Модел. и анализ информ. систем. Т. **18**, № **4** (2011) 7–20

УДК 519.681

Типовые примеры использования языка Atoment

Ануреев И.С.[1]

Институт систем информатики имени А.П. Ершова СО РАН

e-mail: anureev@iis.nsk.su

получена 9 октября 2011 года

Ключевые слова: верификация, спецификация, операционная семантика, аксиоматическая семантика, трансформационная семантика, предметно-ориентированные языки, системы верификации

Язык Atoment — предметно-ориентированный язык выполнимых спецификаций, применяемый для описания методов и техник верификации программ. В этой работе представлена коллекция типовых примеров использования языка Atoment, охватывающая такие темы, как модели программ, операционная, трансформационная и аксиоматическая семантики, формальная спецификация языков программирования.

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 накопления, анализа и формализации опыта, накопленного в области интеграции различных методов верификации, — еще одна важная открытая проблема.

Для решения этих проблем мы предлагаем использовать предметно-ориентированный язык разработки методов и средств верификации программ. Предметная

[1]Работа частично поддержана грантом РФФИ 11-01-00028-а и интеграционной программой РАН 14/12.

ориентированность языка позволит разработчику работать в естественной понятийной среде и, за счет этого, ускорить разработку специализированных методов и техник анализа и верификации для новых свойств, а использование единого языка позволит унифицировать и формализовать процессы описания таких свойств и методов их доказательства. Унифицированный язык также позволит создавать новые гибридные методы верификации, накапливать, анализировать и формализовать опыт в области верификации. Насколько нам известно, разработка такого языка выполняется впервые.

Предлагаемый нами язык Atoment [1] учитывает специфику проблемной области, а именно: представление данных (программ, аннотаций, аксиом, свойств, правил вывода и т. п.) в виде деревьев; применение методов верификации не к исходным текстам аннотированных программ, а к программным моделям, которые представляют собой помеченные направленные упорядоченные графы; естественное представление многих практических методов и техник анализа и верификации (методов статического анализа; методов, базирующихся на трансформационной, операционной и аксиоматической семантиках; методов проверки на моделях; автоматных методов; бисимуляционных техник) как преобразований (трансформаций) на этих графах; сложную концептуальную структуру программных систем и языков программирования, включающую сотни понятий.

Язык Atoment также обеспечивает выполнение ряда методологических принципов, которым, на наш взгляд, нужно следовать при разработке методов верификации. Во-первых, переход от текстов программ (аннотаций, аксиом, правил вывода и т. п.) к их моделям должен удовлетворять принципу структурной идентичности. Это означает, что каждой лексической и синтаксической единице текста должна соответствовать в точности одна единица модели. Выполнение этого принципа позволяет отождествлять текст и его модель, не доказывая корректность трансляции текстов в их модели. Во-вторых, трансляция должна удовлетворять принципу естественности. Это означает, что программная модель должна сохранять общепринятую терминологию и обозначения. Выполнение этого принципа как раз создает комфортную понятийную среду разработки методов и техник верификации. В-третьих, язык должен иметь компактный синтаксис и прозрачную семантику. Это уменьшает долю времени, которое будет потрачено при разработке методов верификации на освоение самого языка.

В этой статье описываются типовые примеры применения языка Atoment в задачах спецификации и верификации. На этих примерах демонстрируется выполнение для этого языка сформулированных выше свойств и принципов. Рассмотрены задачи описания программных моделей, разработки операционной, трансформационной и аксиоматической семантик.

2. Введение в язык Atoment

В этом разделе описываются основные понятия языка Atoment в объеме, необходимом для понимания примеров применения языка, представленных в следующих разделах.

Язык Atoment имеет компактный унифицированный синтаксис. Все единицы языка (как синтаксические, так и семантические) представляются выражениями, которые строятся из кирпичиков двух видов — атомов и элементов (в совокупности называемых атоментами) с помощью единственного конструктора выражений `(...)`. Атомы суть синтаксические кирпичики, которые имеют синтаксическое представление. Элементы суть семантические кирпичики, не имеющие синтаксического представления. Формально элементы и атомы — это два непересекающихся множества `Elem` и `At`, а выражения — это множество `Exp`, которое строится следующим образом:

- если `A` — атомент, то `A` — выражение;
- `()` — (пустое) выражение;
- если $\mathtt{A}_1$, ..., $\mathtt{A}_n$ — слоты, то `(`$\mathtt{A}_1$ `...` $\mathtt{A}_n$`)` — (составное) выражение.

Слоты суть "места" для размещения подвыражений. Они определяют семантику подвыражений в контексте выражения. Пусть `B` — выражение. Выделяют 4 вида слотов в зависимости от их роли в выражении: слот-значение и атрибутный слот, имеющие один и тот же вид `B`, типовой слот `B@` и слот-свойство `B@@`. Слот-значение выражения `A` называется атрибутным слотом, если он находится в нечетной позиции выражения, которое получается из `A` удалением всех типовых слотов и слотов-свойств. Смысл этих ролей объясняется ниже.

Выражения, состоящие только из слотов-значений, называются списками. Они аналогичны спискам языка Лисп.

Выражения могут интерпретироваться как данные и как вычислимые сущности. Первая интерпретация используется для представления данных (программ, аннотаций, аксиом, свойств, правил вывода и т. п.) для методов верификации, а вторая — для представления самих методов верификации. Прежде чем описывать семантику (вычисления) выражений, введем понятия состояния и переменной, которые определяют контекст вычисления.

Состояние `s` есть частичная функция из элементов в выражения. Пусть `St` обозначает множество всех состояний. Элемент `E` определен в состоянии `s`, если `s(E)` определено. Выражение `s(E)` называется значением (или содержимым) элемента `E`. Если выражение `s(E)` содержит атрибутный слот `B`, то выражение `B` называется атрибутом `E`. Если за атрибутным слотом `B` следует слот-значение `V`, то `V` называется значением атрибута `B` элемента `E` в состоянии `s`. Заметим, что так как атрибутный слот `B` может входить в выражение `s(E)` более чем один раз, атрибут `B` может иметь несколько значений. Если выражение `s(E)` содержит типовой слот `B@`, то выражение `B` называется типом элемента `E` в состоянии `s`. Аналогично атрибутам элемент может иметь более одного типа, поскольку типовых слотов в выражении `s(E)` также может быть несколько. Если выражение `s(E)` содержит слот `B@@`, то выражение `B` называется свойством элемента `E` в состоянии `s`. Слотом элемента `E` в состоянии `s` называется слот, входящий в выражение `s(E)`.

Состояние имеет естественную графовую интерпретацию и представляет собой помеченный направленный упорядоченный граф. Каждый элемент представляет

вершину графа, а слоты выражения, которое является значением этого элемента в текущем состоянии, — помеченные упорядоченные дуги. Элемент является началом дуг, а подвыражения, входящие в слоты, — концами дуг. Метки дуг суть типы слотов. Порядок дуг задается порядком слотов в выражении. Таким образом, состояния позволяют описывать программные модели, а выражения — методы преобразования этих моделей.

Любой элемент `E` может рассматриваться как понятие. Содержимым понятия `E` является множество элементов, для которых `E@` является слотом. Эти элементы называются экземплярами понятия `E`. Например, элемент `E` со значением `(goto-statement@ label L)` — экземпляр понятия `goto-statement`, который имеет атрибут `label` со значением `L`, т. е. `E` — оператор goto с меткой `L`. Понятия являются средством категоризации элементов. Свойства также можно рассматривать как средство категоризации по признаку (свойству), который не выделен в отдельное понятие.

Понятия и атрибуты используются для представления понятийной (онтологической) структуры программных систем и языков программирования. Для программных систем они описывают онтологию предметной области, а для языка программирования — его категориальный аппарат.

Означивание переменных `vv` есть частичная функция из атоментов в выражения. Пусть `VV` обозначает множество всех означиваний переменных. Если `vv(A)` определено, то `A` называется переменной, а `vv(A)` — значением переменной `A`. Переменные используются, чтобы хранить промежуточные результаты вычислений выражений. Распространим функцию `vv` на выражения следующим образом: `vv(U)` — результат замены в выражении `U` всех вхождений переменных, определяемых означиванием `vv`, на их значения; `vv(U, V)` — результат замены в выражении `U` всех вхождений переменных, определяемых означиванием `vv` и не принадлежащих множеству атоментов `V`, на их значения.

Вычисление выражения `E` возвращает в качестве результата некоторое выражение, которое называется значением выражения `E`. Также вычисление выражения может менять состояние. Результат вычисления выражения зависит от текущего состояния и текущего означивания переменных. Формально семантика выражения `Sem` есть функция из троек `(E, s, vv)`, где `E` — вычисляемое выражение, `s` — текущее состояние, `vv` — текущее означивание переменных, в множество троек `(V, s′, vv′)`, где `V` — значение выражения `E` в состоянии `s` при означивании переменных `vv`, `s′` и `vv′` — состояние и означивание переменных, полученные в результате вычисления выражения `E`. Семантика простых выражений определяется правилами:

- если `A` — атомент и `vv(A)` неопределено или `A` — пустое выражение, то `Sem(A, s, vv) = {(A, s, vv)}`;
- если `A` — атомент и `vv(A)` определено, то `Sem(A, s, vv) = {(vv(A), s, vv)}`.

Семантика составных выражений задается семантическими схемами. Семантическая схема — это элемент со значением вида `(sem A var C where B := D)`. Атрибут `sem` определяет вид выражений, удовлетворяющих схеме. Выражение `A`, являющееся значением этого атрибута, называется образцом схемы. Атрибут `var` определяет переменные образца `A`, а атрибут `where` — ограничения на значения этих

переменных. Переменные представляются атоментами, а их значения — выражениями. Выражение C, являющееся значением атрибута var, является либо атоментом (представляет одну переменную), либо списком атоментов (представляет набор переменных). Конкретные выражения, удовлетворяющие схеме, получаются заменой всех вхождений переменных из C в образце A на значения, удовлетворяющие условию B. Про такие выражения говорят, что они сопоставляются с образцом. В случае успешного сопоставления с образцом, значения переменных образца подставляются в выражение D и результирующее выражение D′ вычисляется. Результат вычисления сопоставляемого с образцом выражения есть результат вычисления выражения D′. Для библиотечных выражений с предопределенной семантикой выражение D в семантической схеме опускается. Выражение D может содержать выражение (return U), семантика которого заключается в завершении вычисления выражения D. В этом случае значение U есть значение D. Специальный случай (return) есть сокращение для (return ()). Например, выражение one удовлетворяет схеме sem one := (return 1) и вычисление этого выражения возвращает 1. Выражение (sem A var C where B := D), вычисляемое в состоянии s при означивании vv, добавляет новую семантическую схему со значением (sem vv(A, V) var C where vv(B, V) := vv(D, V)), где V — множество переменных образца, определяемое атрибутом var.

Обозначение. Пусть A′, B′, A'_2,... обозначают результаты вычисления выражений A, B, A_2, ... В случае нескольких выражений раньше вычисляется выражение, меньшее в лексикографическом порядке для букв и числовом для индексов. Например, выражение A вычисляется раньше выражения B, а выражение A_1 — раньше выражения A_2. Пусть s′ обозначает состояние, полученное после вычисления этих выражений.

3. Программные модели

Как уже было сказано выше, методы и техники верификации применяются не к исходным текстам аннотированных программ, а к программным моделям, которые представляют собой помеченные направленные упорядоченные графы. В этом разделе мы покажем, как программные модели описываются на языке Atoment на примере конструкций языков C и C#. Мы ограничиваемся теми конструкциями, которые используются в следующих разделах.

Аннотированный оператор while с условием A, телом B и инвариантом I соответствует элементу со значением (while-statement@ condition A body B inv I). Таким образом, элемент, являющийся моделью оператора while, есть экземпляр понятия while-statement и имеет атрибуты condition, body и inv, которые определяют условие A, тело B и инвариант I этого оператора соответственно.

Оператор if с условием A, then-ветвью B и else-ветвью C соответствует элементу со значением (if-statement@ condition A then B else C).

Оператор break соответствует элементу со значением (break-statement@), а оператор continue — (continue-statement@).

Оператор return, возвращающий выражение A, соответствует элементу со значением (return-statement@ expression A), а оператор return, не возвращающий

значения, — `(return-statement@)`.

В языке C# операторы goto бывают трех видов. Обычный оператор goto с меткой `A` соответствует элементу со значением `(goto-statement@ label A)`. Оператор goto с case-меткой `A` соответствует элементу со значением `(goto-statement@ case A)`. Оператор goto с default-меткой соответствует элементу со значением `(goto-statement@ default@@)`. Таким образом, если экземпляр понятия `goto-statement` имеет свойство `default`, он является моделью оператора goto с default-меткой.

Оператор switch с управляющим выражением `A` и телом `B` соответствует элементу со значением `(switch-statement@ expression A body B)`.

Оператор блока `{`A_1 `...` A_n`}` соответствует элементу со значением `(block-statement@` A_1 `...` A_n`)`.

Другие конструкции языков C и C# транслируются в их модели аналогичным образом. Важно отметить, что трансляция сопоставляет каждой конструкции аннотированной программы в точности один элемент на языке Atoment. Также при такой трансляции сохраняется терминология, используемая в этих языках программирования. Таким образом, трансляция текстов аннотированных программ на языках C и C# в программные модели на языке Atoment удовлетворяет принципам структурной идентичности и естественности.

4. Операционная семантика

Если трансляция программ языка программирования в программные модели удовлетворяет принципам структурной идентичности и естественности, операционную семантику достаточно определить на программной модели. В этом разделе мы покажем, как операционная семантика описывается на языке Atoment на примере определения механизма передачи управления в языке C# [3]. Этот механизм задается спецификацией операторов перехода (jump statements) и операторов обработки переходов. Мы рассмотрим только несколько операторов. Спецификация остальных определяется аналогичным образом. Детальная информация может быть найдена в [4, 5]. Спецификация операторов базируется на операционно-онтологическом подходе [6]. Этот подход вводит новый вид операционной семантики – операционно-онтологическую семантику, в которой состояния определяются в онтологических терминах (понятие, атрибут и т. п.).

Прежде всего опишем понятие перехода. Понятие `jump` задает элементы, называемые переходами, которые порождаются операторами перехода языка C#. Оно определяется выражением `(A is jump)` со следующей семантикой:

```
(sem (A is jump) var A :=
 (return ((A is break-jump) or (A is continue-jump) or
          (A is return-jump) or (A is exception-jump) or
          (A is label-jump) or (A is case-jump) or
          (A is default-jump))))
```

Понятия `break-jump`, `continue-jump` и `return-jump` описывают переходы, вызываемые операторами break, continue и return соответственно. Понятие

`exception-jump` описывает исключения, посылаемые операторами throw или средой исполнения программы. Понятия `label-jump`, `case-jump` и `default-jump` описывают переходы, вызываемые соответствующими видами операторов goto. Выражение `(A is B)`, если оно не переопределено семантическими схемами для конкретных `A` и `B`, возвращает `true`, если A' — экземпляр понятия B' и `()`, в противном случае. Выражение (`A`$_1$ `or ... or` `A`$_n$) определяет дизъюнкцию выражений A'_1, ..., A'_n. Ложное значение специфицируется пустым выражением `()`, а истинностное значение — любым другим выражением.

Операционная семантика операторов перехода и операторов обработки переходов задается выражением `(execute A)`, где `A` — соответствующий оператор.

Семантика оператора break имеет вид:

```
(sem (execute A) var A where (A is break-statement) :=
 (return (new (break-jump@))))
```

Выражение `(new U)`, вычисляемое в состоянии `s`, порождает новый элемент `E` со значением U', где U' — результат вычисления подвыражений всех слотов выражения `U`, и расширяет область определения функции `s` на этот элемент (`s'(E) = U'`). В нашем случае порождается новый экземпляр понятия `break-jump`. Специальный случай `(new)` есть сокращение для `(new ())`.

Семантика оператора return имеет вид:

```
(sem (execute A) var A where (A is return-statement) :=
 (seq@
  (var@ B)
  (if (A . value) then (B := (execute (A . value))))
  (if (B is jump) then (return B))
  (if (A . value) then (return (new (return-jump@ value B)))
   else (return (new (return-jump@)))))
```

Выражение (`var@` `U`$_1$ `...` `U`$_n$) определяет переменные U_1, ..., U_n, присваивая им значение `()`. Область существования этих переменных ограничена ближайшими круглыми скобками, окружающими это выражение. В нашем случае определяется переменная `B`, которая хранит результат выполнения выражения оператора return. Выражение `(U . V)` возвращает значение атрибута V' для элемента U'. Выражение `(if U then V else W)` аналогично обычному условному оператору. Условие U' ложно тогда и только тогда, когда оно возвращает ложное значение `()`. Специальный случай `(if U then V)` есть сокращение для `(if U then V else ())`. Выражение `(U := V)` присваивает переменной `U` значение V'. Выражение (`seq@` `A`$_1$ `...` `A`$_n$) последовательно вычисляет A_1, ..., A_n и возвращает A'_n.

Семантика оператора while имеет вид:

```
(sem (execute A) var A where (A is while-statement) :=
 (seq@
  (var@ B) (B := (execute (A . condition)))
  (if (B is jump) then (return B))
  (if (B = true)
```

```
then
(seq@
 (var@ C) (C := (execute (A . body)))
 (if ((C is jump) and (not (C is continue-jump)))
  then (return C) else (execute A)))
else ((return)))))
```

Выражение `(A = B)` возвращает `true`, если выражения A' и B' совпадают, и `()` в противном случае. Выражение `(A and B)` возвращает `true`, если A' = `true` и B' = `true`, и `()` в противном случае. Выражение `(not A)` возвращает `true`, если A' = `()`, и `true` в противном случае.

5. Трансформационная семантика

Трансформационная семантика описывает, как программы одного языка сводятся к программам другого языка (в частности, подъязыка этого же языка) посредством трансформаций. Мы рассмотрим спецификацию трансформации, элиминирующей оператор break. Эта трансформация взята из алгоритма трансляции C-light-программ в C-kernel-программы [7, 8], используемого в двухуровневом методе верификации C-программ [9, 10, 11]. Трансформация рекурсивно разбирает операторы программы до тех пор, пока не встретит оператор break. Все вхождения этого оператора заменяются на операторы goto в соответствии со следующими правилами:

```
switch(e){A break; B} -> {switch(e){A goto L; B} L:}
while(e){A break; B} -> {while(e){A goto L; B} L:}
```

Здесь `A, B` — программные фрагменты, L — метка.

Прежде всего определим понятие `label-place`. Оно описывает место, на которое передается управление в случае, если мы встретили оператор break. Атрибут `statement` этого понятия указывает на `... {A break; B}`, если в `A` нам не встретился оператор break, и указывает на блок `{... L:}` в противном случае. Метка `L` определяется атрибутом `label` этого понятия.

Трансформация описывается выражением `(eliminate-break A label-place LP)` со следующей семантикой:

```
(sem (eliminate-break A label-place LP) var (A LP)
 where ((A is statement) or (A = ())) :=
 (seq@
  (if ((A is while-statement) or (A is switch-statement))
   then (seq@
         (eliminate-break (A . body)
          label-place (new (label-place@ statement A)))
         (return)))
  (if (A is break-statement)
   then
```

```
 (seq@
  (var@ Lab)
  (if (LP . label)
   then (Lab := (LP . label))
   else // создание нового блока {... L:}
   (seq@
    (Lab := (new))
    (var@ (LabSt St St1))
    (LabSt := (new (labelled-statement@ label Lab)))
    (St := (LP . statement))
    (St1 := (new))
    (value St1 := (value St))
    (value St := (block-statement@ St1 LabSt))))
  (value A := (new (goto-statement@ label Lab)))
  (return)))
(if (A is block-statement)
 then
 (seq@
  (foreach X in A do (eliminate-break X label-place LP))
  (return)))
(if (A is if-statement)
 then
 (seq@
  (eliminate-break (A . then) label-place B)
  (eliminate-break (A . else) label-place B)
  (return)))
... // другие операторы
(if (A = ()) then (return))))
```

Выражение `(value A := B)` присваивает элементу `A′` значение `B′`. Выражение `(value A)`, вычисляемое в состоянии `s`, возвращает выражение `s(A′)`. Выражение `(foreach X in Y do Z)` для каждого слота-значения выражения `s′(Y′)`, перебираемых слева направо, вычисляет выражение, полученное заменой всех вхождений атомента `X` в `vv′(Z, {X})` на соответствующий слот-значение. Здесь `(Y′, s′, vv′)` — результат вычисления выражения `Y`.

6. Аксиоматическая семантика

В этом разделе мы покажем, как аксиоматическая семантика описывается на языке Atoment на примере спецификации генератора условий корректности для языка C#. Мы представим общую структуру генератора и спецификацию правила для оператора while. Правила для других операторов описываются аналогичным образом. Генератор реализует стратегию прямого прослеживания, которая применяется к первому оператору S программного фрагмента $S\,D$ тройки Хоара $\{P\}\,S\,D\,\{Q\}$, где P и Q — пред- и постусловия. Так, специфицируемое правило для оператора

while при применении этой стратегии имеет вид:

$$\frac{P \Rightarrow Inv \quad \{Inv \wedge B\}\, C\, \{Inv\} \quad \{I \wedge not(B)\}\, D\, \{Q\}}{\{P\}\, S\, D\, \{Q\}},$$

где Inv — инвариант while-оператора S вида $while(B)\, C$.

Генератор описывается выражением `(generate A)`, где `A` — список троек Хоара. Это выражение возвращает список порожденных условий корректности:

```
sem (generate A) var A where (A is (list Hoare-triple)) :=
(seq@
 (if (A is empty-expression) then (return)
  else
  (seq@
   (var@ HT FRAG S HT-pre HT-post)
   (HT := (A . 1 right@@)) (FRAG := (HT . fragment))
   (HT-pre := (HT . pre)) (HT-post := (HT . post))
   (if (FRAG is empty-expression)
    then (return (exp (HT-pre implies HT-post))))
   (var@ S) (S := (FRAG . 1))
   (if (S is while-statement) // правило для оператора while
    then
    (seq@
     (var@ Inv B) (Inv := (S . inv)) (B := (S . condition))
     (A += (new (Hoare-triple@ pre (exp (Inv and B))
                 fragment (S . body) post Inv))
     (delete FRAG . 1)
     (A += (new (Hoare-triple@ pre (exp (Inv and (not B)))
                 fragment FRAG post HT-post))))
    (return (add (exp (HT-pre implies Inv)) to (generate A)))))
   ... // другие операторы
  )))
```

Понятие `Hoare-triple` описывает тройки Хоара. Экземпляры этого понятия имеют обязательные атрибуты `pre`, `post` и `fragment`. Атрибуты `pre` и `post` соответствуют P и Q и определяют пред- и постусловия соответственно. Атрибут `fragment` соответствует программному фрагменту $S\, D$. Экземпляры понятия `(list T)` суть списки экземпляров понятия `T`. Понятие `empty-expression` имеет единственный экземпляр — пустое выражение `()`. Выражение `(A . B right@@)` возвращает `B`′-й с конца элемент списка `A`′. Свойство `right` означает, что подсчет элементов ведется справа налево. Выражение `(exp A)` возвращает выражение `vv(A)` и не меняет состояния и означивания переменных. Выражения `(A implies B)`, `(A and B)` и `(not A)` представляют соответствующие пропозициональные формулы языка аннотаций. Выражение `(A += B)` добавляет выражение `B`′ в конец списка `A`′. Выражение `(delete A . B)` удаляет `B`′-й элемент списка `A`′. Выражение `(add A to B)` добавляет элемент `A`′ в конец списка `B`′ и возвращает полученный список.

7. Родственные работы

Язык Atoment разрабатывался как внутренний язык расширяемой мультиязыковой системы анализа и верификации программ СПЕКТР-2 [12] в рамках интегрированного подхода к верификации императивных программ [13], базирующегося на использовании предметно-ориентированного языка описания средств анализа и верификации программ. Наиболее близким к нашему подходу является подход, который условно может быть назван подходом универсального промежуточного языка верификации. Суть этого подхода заключается в следующем. Определяется язык, называемый универсальным промежуточным языком верификации. Этот язык обеспечивается развитыми средствами анализа и верификации. Верификация программ на любом другом (целевом) языке программирования в рамках этого подхода выполняется в два этапа. На первом этапе программа целевого языка транслируется в программу унифицированного промежуточного языка. На втором этапе к транслированной программе применяются соответствующие средства верификации унифицированного промежуточного языка.

Рассмотрим два наиболее известных представителя класса универсальных промежуточных языков верификации — языки Why и Boogie.

Why (Франция, INRIA) [14] — это платформа, пригодная для верификации многих императивных языков. В ней определен одноименный промежуточный язык Why, в который транслируются программы целевых языков программирования. Цель трансляции — генерация условий корректности в виде, не зависящем от конкретной системы доказательства теорем. Ключевой особенностью этой платформы является богатая библиотека преобразований форматов доказательства, позволяющая создать подходящий входной формат для большого числа существующих доказателей, включая SMT-решатели Simplify, Alt-Ergo, Yices, Z3, CVC3, Gappa и интерактивные системы поддержки доказательства Coq, PVS, Isabelle/HOL, HOL 4, HOL Light, Mizar. Ядро платформы включает следующие инструменты: генератор условий корректности Why, который используется в качестве постпроцессора (back-end) другими средствами верификации, средство Krakatoa для верификации Java-программ, средства Caduceus и Frama-C для верификации C-программ.

Язык Boogie первоначально разработан Microsoft Research в качестве внутреннего языка компоненты статической верификации системы Spec#, ориентированной на разработку надежного программного обеспечения на языке Spec#, который расширяет язык C# конструкциями для ненулевых типов, проверяемых исключений и API контрактами (пре- и постусловиями, инвариантами объектов). Инструментарий для Boogie порождает условия корректности для Boogie-программ, которые затем передаются SMT-решателю. По умолчанию используется решатель Z3. Тесная интеграция этого языка в мультиязыковую среду разработки Microsoft Visual Studio определила его выбор в качестве промежуточного языка в ряде средств верификации, среди которых HAVOC и vcc для языка C, Dafny для одноименного языка, средство верификации для параллельного языка Chalice, AutoProof для языка Eiffel.

Чтобы сделать сравнение компактным, зафиксируем несколько обозначений. Пусть TL обозначает целевой язык (Target Language), UVIL — универсальный промежуточный язык (Universal Verification Intermediate Language). Подход на базе уни-

версального промежуточного языка верификации будем называть UVIL-подходом, а проект в котором он реализуется, UVIL-проектом.

Отличия между подходами заключаются в следующем:

- UIL-подход требует трансляции TL-программ в UVIL-программы. Эта трансляция может оказаться сложной задачей (например, трансляция Spec#-программ в Boogie-программы) не только в плане разработки соответствующего алгоритма, но и в плане доказательства его корректности, так как конструкции языка UVIL жестко фиксированы. В нашем подходе, поскольку модели могут быть произвольными в силу гибкости и расширяемости языка Atoment, можно выбрать модель, удовлетворяющую принципам структурной идентичности и естественности. В этом случае алгоритм трансляции будет прямолинейным, так как TL-программы транслируются в идентичные им модели, и соответственно нам не нужно будет обосновывать его корректность.

- В случае, если трансляция языка TL в язык UVIL выполнена, для TL-программ сразу становятся доступными все методы и техники верификации, накопленные для языка UVIL. (Однако при этом требуется обеспечить обратные зависимости, которые позволили бы выполнять интерпретацию результатов верификации для языка UVIL в терминах языка TL. Сложность этих зависимостей определяется тем, насколько отличаются языки TL и UVIL.) В нашем подходе для языка TL требуется разрабатывать свои средства верификации.

- Для UIL-подхода характерна "закрытость" разработки, которая означает, что средства верификации, разрабатываемые и реализуемые участниками (экспертами и программистами) UIL-проекта, являются черными ящиками для пользователей соответствующей системы верификации. Пользователи вынуждены ждать появления новых версий системы или включения новых средств в текущую версию. В то же время сложность разработанных средств верификации и их недостаточная документированность препятствуют самостоятельному анализу и модификации таких систем обычными пользователями даже при наличии исходного кода. В нашем подходе пользователь может как сам разрабатывать средства верификации, специфичные решаемой им задачи, так и использовать уже имеющиеся, при необходимости легко модифицируя и подстраивая их под свою задачу. Кроме того, разработка новых средств верификации в UVIL-подходе требует их реализации на языке программирования общего назначения, в то время как в нашем подходе она сводится к их спецификации на предметно-ориентированном языке Atoment в привычном концептуальном окружении. Удобный специализированный язык позволяет пользователю описывать в естественной нотации методы и техники верификации, переносить разработанные техники с одного языка программирования на другой, верифицировать алгоритмы в различных предметных областях, добавляя при необходимости свои языки для их представления, разделять методы и техники верификации с другими пользователями системы и комбинировать их.

8. Заключение

В статье представлены типовые примеры применения языка Atoment для создания программных моделей, спецификации языков программирования, разработки операционной, трансформационной и аксиоматической семантик. Они могут использоваться в качестве образцов для решения более сложных задач. Мы планируем использовать язык Atoment для разработки программных моделей языков программирования, которые широко используются на практике (Java, C/C++, C# и т. д.), формальных выполнимых спецификаций этих языков на базе операционно-онтологического подхода [6] и в качестве встроенного языка расширяемой мультиязыковой системы верификации СПЕКТР [12].

Список литературы

1. Ануреев И.С. Язык Atoment: синтаксис и семантика. Новосибирск, 2010. 39 с. (Препр./РАН. Сиб. отд-ние. ИСИ; №157).

2. Ануреев И.С. Язык Atoment: стандартная библиотека. Новосибирск, 2010. 32 с. (Препр./РАН. Сиб. отд-ние. ИСИ; №158).

3. Standard ECMA-334. C# Language Specification. 4th edition (June 2006). http://www.ecma-international.org/publications/standards/Ecma-334.htm.

4. Ануреев И.С. Операционно-онтологическая семантика операторов безусловной передачи управления в языке C# // Материалы международной конференции "Космос, астрономия и программирование" (Лавровские чтения) / Санкт-Петербургский государственный университет. СПб., 2008. С. 259–266.

5. Ануреев И.С. Операционно-онтологическая семантика обработки исключений // Материалы международной конференции "Космос, астрономия и программирование" (Лавровские чтения) / Санкт-Петербургский государственный университет. СПб., 2008. С. 15–22.

6. Ануреев И.С. Операционно-онтологический подход к формальной спецификации языков программирования // Программирование. 2009. №1. С. 1–11.

7. Непомнящий В.А., Ануреев И.С., Промский А.В. На пути к верификации C-программ. Язык C-light и его трансформационная семантика // Проблемы программирования. 2006. №2–3. С. 359–368.

8. Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В. Ориентированный на верификацию язык C-light // Формальные методы и модели информатики: Сборник научных трудов. Серия «Системная информатика». Новосибирск: Издательство СО РАН. 2004. №9. С. 51–134.

9. Nepomniaschy, V. A., Anureev, I. S., Promsky, A. V. Verification-oriented language C-light and its structural operational semantics // Proc. of Conf. 2003. PSI-2003, LNCS 2890. P. 1–5.

10. Непомнящий В.А., Ануреев 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.

11. Непомнящий В.А., Ануреев И.С., Промский А.В. На пути к верификации С программ. Аксиоматическая семантика языка C-kernel // Программирование. 2003. №6. С. 5–15.

12. Непомнящий В.А., Ануреев И.С., Атучин М.М., Марьясов И.В., Петров А.А., Промский А.В. Система анализа и верификации С-программ СПЕКТР-2 // Моделирование и анализ информационных систем. 2010. Том. 17, №4. С. 88–100.

13. Anureev I.S. Integrated approach to analysis and verification of imperative programs // Joint NCC&IIS Bulletin, Series Computer Science. 2011. Vol. 32. 18 p. (To appear).

14. Why Home Page. – http://why3.lri.fr/.

15. Boogie: A Modular Reusable Verifier for Object-Oriented Programs / Barnett M., Chang B., DeLine R., Jacobs B. and Leino R. // FMCO 2005. 2006. LNCS 4111. P. 364–387.

Typical Examples of Atoment Language Using

Anureev I.S.

Keywords: verification, specification, operational semantics, axiomatic semantics, transformational semantics, domain-specific languages, verification systems

Atoment is a domain-specific language of executable specifications, used to describe methods and techniques of program verification. In this paper a collection of typical examples of the use of the Atoment language, covering topics such as program models, operational, transformational and axiomatic semantics, formal specification of programming languages is presented.

Сведения об авторе:
Ануреев Игорь Сергеевич,
Институт систем информатики имени А.П. Ершова СО РАН,
старший научный сотрудник