества: `ontological-symbols` и `instantiation-symbols`.

Элементы `ontological-symbols` называются онтологическими символами. Они специфицируют элементы (понятия, атрибуты, отношения и т. п.) онтологии языка программирования.

Рассмотрим пример множества `ontological-symbols`, подставляя буквы `a`, `b`, `c` вместо спецификаторов аргументов, чтобы определить неформальный смысл символов, входящих в него:

- `(a is concept)` означает, что `a` — понятие;
- `(a is attribute of b)` означает, что `a` — атрибут понятия `b`;
- `(a is attribute of b of type c)` означает, что `a` — атрибут понятия `b` типа `c`. Типом является некоторое понятие. Например, `(a is attribute of b of integer)` означает, что `a` — целочисленный атрибут понятия `b`.

Элементы `instantiation-symbols` называются символами экземпляризации. Они специфицируют связи элементов онтологии с экземплярами.

Рассмотрим пример множества `instantiation-symbols`:

- `(a is b)` означает, что `a` — экземпляр понятия `b`;
- `(a of b)` означает значение атрибута `a` экземпляра `b` некоторого понятия. Например, если `b` — оператор присваивания, имеющий вид `(u := v)`, то выражение `(left-side of b)` означает левую часть `u` этого оператора;
- `(a of b of c)` означает значение атрибута `a` экземпляра `b` понятия `c`. Этот символ используется, чтобы разрешить конфликт в случае, когда `b` является экземпляром нескольких понятий, имеющих атрибут `a`.

Теперь мы можем формально определить операционно-онтологическую семантику языка программирования. Пусть `L` — язык программирования. Операционно-онтологическая семантика языка `L` определяется

как пара (p ots), где p ∈ programs, ots — онтологическая система переходов.

Программа p называется спецификацией онтологии языка L. Ее выполнение строит онтологию языка L. OTS ots определяет семантику выполнимых понятий языка L. Под выполнимым понятием понимается понятие, описывающее некоторый класс выполнимых сущностей языка L.

Заметим, что онтология современного языка программирования включает не только описание его инструкций и составляющих их элементов, но также описание таких фундаментальных концепций, как просачивание исключений, разрешение перегрузки операций, взаимодействие приложения с операционным окружением, нахождение динамического типа объекта в объектно-ориентированных языках программирования и т. д.

OTS ots должна сохранять онтологию языка L. Онтологическая система переходов сохраняет онтологию, если при переходе из состояния в состояние интерпретация онтологических символов не меняется.

7. ПРИМЕР

Рассмотрим применение ориентированных на программы систем переходов на примере игрушечного языка L.

Пусть s — текущее состояние. Язык L допускает следующие виды выражений:

- (block p) последовательно вычисляет выражения из (p ∈ programs);
- x возвращает значение переменной (x ∈ atoms). Это выражение называется выражением доступа к переменной. В случае, если переменной с именем x нет, программа порождает (fail). Для простоты будем считать, что все переменные целочисленные. Целые числа определяются предопределенным символом (_ is integer);
- (x := e) присваивает переменной x значение выражения e. В случае если (x := e) — первое присваивание атому x, это присваивание выполняет роль декларации переменной x с инициализатором e. Для простоты будем считать, что e — либо выражение доступа к переменной, либо целое число;
- c возвращает c, если ((value of (c is integer) in s) = true).

- `(if x then y else z)` — условный оператор с условием (`x` ∈ `expressions`), `then`-ветвью (`y` ∈ `programs`) и `else`-ветвью (`z` ∈ `programs`);
- `(while x do y)` — цикл с условием (`x` ∈ `expressions`) и телом (`y` ∈ `programs`);
- `(random)` возвращает произвольное целое число.

В первую очередь рассмотрим применение OS-STS для разработки операционной семантики языка L. Опишем шаги этой разработки.

На первом шаге определяется множество `modifiable-symbols` для языка L. Оно включает символы (`_ is variable`) и (`value of _`) в дополнение к обязательным символам. Символ (`_ is variable`) определяет множество переменных программы на языке L. Символ (`value of _`) хранит значения переменных L-программы.

На втором шаге определяются правила перехода, задающие семантику L-выражений:

```
(if (block a) var (seq a) then a)

(if a var a
 then (assert (a is variable)) ((value) ::= (value of a)))

(if a var a then (assume (a is integer)) ((value) ::= a))

(if (a := b) var a b
 then b ((a is variable) ::= true) ((value of a) ::= (value)))

(if (if a then b else c) var a (seq b) (seq c)
 then a (assume ((value) = true)) b)

(if (if a then b else c) var a (seq b) (seq c)
 then a (assume ((value) = false)) c)

(if (while a do b) var a (seq b)
 then (assume a) b (while a do b))

(if (while a do b) var a (seq b) then (assume (not a)))

(if (random) then (modify ((: value) is integer)))
```

На третьем шаге определяется множество `predefined-symbols`. Оно

состоит из символов, которые вводятся при описании операционной семантики выражений языка программирования. Описание этих символов и их интерпретации завершает определение состояния программы на этом языке. В нашем случае `predefined-symbols` включает символы `(_ = _)`, `(not _)` и `(_ is integer)` с обычной интерпретацией (равенство, отрицание и «быть целым числом»).

Заметим, что, если язык программирования имеет синтаксис, отличный от синтаксиса выражений, то требуется предварительный шаг. Этот шаг заключается в определении отношения эквивалентности между конструкциями этого языка и выражениями и трансляции его конструкций в эквивалентные им выражения. Поскольку такую трансляцию можно рассматривать как вид денотационной семантики, формальная семантика такого языка программирования определяется как комбинированная денотационно-операционная семантика.

Рассмотрим теперь применение SL-STS для построения логики безопасности для языка `L`. SL-STS определена так, что ее правила для многих конструкций языков программирования синтаксически совпадают с правилами соответствующей OP-STS. Так, для языка `L` правила для всех конструкций, за исключением правила для цикла, совпадают по синтаксису с правилами операционной семантики языка `L`.

Цикл `(while a do b)` заменяется аннотированным циклом `(while a invariant i do b)` с инвариантом `i`. Логика безопасности для цикла определяется следующим образом:

```
(if (while a invariant i do b) var a i (seq b)
 then (assert i) (stop))

(if (while a invariant i do b) var a i (seq b)
 then (precondition ::=* i) (assume a) b (assert i) (stop))

(if (while a invariant i do b) var a i (seq b)
 then (precondition ::=* i) (assume (not a)))
```

В заключение рассмотрим применение онтологических систем переходов для разработки операционно-онтологической семантики.

Рассмотрим сначала программу `p`, которая строит онтологию языка `L`:

```
((block is concept) ::= true)
((sequence is attribute of block) ::= true)
```

```
((identifier is concept) ::= true)

((variable is concept) ::= true)
((value is attribute of variable)) ::= true)

((constant is concept) ::= true)

((assignment is concept) ::= true)
((left-side is attribute of assignment)) ::= true)
((right-side is attribute of assignment)) ::= true)

((if-statement is concept) ::= true)
((condition is attribute of if-statement)) ::= true)
((then is attribute of if-statement)) ::= true)
((else is attribute of if-statement)) ::= true)

((while-statement is concept) ::= true)
((condition is attribute of while-statement)) ::= true)
((body is attribute of while-statement)) ::= true)

((random is concept) ::= true)
```

Теперь можно определить онтологическую систему переходов, которая специфицирует операционно-онтологическую семантику с помощью следующего набора правил:

```
(if a var a then (assume (a is block)) (* sequence of a))

(if a var a then (assume (a is identifier))
  (assert (a is variable)) ((value) ::= (value of a)))

(if a var a then (assume (a is integer)) ((value) ::= a))

(if a var a then (assume (a is assignment))
 (* right-side of a) ((a is integer) ::= true)
 ((value of (* left-side of a)) ::= (value)))

(if a var a then (assume (a is if-statement))
  (* condition of a) (assume ((value) = true)) (then of a))
```

```
(if a var a then (assume (a is if-statement))
  (* condition of a) (assume ((value) = false)) (else of a))

(if a var a then (assume (a is while-statement))
  (assume (* condition of a)) (* body of a) a)

(if a var a then (assume (a is while-statement))
 (assume (not (* condition of a))))

(if a then (assume (a is random))
 (modify ((: (value)) is integer)))
```

8. ЗАКЛЮЧЕНИЕ

В работе представлен язык для описания (спецификации) специализированных систем переходов, и рассмотрены три класса таких систем: ориентированные на операционную семантику системы переходов, ориентированные на логику безопасности системы переходов и онтологические системы переходов.

Мы планируем унифицировать разработанные ранее различные виды операционных [8, 27] и аксиоматических [5, 6, 9, 11, 12, 15, 25] семантик на базе ориентированных на операционную семантику систем переходов и ориентированных на логику безопасности систем переходов, соответственно, и интегрировать их в мультиязыковую систему анализа и верификации программ Спектр [7, 18]. Ориентированные на программы системы будут использованы в новом определении предметно-ориентированного языка Atoment [3, 19], на котором базируется система Спектр, и в портале знаний по компьютерным языкам [24] для формальной спецификации компьютерных языков.

Мы планируем также разработать новые виды специализированных систем переходов для спецификации методов упрощения условий корректности [1, 14, 17, 20, 23], контекстных машин [16] и разработанных ранее алгоритмов трансляции программ в рамках многоуровневых методов верификации C-light [10, 15] и C# [11, 26] программ.

СПИСОК ЛИТЕРАТУРЫ

1. Ануреев И.С. Метод элиминации структур данных, основанный на системах переписывания формул // Программирование. 1999. №4. С. 5–15.

2. Анореев И.С. Операционно-онтологический подход к формальной спецификации языков программирования // Программирование. 2009. №1. С. 1–11.
3. Анореев И.С. Типовые примеры использования языка Atoment // Моделирование и анализ информационных систем. 2011. Т. 18, №4. С. 7–20.
4. Анореев И.С. Язык описания онтологических систем переходов OTSL как средство формальной спецификации программных систем // Вестник НГУ, серия Информационные технологии. 2008. Т. 6, №3. С. 24–34.
5. Анореев И.С., Марьясов И.В., Непомнящий В.А. Верификация С-программ на основе смешанной аксиоматической семантики // Моделирование и анализ информационных систем. 2010. Т. 17, №3. С. 5–28.
6. Атучин М.М., Анореев И.С. Атрибутные аннотации и их применение в дедуктивной верификации С-программ // Моделирование и анализ информационных систем. 2011. Т. 18, №4. С. 21–33.
7. Непомнящий В.А., Анореев 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.
8. Непомнящий В.А., Анореев И.С., Михайлов И.Н., Промский А.В. На пути к верификации С программ. Язык C-light и его формальная семантика // Программирование. 2002. №6. С. 1–13.
9. Непомнящий В.А., Анореев И.С., Промский А.В. На пути к верификации С программ. Аксиоматическая семантика языка C-kernel // Программирование. 2003. №6. С. 5–15.
10. Непомнящий В.А., Анореев И.С., Промский А.В. На пути к верификации С-программ. Язык C-light и его трансформационная семантика // Проблемы программирования. 2006. №2–3. С. 359–368.
11. Непомнящий В.А., Анореев И.С., Промский А.В., Дубрановский И.В. На пути к верификации C# программ: трехуровневый подход // Программирование. 2006. №4. С. 4–20.
12. Шилов Н.В., Анореев И.С., Бодин Е.В. О генерации условий корректности для императивных программ // Программирование. 2008. №6. С. 1–20.
13. Anureev I.S. A language of actions in ontological transition systems // Joint NCC&IIS Bulletin, Series Computer Science. 2007. Vol. 26. P. 19–38.
14. Anureev I.S. A method for simplification procedures design based on formula rewriting system // Joint NCC&IIS Bulletin, Series Computer Science. 1998. Vol. 8. P. 1–18.
15. Anureev I.S. A three-stage method of C program verification // Joint NCC&IIS Bulletin, Series Computer Science. 2008. Vol. 28. P. 1–29.
16. Anureev I.S. Context machines as formalism for specification of dynamic systems // Joint NCC&IIS Bulletin, Series Computer Science. 2009. Vol. 29. P. 1–16.
17. Anureev I.S. Formula rewriting systems and their application to automated program verification // Joint NCC&IIS Bulletin, Series Computer Science. 1999. Vol. 10. P. 1–5.
18. 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.
19. Anureev I.S. Introduction to the Atoment language // Joint NCC&IIS Bulletin, Series Computer Science. 2010. Vol. 31. P. 1–16.

20. Anureev I.S. Multi-branch narrowing: satisfiability and termination // Joint NCC&IIS Bulletin, Series Computer Science. 2000. Vol. 13. P. 1–11.
21. Anureev I.S. Ontological models in OTSL // Problems in Programming. 2008. №2-3. P. 41–49.
22. Anureev I.S. Ontological transition systems // Joint NCC&IIS Bulletin, Series Computer Science. 2007. Vol. 26. P. 1–18.
23. Anureev I.S. Program verification based on specification language Simple // Joint NCC&IIS Bulletin, Series Computer Science. 2001. Vol. 15. P. 1–18.
24. Anureev I.S. Bodin E.V., Gorodnyaya L.V., Marchuk A.G., Murzin F.A., Shilov N.V. On the problem of computer language classification // Joint NCC&IIS Bulletin, Series Computer Science. 2008. Vol. 28. P. 31–42.
25. Anureev I.S. Bodin E.V., Shilov N.V. Effective generation of verification conditions for non-deterministic unstructured programs // Joint NCC&IIS Bulletin, Series Computer Science. 2007. Vol. 26. P. 39–64.
26. Nepomniashy V.A., Anureev I.S., Dubranovsky I.V., Promsky A.V. A three-level approach to C# program verification // Joint NCC&IIS Bulletin, Series Computer Science. 2004. Vol. 20. P. 61–85.
27. Nepomniaschy V.A., Anureev I.S., Promsky A.V. Verification-oriented language C-light and its structural operational semantics // PSI-2003. Proc. of Conf. Lect. Notes Comput. Sci. 2003. Vol. 2890. P. 1–5.

И.С. Ануреев

СИСТЕМЫ ПЕРЕХОДОВ, ОРИЕНТИРОВАННЫЕ НА РАЗРАБОТКУ СРЕДСТВ СПЕЦИФИКАЦИИ И ВЕРИФИКАЦИИ ПРОГРАММНЫХ СИСТЕМ

Препринт
165

Рукопись поступила в редакцию 6.11.2012
Рецензент Н.В. Шилов
Редактор Т.М. Бульонкова

Подписано в печать 6.12.2012
Формат бумаги 60×84 1/16 Объем 1,65 уч.-изд.л., 1,8 п.л.
Тираж 60 экз.

Центр оперативной печати "Оригинал 2"
г.Бердск, ул. Островского, 55, оф. 02, тел. (383) 214 45 35