Направленная генерация тестовых данных для анализаторов статической семантики.

С.В. Архипова, С.В. Зеленов

Аннотация. В статье представлен метод SemaTESK[1] автоматической генерации множеств тестов для фронт-эндов в трансляторах. Метод ориентирован на тестирование анализаторов статической семантики. Наиболее известные методы генерации семантических тестов работают путем фильтрации предварительно сгенерированных более или менее случайным образом синтаксических тестов. В отличие от этих методов, SemaTESK позволяет непосредственно генерировать тесты для контекстных условий. Это очень ощутимо сокращает время генерации и позволяет достигать сформулированные в статье критерии полноты. Предложенный метод специфицирования статической семантики позволяет формализовать неформальные требования, содержащиеся в нормативных документах (например, в стандартах). Метод включает в себя язык SRL для компактного формального специфицирования контекстных условий, а также инструмент STG для эффективной генерации множеств тестов из SRL-спецификаций. Метод SemaTESK был успешно применен в ряде проектов, в том числе по тестированию анализаторов статической семантики языков C и Java.

1. Введение

Формальные языки чрезвычайно широко используются практически во всех сферах ИТ: языки программирования являются основным инструментом при разработке ПО, языки запросов используются для управления базами данных, языки разметки применяются в различных системах обработки документов (например, в браузерах, текстовых процессорах), и т.п. Транслятор — это программа, которая преобразует текст с формального языка в некоторую подходящую для дальнейшей работы с ним форму. Так, например, компилятор транслирует программу в исполняемый модуль, СУБД транслирует запрос с языка высокого уровня (такого, как SQL) в последовательность низкоуровневых опреаций над базой данных, браузер транслирует информационную страницу в команды рисования, и т.п.

[1] SemaTESK — абревиатура от «Semantics Testing Kit» (инструментарий тестирования семантики – *англ.*)

Ошибки в трансляторе приводят к трудно-диагностируемым и трудно-исправимым ошибкам в результатах трансляции, поскольку свойства результата трансляции при этом непредсказуемым образом расходятся с требованиями спецификации входного языка. Отсутствие уверенности в корректности работы транслятора ставит под сомнение корректность результатов трансляции. Таким образом валидация и верификация транслятора является важнейшей деятельностью для успешного его внедрения.

Валидация и верификация транслятора всегда очень сложна в силу чрезвычайной сложности его входных и выходных данных. На вход транслятор принимает документ с сильно разветвленной синтаксической структурой и огромным набором контекстных условий, накладываемых спецификацией языка. Выходные данные представляются на машинном или некотором промежуточном языке и обладяют подобной, а порою и гораздо большей сложностью.

Как правило, чтобы подступиться к задаче высокой сложности, ее предварительно разбивают на несколько подзадач. Традиционно, задачу верификации и валидации транслятора разбивают на подзадачи в соответствии с различными аспектами функциональности транслятора, которые в сумме покрывают всю его функциональность.

Обычно транслятор включает в себя следующие функции:

1. анализ синтаксической корректности входного текста, его синтаксический разбор;
2. проверка статической семантики[2] входных данных;
3. генерация выходных данных.

В настоящей статье мы остановимся на задаче валидации и верификации проверки статической семантики. Мы употребляем словосочетания «статическая семантика» и «контекстные условия» как синонимичные. В рамках статьи мы рассматриваем целевую функцию проверки статической семантики как булевскую функцию $f: S(L) \rightarrow B$, где $S(L)$ — это множество синтаксически корректных предложений данного формального языка L, а $B = \{\texttt{true}, \texttt{false}\}$. Для данного предложения s функция f возвращает результат **`true`**, если s удовлетворяет всем контекстным условиям языка L, и результат **`false`**, если в s нарушается хотя бы одно контекстное условие языка L.

[2] Для данного формального языка его *статическая семантика* описывает такие свойства, которые могут быть проверены во время компиляции (к таковым, например, относятся: ограничения областей видимости, именования объектов, типовые ограничения и т.п.), в то время как *динамическая семантика* описывает свойста языка, относящиеся ко времени выполнения (т.е. значение, смысл языковых конструкций).

В качестве основного средства верификации и валидации мы используем тестирование [3] на основе формальных спецификаций и моделей [20]. В ходе тестирования качество целевого ПО оценивается на некоторых специально созданных входных данных. При этом встает следующая проблема: множество тестовых данных должно быть представительным, чтобы результаты тестирования действительно отражали реальное качество целевого ПО. Кроме того, существует еще одна проблема: богатство конструкций языка являются причиной того, что представительное множество тестов обязано содержать очень большое количество тестовых ситуаций, вследствие чего создавать качественные тесты вручную оказывается практически невозможно. Использование формального описания целевого ПО позволяет как формулировать адекватные критерии тестового покрытия, так и организовать автоматическую генерацию тестов.

1.1. Близкие работы

В настоящее время традиционно для спецификации свойств формального языка используются грамматики некоторого подходящего вида.

В подходах, представленных в работах [8], [9], [21], используются грамматики в виде расширенной BNF, снабженные специальными фрагментами программного кода, которые содержат информацию семантического характера: разного рода вспомогательные вычисления и проверки условий, управляющие генерацией соответствующих частей тестов. Однако, грамматики такого рода не могут претендовать на роль *спецификации* языка. Они скорее похожи на *программы* для генерации тестов.

В работе [4] описан генератор тестов Korat, который использует спецификацию тестовых данных в виде java-метода, проверяющего корректность структуры данных. Кроме того, генератору требуется предоставить набор параметров, отвечающих за то, чтобы множество генерируемых данных было конечным.

В статье [18] представлен подход TestEra для тестирования Java-программ на основе спецификаций. Спецификации записываются в виде формул логики первого порядка на языке Alloy [14]. В этом подходе также требуется задавать ограничения на размер генерируемых тестов.

К сожалению, ни Korat, ни TestEra не позволяют формулировать адекватные критерии полноты. В обоих подходах полностью генерируются все неизоморфные структуры, удовлетворяющие заданным ограничениям. Дополнительным препятствием к использованию этих подходов для тестирования анализаторов статической семантики является то, что найти подобный анализатор (как эталонный, так и целевой), написанный на Java, практически не возможно.

В работе [7] предложен метод автоматического тестирования инструментов для рефакторинга. Чтобы сгенерировать тесты для определенного вида рефокторинга требуется разработать соответствующий генератор на базе

специализированной библиотеки ASTGen. Основным недостатком этого подхода является то, что генератор приходится разрабатывать вручную.

В статьях [11], [12] представлены подходы к автоматической генерации тестов для анализаторов статической семантики, основанные на использовании спецификации семантики в виде атрибутной грамматики [19]. В работе [16] к этой задаче предложен другой похожий подход, в котором используются спецификации в форме ASM [10]. Авторы этих подходов рассматривают различные критерии тестового покрытия для тестирования анализатора статической семантики и предлагают соответствующие автоматические генераторы тестов. Основной принцип работы этих генераторов состоит в том, что сперва генератор создает множество синтаксически корректных предложений языка, которые затем проверяются на коректность с помощью соответствующего интерпретатора атрибутной грамматики или ASM, при этом во множество тестов попадают только семантически корректные предложения.

Таким образом, в этих подходах спецификация семантики используется лишь для того, чтобы сформулировать критерии прокрытия, а также чтобы *проверить* семантическую корректность сгенерированных *синтаксически корректных* предложений. Иными словами, собственно *генератор* предложений использует из всей спецификации только информацию о *синтаксической* структуре, а спецификация семантики при генерации *напрямую* не используется. В общем случае процесс генерации занимает громадное количество времени, поскольку для того, чтобы получить несколько сотен семантически корректных тестов, генератору приходится создать миллионы синтаксически корректных предложений-претендентов. В статьях [12], [17] авторы этих подходов предлагают некоторые оптимизации процесса генерации, основанные на том, что генератор пытается определять некорректность подпредложений в генерируемом предложении как можно раньше, до окончательной генерации всего предложения, однако по сути процесс генерации остается прежним.

1.2. Основные идеи предлагаемого метода

Прежде чем сформулировать основные идеи предлагаемого метода, обсудим подробнее подходы к генерации тестов для анализаторов статической семантики в контексте использования атрибутных грамматик. Подобные аргументы верны также и для подходов, основанных на ASM.

Атрибутные грамматики хорошо себя зарекомендовали для разработки анализаторов статической семантики. Однако, как представляется, они не пригодны для направленной генерации тестов. Спецификацию статической семантики языка в виде атрибутной грамматики можно трактовать как предикат P от дерева абстрактного синтаксиса t, построенного по предложению языка. Для данного дерева не представляет никакой сложности вычислить значение этого предиката, т.е. *проверить* семантическую

корректность. Однако для того, чтобы *построить* семантически корректный тест, требуется решить уравнение *P(t)* = **`true`** относительно *t*. Для достижения этой цели в подходах, основанных на использовании атрибутных грамматик, осуществляется перебор различных деревьев и проверка для них значения предиката *P*.

Мы считаем, что наиболее эффективным способом генерации семантиччески корректных тестов является использование некоторого алгоритма, который направленно строит решения уравнения *P(t)* = **`true`**. С нашей точки зрения основным препятствием для решения этого уравнения в подходах, основанных на использовании атрибутных грамматик, является форма классической атрибутной грамматики. Атрибутные грамматики – это очень мощный инструмент, однако они обладают рядом слабых мест, которые мы продемонстрируем на следующем примере.

Пример 1

Рассмотрим некоторый процедурный язык программирования, в котором конструкции объявления переменной и присваивания значения переменной отделены друг от друга. При этом для этих конструкций имеется два контекстных условия:

- Все имена переменных, объявленных в одной процедуре, должны быть различны;
- Имя переменной в конструкции присваивания должно быть объявлено в той же процедуре.

В классической атрибутной грамматике соответствующая часть спецификации языка выглядит примерно так:

```
 1: Procedure ::= ( Stmt )*
 2:  {
 3:    attribute SymbolTable vars;
 4:    attribute Boolean ok;
 5:  };
 6:
 7: Stmt ::= VarDecl | Assignment | ... ;
 8:
 9: VarDecl ::= “var” <name:ID>
10:  {
11:    Procedure.ok &= !Procedure.vars.has( name );
12:    Procedure.vars.add( name );
13:  };
14:
15: Assignment ::= <name:ID> “=” ...
16:  {
17:    Procedure.ok &= Procedure.vars.has( name );
18:  };
```


Спецификация приведенных контекстных условий основана на использовании таблицы символов `vars`, объявленной в правиле `Procedure` (строка 3). В этой таблице собирается информация об именах переменных, объявленных в данной процедуре. Информация заносится в таблицу в правиле `VarDecl` (строка 12), а проверка наличия имени в таблице производится в правилах `VarDecl` (строка 11) и `Assignment` (строка 17). Результаты проверок накапливаются в булевском атрибуте `ok`, объявленном в правиле `Procedure` (строка 4).

Этот пример демонстрирует следующие слабые стороны классической атрибутной грамматики:

- атрибуты, объявленные в правиле `Procedure`, являются своего рода *глобальными переменными*, наличие которых влечет хорошо известные *проблемы поддержки* спецификации;
- для формализации *одного контекстного условия* требуется написать много строк кода в *нескольких различных частях* спецификации (так, например, второе контекстное условие формализуется в четырех строках (3, 4, 12 и 17), которые относятся к трем различным правилам), что приводит к слабой прослеживаемости[3] требований стандарта языка в коде спецификации.

►

Мы полагаем, что перечисленные слабые стороны являются основной причиной того, что направленную генерацию семантически корректных тестов не удается осуществлять на основе спецификаций в форме атрибутных грамматик. Действительно, для того, чтобы разрешать контекстные условия, записанные в такой форме, генератор должен иметь очень большой интеллект.

В настоящей статье мы представляем метод SemaTESK автоматической генерации тестов для анализаторов статической семантики. Этот метод реализует технологию UniTESK [5], [22] автоматизированного тестирования, основанную на использовании спецификаций и моделей целевой системы. Метод SemaTESK включает в себя специализированный язык SRL[4], предназначенный для записи формальных спецификаций контекстных условий в компактной форме, пригодной для неправленной генерации тестов, а также генератор тестов STG[5], который позволяет автоматически эффективно генерировать тесты по заданной SRL-спецификации статической семантики.

Статья состоит из семи разделов и списка литературы. Первый раздел является вводным и посвящен постановке проблем и обзору близких работ. В

[3] Английский эквивалент – слово «traceability».

[4] SRL – аббревиатура от «Semantics Relation Language» (язык семантических отношений – *англ.*).

[5] STG – аббревиатура от «Semantics Tests Generator» (генератор семантических тестов – *англ.*).

разделе 2 представлены основные возможности языка SRL. В разделе 3 сформулированы используемые в методе SemaTESK притерии полноты. Раздел 4 посвящен краткому изложению основных принципов работы генератора STG. В разделе 5 дается описание автоматического оракула. В разделе 6 представлены результаты практического применения метода. Раздел 7 завершает статью.

2. Язык SRL.

2.1. Основные особенности языка

Начнем со следующего примера.

Пример 2

Контекстное условие «Имя переменной в конструкции присваивания должно быть объявлено в той же процедуре» (см. Пример 1) записывается на SRL так:

```
one-to-many relation DeclareAssignedVarName
{
  ordered
  equal
  target Assignment {name}
  source VarDecl {name}
  context: same Procedure
}
```

Этот *дескриптор контекстного условия* можно прочитать следующим образом: «Если в предложении встретилось присваивание (`Assignment`), то в той же процедуре (**`same`** `Procedure`) должно также иметься объявление переменной (`VarDecl`), причем атрибут `name` объявления `VarDecl` равен (**`equal`**) атрибуту `name` присваивания `Assignment`, присваивание `Assignment` должно находиться после объявления `VarDecl` (т.е. присваивание и объявление должны быть упорядочены (**`ordered`**)), и имя `name` переменной, объявленной в одном (`one`) объявлении может быть использовано во многих (`many`) присваиваниях (ср. ключевое слово **`one-to-many`**)».

►

Этот пример иллюстрирует следующие основные особенности языка SRL:

- Статическая семантика языка формализуется в виде набора дескрипторов контекстных условий над атрибутированной контекстно-свободной грамматикой. В отличие от практики, принятой в атрибутных грамматиках, в SRL дескрипторы контекстных условий не относятся к каким-то конкретным правилам грамматики, поскольку зачастую бывает трудно выбрать, к какому правилу

отнести данный дескриптор (так, например, должны ли мы отнести дескриптор `DeclareAssignedVarName` (см. Пример 2) к правилу `Assignment`? или к правилу `VarDecl`? или к `Procedure`? Как мы видели ранее (см. Пример 1), в соответствующей атрибутной грамматике разные части кода для этого контекстного условия относятся ко всем этим упомянутым правилам!).

- Одному контекстному условию, выраженному на естественном языке в тексте стандарте данного языка, сответствует один блок кода (дескриптор контекстного условия) на SRL. Этим обеспечивается эффективная прослеживаемость требований стандарта языка в коде SRL-спецификации.
- Дескрипторы контекстных условий имеют форму отношений вида «объявление объекта – использование объекта»[6]. Основой дескриптора является пара конструкций, начинающихся с ключевых слов **`target`** и **`source`**. Грубо говоря, для любого вхождения в предложение элемента, описанного конструкцией **`target`** (что соответствует использованию объекта), должно существовать соответствующее вхождение элемента, описанного конструкцией **`source`** (что соответствует объявлению этого объекта).

В настоящее время как язык SRL, так и генератор STG продолжают развиваться. Каждое очередное применение генератора открывает новые перспективы его улучшения, «если бы только SRL обладал такими-то новыми возможностями». Тем не менее, основные принципы SRL остаются незыблемыми и представлены в последующих частях данного раздела.

2.2. Форма синтаксической грамматики

В методе SemaTESK для формализации контекстно-свободной грамматики, на основе которой будут описываться контекстные условия, используется язык TreeDL[7] [2], предназначенный для описания структуры абстрактного синтаксического дерева в виде набора возможных видов его узлов посредством спецификации для каждого вида узла множества его дочерних узлов и дополнительных атрибутов.

[6] Как показывает практика, большинство контекстных условий можно свести к такому виду. Однако, если этого сделать не удается, SRL-спецификацию контекстного условия можно снабдить подходящим Java-кодом (подробности см. в [1]).

[7] TreeDL – аббревиатура от «Tree Description Language» (язык описания деревьев – *англ.*)

Пример 3

Рассмотренная выше (см. Пример 1) часть контекстно-свободной грамматики может иметь следующее TreeDL-описание:

```
node Procedure
{
  child Stmt* statements;
}

abstract node Stmt
{}

node VarDecl : Stmt
{
  child ID name;
}

node Assignment : Stmt
{
  child ID name;
  ...
}

node ID
{
  attribute string value;
}
```

►

TreeDL-форма грамматики имеет перед BNF ряд преимуществ, которые делают ее более привлекательной для целей специфицирования статической семантики формальных языков:

- В рамках одного узла все дочерние узлы и атрибуты имеют уникальные имена, что позволяет однозначно обращаться к элементам узлов.
- В узлы можно добавлять дополнительные (т.е. относящиеся не к синтаксису, а к семантике) атрибуты.

2.3. Дескриптор контекстного условия

Основной минимальной единицей SRL-описания семантики является *дескриптор контекстного условия*. Каждый дескриптор определяет отношение между двумя узлами дерева, которые называются, соответственно, *источник* (специфицируется с помощью ключевого слова **`source`**) и *цель* (специфицируется с помощью ключевого слова **`target`**). В общих словах,

каждое такое отношение определяет, как структура узла-цели зависит от структуры узла-источника[8]. В большинстве случаев можно рассматривать узел-цель как точку использования некоторого объекта, а узел-источник как точку объявления этого объекта.

В дескрипторе `DeclareAssignedVarName` (см. Пример 2) узел-источник и узел-цель специфицированы посредством указания своих типов:

```
target Assignment {...}
source VarDecl {...}
```

Это описание означает, что целью для данного отношения может являться любой узел типа `Assignment`, а источником — любой узел типа `VarDecl`. Однако, источник и цель можно специфицировать и более подробно (см. подраздел 2.4).

В действительности каждый дескриптор контекстного условия определяет отношение между некоторыми поддеревьями в узле-источнике и узле-цели. В дескрипторе `DeclareAssignedVarName` (см. Пример 2) определяется отношение между полем `name` источника и полем `name` цели:

```
target Assignment {name}
source VarDecl {name}
```

Эти поддеревья можно трактовать как аргументы дескриптора контекстного условия. Зависимость между аргументами задается посредством указания подходящего вида зависимости. Для этого в языке SRL имеется несколько ключевых слов, с помощью которых, как показывает практика, возможно описать подавляющее большинство контекстных условий типичных языков программирования (см. подраздел 2.5). В дескрипторе `Declare-AssignedVarName` (см. Пример 2) использовано ключевое слово **`equal`** для указания того, что аргументы должны быть равны между собой.

В некоторых случаях контекстное условие определяет отношение только между узлами, которые находятся в некотором специфическом окружении. Так, дескриптор `DeclareAssignedVarName` (см. Пример 2) специфицирует отношение только между таким присваиванием и объявлением переменной, которые расположены в одной процедуре. Необходимое окружение специфицируется в дескрипторе контекстного условия с помощью ключевого слова **`context`** одним из перечисленных ниже двух способов:

- Если и источник, и цель должны располагаться в некотором *одном и том же* поддереве с корнем в узле типа `<RootNodeType>`, то используется конструкция SRL с ключевым словом **`same`** (см. Пример 2):

[8] Наименования «источник» и «цель» обусловлены тем, что знание о структуре узла-источника является своего рода источником информации для целей корректного формирования структуры узла-цели.

```
context: same <RootNodeType>
```

- Если источник и цель должны располагаться в *различных* поддеревьях с корнями в узлах, соответственно, типов <SourceRootNodeType> и <TargetRootNodeType> (эти типы могут быть как различными, так и совпадающими), то используется конструкция SRL с ключевым словом **differ** (см. Пример 4 ниже) при этом окружение узла-источника специфициуется с помощью ключевого слова **source_context**, а окружение узле-цели – с помощью ключевого слова **target_context**:

```
context: differ source_context
<SourceRootNodeType>
                target_context
<TargetRootNodeType>
```

Пример 4

Рассмотрим фрагмент TreeDL-описания Java-класса:

```
node Class
{
  attribute string name;
  attribute string baseClassName;
  ...
}
```

Здесь атрибут name означает имя класса, baseClassName – имя базового класса. Дескриптор контекстного условия «Имя базового класса должно являться именем некоторого другого класса» имеет следующую SRL-спецификацию:

```
one-to-many relation DeclaredBaseClass
{
  unordered
  equal
  target target_context { baseClassName }
  source source_context { name }
  context: differ source_context Class
                  target_context Class
}
```

Ключевое слово **differ** отражает тот факт, что исходный и базовый классы – суть *разные* классы.

►

Если узел-источник должен предшествовать узлу-цели в дереве[9], то дескриптор контекстного условия должен быть помечен ключевым словом **ordered** (см. Пример 2). В противном случае, он должен быть помечен ключевым словом **unordered** (см. Пример 4).

Для спецификации того, сколько узлов могут состоять друг с другом в том отношении, которое определяется данным дескриптором, в дескрипторе требуется указать тип отношения (см. подраздел 2.6). В дескрипторе DeclareAssignedVarName (см. Пример 2) тип отношения задан ключевым словом **one-to-many**, которое специфицирует то, что имя, определенное в одном (one) объявлениии переменной, может участвовать во многих (many) присваиваниях.

Далее в данном разделе приводятся детальные описания некоторых конструкций языка SRL, используемых в дескрипторах контекстных условий.

2.4. Задание узла

Узел-источник и узел-цель задаются посредством указания соответствующих путей в абстрактном синтаксическом дереве. Каждый путь имеет форму цепочки $e_1.\cdots.e_n$ элементов e_i, разделенных точками, и определяет в дереве некоторое соответствующее ему множество узлов. Это множество определяется индуктивно по длине цепочки элементов e_i следующим образом. Пусть S_0 — пустое множество, а S_k ($k = 1, \ldots, n$) — множество узлов, соответствущее пути $e_1.\cdots.e_k$. Тогда множество S_k ($k = 1, \ldots, n$) определяется множеством S_{k-1} и видом элемента e_k.

В языке SRL поддерживаются следующие виды элементов пути:

- <NodeType> – если $k = 1$, то такой элемент определяет все узлы типа <NodeType>; если $k > 1$, то такой элемент определяет все такие узлы типа <NodeType>, каждый из которых при этом является дочерним узлом для одного из узлов из множества S_{k-1} (см. Пример 5 ниже).
- <fieldName> – определяет все такие узлы, каждый из которых является полем (т.е. дочерним узлом или атрибутом) с именем <fieldName> в одном из узлов из множества S_{k-1} (см. Пример 5 ниже).
- **parent** – определяет все такие узлы, каждый из которых является родительским для одного из узлов из множества S_{k-1} (см. Пример 5 ниже).
- ^<ParentNodeType> – определяет все такие узлы, каждый

[9] Предшествование следует понимать в том смысле, что в тексте теста фрагмент, соответствующий источнику, должен располагаться до фрагмента, соответствующего цели.

из которых является ближайшим из (пра-)родительских узлов типа `<ParentNodeType>` для одного из узлов из множества S_{k-1} (см. Пример 5 ниже).

- **`target`** – определяет узел-цель данного дескриптора контекстного условия (такой элемент допустим только при спецификации узла-источника и только при $k = 1$, см. Пример 7 ниже).
- **`context`** – определяет узел, заданный конструкцией **`context: same`** данного дескриптора контекстного условия (такой элемент допустим только при $k = 1$, см. Пример 6 ниже).
- **`source_context`** и **`target_context`** – определяет узел, заданный, соответственно, конструкцией **`source_context`** и **`target_context`** данного дескриптора контекстного условия (такой элемент допустим только при $k = 1$, см. Пример 4).

Пример 5

В приведенной выше структуре дерева (см. Пример 3): описание узла `Assignment.name` задает все имена во всех присваиваниях; описание узла `Assignment.^Procedure.VarDecl` задает все объявления, содержащиеся в процедуре, в которой находится (какое-нибудь) присваивание, описание `ID.`**`parent`** задает все узлы, ссылающиеся на имя переменной.

►

2.5. Виды зависимостей

Для задания вида зависимости между аргументами дескриптора контекстного условия в языке SRL имеются следующие ключевые слова:

- **`equal`** – специфицирует, что значения аргументов дескриптора должны быть равны (точнее, что поддеревья аргументов дескриптора изоморфны – см. Пример 2).
- **`unequal`** – специфицирует, что значения аргументов дескриптора должны быть различны (точнее, что поддеревья аргументов дескриптора не изоморфны – см. Пример 8 ниже).
- **`present`** – специфицирует, что если в дереве имеется аргумент, заданный в конструкции **`source`**, то должен существовать и аргумент, заданный в конструкции **`target`** (см. Пример 6 ниже).
- **`absent`** – специфицирует, что если в дереве имеется аргумент, заданный в конструкции **`source`**, то аргумент,

заданный в конструкции **`target`**, должен отсутствовать (см. Пример 7 ниже).

- **`compatible`** – специфицирует, что аргумент, заданный в конструкции **`source`**, должен быть совместим (в смысле совместимости типов в языках программирования – см. подраздел 2.7) с аргументом, заданным в конструкции **`target`** (в этом случае аргументы дескриптора контекстного условия должны описывать типы – см. Пример 9 ниже).

Пример 6

Рассмотрим фрагмент TreeDL-описания Java-класса и Java-метода:

```
node Class
{
  attribute bool isAbstract;
  child Method* methods;
  ...
}

node Method
{
  attribute bool isAbstract;
  ...
}
```

Здесь атрибут `isAbstract` означает, что метод или класс является абстрактным. Дескриптор контекстного условия «Абстрактный метод может быть только в абстрактном классе» имеет следующую SRL-спецификацию:

```
many-to-many relation
AbstrMethodOnlyInAbstrClass
{
  unordered
  present
  target context { isAbstract }
  source context.Method { isAbstract }
  context: same Class
}
```

►

Пример 7

Рассмотрим фрагмент TreeDL-описания Java-метода:

```
node Method
{
  attribute bool isAbstract;
  attribute bool isStatic;
  ...
}
```

Здесь атрибут `isAbstract` означает, что метод является абстрактным, `isStatic` – что метод является статическим. Дескриптор контекстного условия «Статический метод не может быть абстрактным» имеет следующую SRL-спецификацию:

```
one-node relation
StaticMethodIsNotAbstract
{
  unordered
  absent
  target Method { isAbstract }
  source target { isStatic }
}
```

Конечно, перечисленные виды зависимостей не покрывают все многообразие тех ситуаций, которые могут возникнуть в семантике разных языков. На тот случай, когда ни один из встроенных видов зависимостей не подходит для специфицирования некоторого контекстного условия, в языке SRL имеется возможность описать специфический вид зависимости, воспользовавшись ключевым словом **`custom`** и задав дополнительный программный код, который реализует обработку данного дескриптора контекстного условия в генераторе тестов (однако, рассмотрение этой возможности выходит далеко за рамки настоящей статьи; подробности можно найти в [1]). Как показывает практика, эта возможность требуется чрезвычайно редко. Например, в таком сложном языке как Java лишь два контекстных условия (из почти трехсот) потребовали подобного описания специфического вида зависимости[10].

2.6. Тип отношения

Тип отношения в дескрипторе контекстного условия отражает то, сколько узлов могут состоять друг с другом в данном отношении. В языке SRL имеются следующие ключевые слова для задания типа отношения:

- **`one-to-many`** – специфицирует, что один (`one`) узел-источник может соответствовать многим (`many`) узлам-целям, и для каждого узла-цели должен существовать удовлетворяющий данному контекстному условию узел-источник (см Пример 2).
- **`many-to-many`** – специфицирует, что много (`many`) узлов-источников могут соответствовать многим (`many`) узлам-целям, и каждая пара из узла-источника и узла-цели (в случае, когда эти узлы не совпадают) удовлетворяет данному контекстному условию (см. Пример 8 ниже).
- **`one-node`** – специфицирует, что данное контекстное условие наложено на один узел (точнее, на поддерево с корнем в этом узле). Этот узел специфицируется как узел-цель в конструкции **`target`**, а расположение узла-источника определяется относительно узла цели (т.е. путь в конструкции **`source`** начинается с элемента **`target`** – см. подраздел 2.4, а также Пример 7 и Пример 9).

[10] В данном случае эти «нестандартные» контекстные условия относятся к семантике сигнатур методов.

Пример 8

Контекстное условие «Все имена переменных, объявленных в одной процедуре, должны быть различны» (см. Пример 1) записывается на языке SRL так:

```
many-to-many relation DifferentVarNames
{
  unordered
  unequal
  target VarDecl {name}
  source VarDecl {name}
  context: same Procedure
}
```

►

2.7. Совместимость типов

Существенную часть статической семантики языков программирования занимает семантика типов, которая обычно сводится к условиям совместимости типов переменных и выражений в различных контекстах.

В SRL спецификация того, какие типы в данном языке являются совместимыми, осуществляется посредством задания частично упорядоченных (в соответствии с отношением «типы совместимы») множеств типов данного языка. Такие множества задаются перечислением линейно упорядоченных подмножеств. Специфиакия цепочки совместимых между собой слева направо типов производится в SRL перечислением типов из этой цепочки внутри конструкции **`typeset`**.

Пример 9

Пусть присваивание `Assignment` (см. Пример 1) имеет своей правой частью некоторое выражение `Expression`:

```
  ...
Assignment ::= <name:ID> “=” Expression
  ...
```

Рассмотрим следующую соответствующую часть TreeDL-описания:

```
node Assignment : Stmt
{
  child ID name;
  child Expression rhs;
  attribute Type lhs_type;
}

abstract node Expression
{
  attribute Type type;
}
```

Для того, чтобы описать семантику типов, в узел `Assignment` добавлен атрибут `lhs_type`, а в узел `Expression` – атрибут `type`. Таким образом, контекстное условие «В переменную можно присваивать только значение совместимого типа» записывается на языке SRL так:

```
one-node relation AsgnTypes
{
  unordered
  compatible
  target Assignment {rhs.type}
  source target {lhs_type}
}
```

Пусть данный язык имеет типы `short`, `int` и `long`. Совместимость типов для этого языка специфицируется следующей цепочкой типов:

```
typeset PrimitiveTypes
{
  PrimitiveType.SHORT,
  PrimitiveType.INT,
  PrimitiveType.LONG
}
```

Здесь конструкция `PrimitiveType.SHORT` и прочие – суть возможные значения атрибута узла, отвечающего за тип выражения.

►

Язык SRL позволяет также описывать совместимость пользовательских типов (таких как классы в объектно-ориентированном языке программирования).

Рассмотрение этой возможности выходит за рамки настоящей статьи (подробности можно найти в [1]).

3. Критерии полноты

3.1. Семантически корректные тесты

Для данной SRL-спецификации на первый взгляд достаточно сформулировать следующий критерий полноты генерации семантически корректных тестов: «Для данной спецификации должны быть покрыты все дескрипторы контекстных условий», где покрытие одного дескриптора понимается в смысле следующего определения:

Определение 1. Данный дескриптор контекстного условия называется *покрытым*, если во множестве тестов существует такое предложение, чье абстрактное синтаксическое дерево содержит два узла, подходящих на роль источника и цели для данного дескриптора.

Пример 10

Предположим, что рассмотренный выше процедурный язык (см. Пример 1) позволяет использование вложенных блоков в теле процедуры. Любой разработчик компиляторов несомненно скажет, что следующие две ситуации заставят работать анализатор семантики по-разному, хотя они покрывают один и тот же дескриптор `DeclareAssignedVarName` (см. Пример 2):

```
{                    {
  var A;               var A;
  A = 1;               {
}                        A = 1;
                       }
                     }
```

Различия в работе анализатора на данных двух ситуациях обусловлено тем, что в варианте слева объявление и присваивание находятся в одном и том же блоке, а в варианте справа – в разных: присваивание находится во вложенном блоке с более глубоким уровнем вложенности.

►

Таким образом, сформулированный выше критерий «Все дескрипторы» не достаточен для хорошего тестирования. Этот критерий можно улучшить так: «Для данной спецификации должны быть покрыты все дескрипторы контекстных условий во всех возможных различных окружениях узла-источника и узла-цели», где окружение понимается в смысле следующего определения:

Определение 2. *Окружением* данного узла абстрактного синтаксического дерева называется цепочка узлов на пути от данного узла до корня дерева.

3.2. Семантически некорректные тесты

При тестировании анализатора статической семантики помимо проверки того, что на семантически корректных предложениях возвращается положительный вердикт, имеется еще одна задача: проверить, что на семантически некорректных предложениях возвращается отрицательный вердикт.

Если в предложении нарушено некоторое контекстное условие, естественно ожидать, что анализатор отвергнет это предложение и, кроме того, выдаст адекватную диагностику. Однако, если в предложении нарушено более одного контектного условия, то в общем случае бывает очень сложно предсказать соответствующую диагностику, поскольку анализатор может, например, завершить свою работу немедленно после обнаружения первой семантической ошибки, а может попытаться произвести восстановление своего внутреннего состояния, некоторым образом «домыслив», что должно быть в предложении на месте ошибочного фрагмента, причем, во-первых, стратегия восстановления от ошибки никогда не специфицируется в стандарте языка, а является частью реализации конкретного анализатора, а во-вторых, в результате такого восстановления могут возникнуть новые «наведенные» ошибки, которых не было в исходном предложении. В связи с этим мы предлагаем генерировать семантически некорректные тесты, каждый из которых содержит лишь одно нарушение контекстного условия.

Предположим, что существует генератор семантически корректных тестов. Для того, чтобы сгенерировать тесты, нарушающие некоторое контекстное условие, будем с помощью этого генератора генерировать тесты, которые удовлетворяют отрицанию данного контекстного условия в смысле следующего определения.

Определение 3. Для данного дескриптора контекстного условия R его *отрицанием* называется такой дескриптор R', что если предложение удовледворяет R', то это предложение нарушает R.

Пример 11

Пример отрицания дескриптора `DeclareAssignedVarName` (см. Пример 2) выглядит так:

```
many-to-many relation
DeclareAssignedVarName_neg
{
  ordered
  unequal
  target Assignment {name}
  source VarDecl {name}
  context: same Procedure
}
```

►

Пусть S – спецификация статической семантики некоторого языка, записанная на языке SRL, и пусть R – некоторый дескриптор из S. Рассмотрим следующее *R-отрицание* исходной спецификации: $S^{(R)} = \{ R' \} \cup S \setminus \{ R \}$. По аналогии со случаем семантически корректных тестов (см. подраздел 3.1) можно сформулировать следующий критерий полноты для семантически некорректных тестов: «Для данной спецификации S и для каждого дескриптора R из S должны быть покрыты все отрицания R' в $S^{(R)}$ во всех возможных различных окружениях узла-источника и узла-цели».

4. Генератор семантических тестов STG

Язык SRL предназначен для описания статической семантики целевого языка в виде, пригодном для автоматической генерации тестов для анализатора статической семантики. В методе SemaTESK, представляемом в данной статье, генератор тестов STG генерирует множества тестов, удовлетворяющие сформулированным в разделе 3 критериям полноты. Генератор STG принимает на вход TreeDL-грамматику целевого языка и SRL-спецификацию его статической семантики. Ядром генератора тестов является компонент, который строит синтаксические деревья в соответствии с данной грамматикой и данными контекстными условиями. Каждое построенное дерево затем транслируются в конкретное предложение целевого языка посредством обхода дерева и создания текстового представления узлов. Прежде чем сформулировать в общих чертах (подробности см. в [1]) алгоритм генерации тестов, дадим несколько определений.

Определение 4. Поддерево абстрактного синтаксического дерева называется *синтаксически полным*, если ему соответствует синтаксически корректное предложение данного языка.

Определение 5. Контекстное условие, соответствующее некоторому узлу-цели и описанное в некотором дескрипторе, называется *разрешимым* для данного синтаксического дерева, если в дереве имеются все необходимые элементы (узлы и атрибуты) в требуемых контекстах, которые вместе с узлом-целью удовлетворяют этому дескриптору.

Определение 6. Абстрактное синтаксическое дерево называется *семантически полным*, если ему соответствует семантически корректное предложение данного языка.

Заметим, что в семантиески полном дереве все контекстные условия являются разрешимыми.

Определение 7. Поддерево абстрактного синтаксического дерева называется *первичным* относительно данного дескриптора контекстного условия, если оно содержит узлы, удовлетворяющие спецификации источника и цели для этого дескриптора.

Дерево, являющееся первичным относительно некоторого дескриптора контекстного условия, можно трактовать как дерево, содержащее только узел-источник и узел-цель этого дескриптора с их окружениями.

Для каждого дескриптора контекстного условия из данной SRL-спецификации генератор STG генерирует семантически корректные тесты в соответствии со следующим алгоритмом:

1. Создается множество всех возможных первичных деревьев[11] (в соответствии с заданным значением глубины рекурсии).
2. Каждое первичное дерево t_{prime} из построенного множества генератор достраивает минимальным[12] образом до синтаксически полного дерева t.
3. Полученное синтаксически полное дерево t генератор пытается достроить до соответствующего семантически полного дерева t' (см. ниже).
4. Если удается построить семантически полное дерево t', то генератор печатает его текст на целевом формальном языке.

Для направленной генерации тестов, удовлетворяющих заданным контекстным условиям, генератор STG принимает во внимание как граф атрибутной зависимости, так и синтаксическую структуру строящегося дерева[13]. Построение семантически корректного теста происходит пошагово: в синтаксическом дереве, полученном на предыдущем шаге, генератор находит все неразрешимые контекстные условия (если таковых нет, то дерево уже является семантически полным) и пытается добиться их разрешимости путем модифицирования дерева в соответствии со следующими правилами:

1. Первичное дерево t_{prime} во всех случаях остается без изменений[14].
2. Если для неразрешимого дескриптора, специфицирующего отношение типа **`one-to-many`**, в дереве не нашлось подходящего узла-источника, то генератор пытается добавить в дерево новое поддерево, содержащее узел, удовлетворяющий спецификации источника в этом дескрипторе.
3. Если вид зависимости неразрешимого дескриптора требует наличия в дереве некоторого специфического узла, которого в дереве нет, то генератор пытается добавить в дерево новый узел с требуемыми свойствами (по аналогии с правилом 2).

Если генератору не удается добиться разрешимости некоторого дескриптора, то текущее семантически неполное дерево отбрасывается. Иначе генератор переходит к очередному шагу построения семантически корректного теста (поиск неразрешимых контекстных условий, которые могли появиться в результате изменений в дереве на предыдущем шаге, и попытки добиться их разрешимости).

[11] Это множество состоит из таких первичных деревьев, в каждом из которых имеется узел-цель и узел источник текущего дескриптора, причем в этом множестве для любой возможной комбинации окружений цели и источника имеется по крайней мере одно соответствующее первичное дерево.

[12] Все списки создаются с минимальным возможным размером, альтернативы раскрываются в наиболее простой вариант (например, пустое подпредложение, терминал и т.п.)

[13] В отличие от подходов, основанных на использовании атрибутных грамматик, в которых при переходе к работе с собственно семантической составляющей синтаксическая структура дерева забывается, а рассматривается лишь граф атрибутной зависимости.

[14] Ситуация, соответствующая первичному поддереву, является целевой (в смысле покрытия) для генерируемого теста.

5. Оракул

Мы рассматриваем анализатор статической семантики как булевскую функцию.

В методе SemaTESK для прогона сгенерированных семантически корректных тестов используется автоматический оракул, который выносит для данного теста положительный вердикт только если целевой анализатор вернул на этом тесте значение **`true`**. На практике это обычно означает, что анализатор статической семантики успешно завершил свою работу, не выдав никаких сообщений об ошибках.

Для прогона сгенерированных семантически некорректных тестов также используется автоматический оракул, который в свою очередь выносит для данного теста положительный вердикт только если целевой анализатор вернул на этом тесте значение **`false`**. На практике это обычно означает, что анализатор статической семантики завершил свою работу и при этом выдал сообщения о нарушениях контекстных условий.

6. Апробация

Метод SemaTESK был применен в следующих крупных проектах:

- тестирование обработчиков заголовков сообщений протокола IPMP-21 [13];
- тестирование анализатора языка C в компиляторе GCC;
- тестирование транслятора CTESK [6], разработанного в ИСП РАН;
- тестирование транслятора JavaTESK [15], разработанного в ИСП РАН;

Некоторые свойства целевых языков этих проектов представлены в следующей таблице:

Язык	Количество дескрипторов контекстных условий	Размер спецификации (строк)	Количество сгенерированных тестов
IPMP-21 XML	4	28	54
C	85	1019	около 10000
Java	278	3350	около 32000

Основной целью пилотного проекта по тестированию IPMP-21 [13] являлась демонстрация применимости метода SemaTESK к формализации статической семантики для генерации семантически корректных XML документов.

Целью пилотного проекта по тестированию GCC являлась демонстрация применимости метода SemaTESK к формализации статической семантики сложных языков программирования.

В проектах по тестированию трансляторов CTESK [6] и JavaTESK [15], несмотря на то, что к тому моменту уже было проведено довольно тщательное тестирование с помощью вручную созданных тестов, при тестировании методом SemaTESK в анализаторах семантики было найдено несколько ошибок.

6.1. Оценка эффективности метода

Опыт, полученный в проектах по тестированию с использованием метода SemaTESK, позволяет ориентировочно оценить выигрыш, который дает использование метода перед написанием тестов вручную. Один тест для анализатора статической семантики содержит примерно от 10 до 30 строк кода. Следовательно, для получения 10 тестов вручную требуется написать примерно 100—300 строк кода. С другой стороны, как видно из приведенной выше таблицы, при использовании метода SemaTESK для получения 10 тестов требуется написать в среднем примерно 1—2 строки кода. Таким образом, метод SemaTESK дает примерно стократный выигрыш в трудозатратах на разработку тестов.

6.2. Некоторые дополнительные замечания

Метод SemaTESK разрабатывался именно с целью организовать направленную генерацию тестовых данных для анализаторов статической семантики формальных языков. Таким образом, SRL-спецификация вряд ли может быть использована для генерации собственно анализаторов статической семантики, и тем более для генерации эффективных таких анализаторов.

В основном, метод SemaTESK применялся для тестирования анализаторов статической семантики языков программирования. Причем, в настоящее время метод все еще находится в состоянии развития: при каждом новом применении метода может потребоваться доработка как языка SRL, так и генератора STG. Тем не менее, мы уверены, что этот метод пригоден также

для тестирования анализаторов содержимого формальных документов, телекоммуникационных сообщений, запросов к базам данных и т.п.

Стоит отметить, что наиболее перспективной областью применения SemaTESK является генерация тестов для диалектов формальных языков, находящихся в стадии разработки. Это обусловлено в первую очередь тем, что для получения спецификации диалекта языка гораздо проще внести несколько, как правило, небольших, изменений в спецификацию исходного языка, чем создавать спецификацию диалекта с нуля. С другой стороны, на этапе разработки диалекта языка в его семантику довольно часто вносятся изменения, а при изменении контекстных условий намного легче оказывается модифицировать несколько дескрипторов в соответствующей SRL-спецификации, чем проводить ревизию всех вручную написанных тестов, которые затрагивают изменившиеся контекстные условия.

7. Заключение

В настоящей статье представлени метод SemaTESK, который реализует подход, основанный на использовании спецификаций, к тестированию анализаторов статической семантики. Метод дает возможность разрабатывать спецификации статической семантики на специализированном языке SRL в такой форме, которая, во-первых, обеспечивает хорошую прослеживаемость требований стандарта в формальной спецификации, а во-вторых, подходит для эффективной направленной автоматической генерации семантических тестов. Метод предоставляет соответствующий генератор тестов STG, который позволяет получать множества тестов, удовлетворяющих адекватным критериям полноты, формулируемым на основе SRL-спецификации целевого языка. Генератор STG принимает на вход SRL-спецификацию и автоматически генерирует как семантически корректные, так и семантически некорректные (с четко определенными видами некорректности) тесты.

Метод SemaTESK был применен в ряде проектов, включая тестирование анализаторов статической семантики таких сложных языков программирования, как C и Java. Полученные практические результаты подтверждают эффективность метода SemaTESK.

Литература

[1] М.В. Архипова. Автоматическая генерация тестов для семантических анализаторов трансляторов. // Диссертация на соискание ученой степени к.ф.-м.н., Москва (2006)

[2] А.В. Демаков. TreeDL: язык описания структуры деревьев. `http://treedl.sourceforge.net/treedl/treedl_ru.html`

[3] B. Beizer. Software Testing Techniques. Second edn. // van Nostrand Reinhold (1990)

[4] C. Boyapati, S. Khurshid, D. Marinov. Korat: Korat: Automated Testing Based on Java Predicates. // Proc. of International Symposium on Software Testing and Analysis (ISSTA). (2002)

[5] I. Bourdonov, A. Kossatchev, V. Kuliamin, A. Petrenko. UniTesK Test Suite Architecture. // LNCS 2391 (2002) 77–88
[6] CTESK – инструмент для тестирования программного обеспечения, реализованного на языке C. http://www.unitesk.ru/content/category/5/13/32/
[7] B. Daniel, D. Dig, K. Garcia, D. Marinov. Automated Testing of Refactoring Engines. // ESEC/FSE. (2007) 185–194
[8] A. Duncan, J. Hutchison. Using Attributed Grammars to Test Designs and Implementation. // Proceedings of the 5th international conference on Software engineering. (1981) 170–178
[9] R.F. Guilmette. TGGS: A Flexible System for Generating Efficient Test Case Generators // (1995)
[10] Y. Gurevich. Abstract State Machines: An Overview of the Project. // LNCS 2942 (2004) 6–13
[11] J. Harm. Automatic Test Program Generation from Formal Language Specifications. // Rostocker Informatik-Berishte 20 (1997) 33–56
[12] J. Harm, R. Lammel. Two-dimensional Approximation Coverage. // Informatica 24 (3) (2000)
[13] IPMP: Intellectual Property Management and Protection in MPEG Standards. http://www.chiariglione.org/mpeg/standards/ipmp/
[14] D. Jackson. Alloy: a lightweight object modelling notation. // ACM Trans. Softw. Eng. Methodol. 11 (2) (2002) 256–290
[15] JavaTESK – инструмент для тестирования программного обеспечения, реализованного на языке Java. http://www.unitesk.ru/content/category/5/25/60/
[16] A. Kalinov, A. Kossatchev, M. Posypkin, V. Shishkov. Using ASM Specification for Automatic Test Suite Generation for mpC Parallel Programming Language Compiler. // Proceedings of Fourth International Workshop on Action Semantic, AS'2002, BRICS note series NS-02-8. (2002) 99–109
[17] A. Kossatchev, P. Kutter, M. Posypkin. Automated Generation of Strictly Conforming Tests Based on Formal Specification of Dynamic Semantics of the Programming Language. // Programming and Computing Software 30 (4) (2004) 218 – 229
[18] S. Khurshid, D. Marinov. // TestEra: Specification-based testing of Java programs using SAT. // Automated Software Engineering Journal 11(4) (2004) 403–434
[19] J. Paakki. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. // ACM Computing Surveys 27 (2) (1995) 196–255
[20] A. Petrenko. Specification Based Testing: Towards Practice. // LNCS 2244 (2001) 287–300
[21] E.G. Sirer, B.N. Bershad. Using Production Grammars in Software Testing. // Second Conference on Domain-Specific Languages. (1999) 1–13
[22] UniTESK: индустриальная технология надежного тестирования. http://www.unitesk.ru/