Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST

С. В. Сыромятников
syrom@ispras.ru

Аннотация. Во многих случаях дефекты программного кода могут быть выявлены путём анализа соответствующего синтаксического дерева. В данной статье рассматриваются преимущества и недостатки данного подхода, в сравнении с более сложными видами статического анализа, и обосновывается необходимость предоставления пользователю интерфейса для написания собственных обнаружителей дефектов. Рассматриваются различные подходы к реализации подобного интерфейса. Описывается новый декларативный язык, позволяющий пользователю описывать дефекты кода, которые требуется обнаруживать, в виде шаблонов для синтаксических деревьев, и рассматриваются некоторые аспекты работы анализатора этого языка.

Ключевые слова: синтаксическое дерево; дефект кода; чекер; программа; код; декларативный интерфейс; язык; шаблон

1. Введение

На сегодняшний день существует достаточное количество средств, позволяющих анализировать исходный код программ и выявлять в нём те или иные дефекты. Как правило, анализу кода предшествует синтаксический разбор, в результате которого строится *синтаксическое дерево*, структура которого отражает синтаксическое строение исходного кода ([1]). Далее для некоторых узлов данного дерева вычисляются соответствующие им семантические характеристики.

Для поиска наиболее существенных дефектов кода необходимо выявлять возникающие в процессе его исполнения зависимости между структурными элементами. Это требует моделирования и анализа потоков команд и данных, что является существенно более сложной задачей по сравнению с анализом статических синтаксических структур ([2, 3]). Однако анализ синтаксических деревьев как таковых тоже оказывается полезным для выявления определённой категории дефектов (*СД-дефектов*). Во многих случаях дефекты такого рода суть отступления от некоторых принятых правил написания программ (*Coding Style*), которые регламентируют именование языковых сущностей, наличие или отсутствие скобок, использование тех или

иных системных функций и т. п. Примером типичного Coding Style СД-дефекта может служить следующий пример.

Пример 1. *Переменная неявно объявлена как целочисленная:*

```
const x;
```

Однако не следует полагать, что неверно расставленными скобками исчерпывается множество дефектов, потенциально находимых по синтаксическим деревьям:

Пример 2. *Слишком большие структуры данных передаются по значению в качестве аргументов функции.*

Интересен также:

Пример 3. *В декларации класса определяется конструктор копирования, но не определяется оператор присваивания*.

Характерной особенностью последнего примера является то, что данный дефект может быть обнаружен и при помощи анализаторов потоков данных, причём более точно, поскольку наличие конструктора копирования при отсутствии оператора присваивания критично практически только в тех случаях, когда класс содержит данные, копируемые нетривиальным образом, например, С-строки. Однако анализом дерева также можно выявить значительное количество подобных дефектов; в силу того, что такой анализ является гораздо менее ресурсоёмким по сравнению с анализом потока данных, во многих случаях его применение оказывается оправданным.

Нередко бывает так, что написать абсолютно универсальный обнаружитель того или иного дефекта, выявляющий все его потенциальные разновидности, не представляется возможным. Подкласс дефектов, достоверное выявление которых оказывается достаточным, часто определяется потребностями конкретного пользователя. Если же говорить о Coding Style, то своды подобных правил, которые имеются во многих крупных компаниях, занимающихся разработкой программного обеспечения, тем более являются специфичными именно для данных организаций. Поэтому представляется логичным, чтобы программные средства, предназначенные для выявления СД-дефектов, давали пользователю возможность самостоятельно описывать ошибки, которые он хочет искать, в терминах синтаксических деревьев.

Одним из вариантов решения этой задачи является предоставление пользователю некоего программного интерфейса (API) для доступа к структурам дерева, при этом выявители дефектов (*чекеры*, или, в данном случае, *СД-чекеры*) пишутся на соответствующем языке программирования (например, С) и требуют компиляции. Как показывает практика, при этом накладные расходы на написание чекера оказываются довольно велики, что в случае несложных дефектов является неудобным. Поэтому более привлекательным представляется другой подход: описывать поддеревья синтаксического дерева, соответствующие тем или иным дефектам, на

некотором специальном языке в виде *шаблонов*, что позволит пользователю во многих случаях обходиться без создания собственных скомпилированных библиотек. Подобный язык, получивший название *KAST*, был нами разработан на базе известного языка поиска на деревьях XPath ([4, 5]).

2. Синтаксические деревья: основные понятия

Понятие *дерева абстрактного синтаксиса* (*ДАС*) определяется, например, в работе [1]. Напомним, что ДАС — это древовидная структура данных, отражающая синтаксическую структуру программного кода. *Узлы* ДАС соответствуют синтаксически завершённым элементам кода - целочисленным константам, бинарным выражениям, условным операторам, определениям функций и так далее. Поскольку более сложные синтаксические конструкции включают в себя более примитивные, у узла ДАС могут быть ссылки на другие узлы, называемые его *дочерними узлами*; так, узел, соответствующий бинарному выражению, имеет ссылки на узлы-операнды.

Именно наличие дочерних узлов делает структуру ДАС истинно древовидной (а не линейной); корень соответствующего дерева обыкновенно соответствует максимальной единице синтаксического анализа (то есть одной единице компиляции). Следует различать *простые* (условие в условном операторе) и *списочные* (набор параметров функции) дочерние ссылки. Последние представляют собой однородный массив, вообще говоря, произвольной длины.

Однако ДАС, будучи хорошим объектом для теоретических исследований, несколько отличается от того, что обыкновенно получается в результате работы синтаксических анализаторов (вполне конкретных синтаксических деревьев). Так, с каждым узлом синтаксического дерева связан ряд характеристик, описывающих свойства соответствующей языковой конструкции (*атрибутов*), например, значения литералов или конкретные типы выражений. Кроме того, после построения дерева на нём осуществляется работа семантического анализатора, в результате чего некоторые узлы оказываются дополнительно снабжены *семантической информацией*. Так, выражение - функциональный вызов дополняется ссылкой на информацию о вызываемой функции. Практика показывает, что нередко использование семантической информации в СД-чекерах делает их более эффективными по скорости и расходу памяти, однако ещё чаще оказывается, что создание тех или иных чекеров без использования семантической информации оказывается попросту невозможным.

Стоит также отметить, что часто разные типы узлов синтаксического дерева обладают сходными свойствами и сходным набором атрибутов и могут использоваться в одних и тех же синтаксических позициях. К примеру, узлы, соответствующие бинарным выражениям (скажем, сложению) имеют две дочерних ссылки — на левый и правый операнд. Эти операнды могут быть выражениями достаточно произвольного вида — идентификаторами, константами, скобочными выражениями и так далее, то есть представляться

узлами, вообще говоря, различных типов. Сказанное означает, что систему типов узлов дерева целесообразно организовывать по иерархическому принципу, и, таким образом, для них вводятся понятия *подтипа* и *супертипа*.

2.1. Существующие системы

Существует достаточно много систем, осуществляющих, в том числе, и поиск дефектов кода по синтаксическим деревьям, как свободно распространяемых - Checkstyle, RATS, Flawfinder и так далее, так и коммерческих - AppPerfect, Jstyle, Coverity и других. ([6-12]). Приведём несколько примеров дефектов, потенциально находимых СД-анализом и обнаруживаемых данными системами (все примеры, кроме последнего, касаются языка java):

- Пустые catch-блоки.
- Лишние конструкторы (конструктор в классе единственный, публичный, без аргументов и без тела).
- Использование оператора 'continue'.
- Использование оператора 'new' для строк.
- Использование потенциально небезопасных функций ('gets' вместо 'fgets').

Большинство этих программных систем не имеет возможностей расширения; небольшое их число предоставляют пользователю API для написания чекеров, а из систем, реализующих декларативный язык описания дефектов, можно упомянуть только PMD и некоторые системы, её интегрирующие (Coverity).

Система PMD ([11]), ориентированная исключительно на язык java, предоставляет пользователю как API для написания чекеров в виде классов java, так и возможность создания шаблонов дефектов на языке Xpath. Рассмотрим следующий пример кода на java:

```
class Foo {
}
class Example {
  void bar(Foo x, Foo y) {
    if (y == null)
      y = x;
    else
      return;
  }
}
```

и попытаемся создавать Xpath-шаблоны, соответствующие тем или иным фрагментам данного кода.

Пример 4. *Найти все параметры методов, имеющие тип 'Foo':*

```
// MethodDeclaration / MethodDeclarator / FormalParameters /
FormalParameter / Type / ReferenceType / ClassOrInterfaceType
[@Image='Foo']
```

Здесь уже видны некоторые недостатки подхода PMD:

- не различаются типы узлов и имена дочерних ссылок (что может провоцировать пользователя на синтаксические ошибки при написании шаблонов);
- отсутствует возможность адресации конкретного члена для списочной дочерней ссылки. Приведённому шаблону, вообще говоря, соответствуют оба параметра метода из примера. Если нас интересует только первый параметр, нам поможет такой шаблон:

```
// MethodDeclaration / MethodDeclarator / FormalParameters /
FormalParameter [ following-sibling::FormalParameter ] / Type /
ReferenceType / ClassOrInterfaceType [@Image='Foo'],
```

но подобное решение никак не может быть признано универсальным.

Кроме того, у пользователя практически отсутствует доступ к семантической информации, что делает реализацию ряда чекеров неэффективной, а многих — просто невозможной. Так, Xpath-интерфейс PMD не позволяет отслеживать вызовы статических методов класса.

Стоит также отметить, что иногда неудобство использования описываемого интерфейса PMD обусловлено недостатками в структуре используемой реализации синтаксического дерева. Так, нижеприведённому шаблону будут соответствовать обе ветви условного оператора — then и else:

```
// IfStatement / Statement
```

3. Интерфейсы описания дефектов

В ряде случаев программная реализация дерева не позволяет прикладным программам удобно работать с ним напрямую при поиске дефектов. Это означает, что для доступа к его структурам может потребоваться более или менее сложный программный интерфейс (API); будучи хорошо спроектированным, такой интерфейс обладает очень большой выразительной мощностью. Схема использования API, так или иначе, остаётся следующей. Используя функции или классы интерфейса, пользователь пишет программный код, выявляющий интересующие его дефекты в синтаксическом дереве (СД-чекер). Это код компилируется в модуль известного формата, например, динамически подгружаемую библиотеку. Впоследствии специальный анализатор, работающий после стадий синтаксического и семантического анализа, подгружает нужные модули и запускает требуемые функции (методы).

Таким образом, объём работ по написанию даже примитивных чекеров оказывается довольно велик, и в случае простых дефектов подобная технология оказывается не очень удобной. Более привлекательным в этом плане представляется другой подход: описывать поддеревья синтаксического дерева, соответствующие тем или иным дефектам, на некотором языке в виде шаблонов. В отличие от описания алгоритмов поиска дефектов, шаблоны могут быть помещены непосредственно в некоторый конфигурационный файл, который не требует компиляции и прочитывается анализатором при каждом запуске. Это приводит к дополнительному усложнению анализатора, но позволяет пользователю во многих случаях обходиться без сложной процедуры создания собственных программных модулей. Недостатком подобного подхода является потенциально меньшая по сравнению с API выразительная мощность, однако, как показывает практика, эта проблема может быть решена, если язык шаблонов будет расширяем.

Синтаксис такого языка шаблонов может быть достаточно произвольным; так, разработанный нами язык KAST основан на языке поиска на деревьях XPath.

4. Описание языка KAST

Основой для языка описания шаблонов был выбран язык XPath ([4,5]). Первоначально XPath был разработан как язык поиска заданных шаблонов в xml-документах, а xml-документы, как известно, представимы в виде деревьев. Типичный запрос на языке XPath, предназначенный для поиска заданных поддеревьев во всём xml-документе (с любым расположением относительно корня соответствующего дерева), имеет следующий вид (#):

*// Node1 {**[conditions1]**} / {axis2::}Node2 {**[conditions2]**} /*
{axis3::}Node2 {[conditions2]} ...

Здесь элементы языкового синтаксиса выделены **полужирным** начертанием, имена метапеременных – *курсивом*, а необязательные элементы заключены в фигурные скобки *{}*.

Рассмотрим для примера следующий Xpath-шаблон:

// A [@id = 2] / parent::B

Если запрос с данным шаблоном применить к приведённому ниже xml-документу, будут найдены узлы, выделенные полужирным шрифтом:

```
<B>
 <A id="2"/>
 <C>
  <A id="2"/>
  <B>
   <A id="2"/>
   <A name="foo"/>
  </B>
```

</C>
</B>

Однако непосредственное использование XPath для поиска шаблонов в синтаксических деревьях оказалось не вполне удобным, главным образом потому, что этот язык не предоставляет возможности указывать имена дочерних ссылок узла в рамках шаблона. Смешивание имён дочерних ссылок с именами типов узлов, как это сделано в PMD, существенно затруднит практическую реализацию, заведомо ограничит выразительную мощность языка и, по-видимому, приведёт к излишней путанице среди пользователей. Кроме того, далеко не все стандартные *оси* (спецификаторы, обозначенные в (#) как *axisi*) и встроенные функции XPath представляются актуальными для целей поиска по деревьям рассматриваемого типа.

Поэтому стандартный XPath был модифицирован. В частности, имена дочерних ссылок стали, по сути, разновидностью осей: $Node_1$ / *ChildName*::$Node_2$; была добавлена поддержка переменных, увеличившая выразительную мощность языка, и произведён ряд других модификаций, преимущественно синтаксического характера.

Шаблоны поддеревьев синтаксического дерева, написанные на KAST'e, имеют следующий вид (##):

// Tun1 [***Спецификация1,1*** *] ... [* ***Спецификация1,k1*** *] /*
Квалификатор1 :: Tun2 [***Спецификация2,1*** *] ... [* ***Спецификация2,k2*** *] ...*
Tunn [***Спецификацияn,1*** *] ... [* ***Спецификацияn,kn*** *]*

Наличие в начале шаблона двух символов '//' обязательно: оно означает, что поиск данного шаблона будет происходить в дереве повсеместно, вне зависимости от положения сопоставляемых узлов относительно корня. Tun_i – имя одного из типов узлов или звёздочка ('*'), соответствующая произвольному имени типа. *Квалификаторы* – это имена дочерних ссылок узла или некоторые предопределённые оси. *Спецификации* бывают двух видов: присваивания (в которых определяются переменные) и ограничения (суживающие множество подходящих узлов). Данный шаблон соответствует некоторому поддереву синтаксического дерева, если:

1. Каждому Tun_i из шаблона соответствует узел дерева, тип которого есть Tun_i или является подтипом Tun_i ($1 \le i \le n$).
2. Для всякого узла, сопоставленного некому Tun_i, выполнены все соответствующие ограничения $Спецификация_{i,j}$ ($1 \le i \le n$, $1 \le j \le ki$).
3. Для всякой пары (Tun_i, Tun_{i+1}) соответствующие узлы (N_i, N_{i+1}) должны быть связаны согласно квалификатору $Квалификатор_i$. $Квалификатор_i$ может быть именем дочерней ссылки; в этом случае

вершина N_i должна ссылаться на вершину N_{i+1} именно данной дочерней ссылкой. Кроме того, в качестве квалификатора могут использоваться следующие *оси*:

- *parent.* Запись "Tun_1 / ***parent*** :: Tun_2" означает, что вершина с типом Tun_2 является родительской по отношению к вершине с типом Tun_1.
- *ancestor.* Запись "Tun_1 / ***ancestor*** :: Tun_2" означает, что вершина с типом Tun_2 является предком (необязательно непосредственным) по отношению к вершине с типом Tun_1.
- *descendant.* Запись "Tun_1 / ***descendant*** :: Tun_2" означает, что вершина с типом Tun_2 является потомком по отношению к вершине с типом Tun_1.
- *following-sibling.* Запись "Тип1 / **following-sibling** :: Тип2" означает, что вершина с типом Тип2 является следующей по отношению к вершине с типом Тип1 в некотором однородном списке вершин (например, соответствующем списку параметров функции).

Ограничения могут содержать числовые значения и операции с ними, логические операции, строковые выражения и *подшаблоны*. Подшаблоны в ограничениях соответствуют формату (##) за тем исключением, что начинаются сразу с квалификатора. Такое изменение формата обусловлено тем, что сопоставление подшаблонов происходит не "где-то в дереве", а начиная непосредственно с текущего узла. Кроме того, в ограничениях могут использоваться вызовы функций, как предопределённых, так и определяемых пользователем. *Присваивания* служат одновременно и объявлениями *переменных*; переменной может присваиваться любое выражение, удовлетворяющее условиям на синтаксис ограничений. Переменная видна всюду в пределах данного шаблона после своего определения и может использоваться в ограничениях и других присваиваниях наравне с другими выражениями.

Поиск шаблона (##) в синтаксическом дереве происходит согласно нижеследующему алгоритму:

1. В дереве ищутся узлы, имеющие тип Tun_1 или тип которых является подтипом типа Tun_1. Найденный такой узел полагается *текущим*.
2. Тип текущего узла сравнивается с Tun_i (если это не было сделано раньше). Если они совпадают или Tun_i является супертипом для типа текущего узла, данный шаг признаётся успешным и происходит переход к пункту 3.

3. Происходит вычисление значений всех переменных и всех ограничений для текущего узла в порядке их следования в шаблоне. В частности, ограничение-подшаблон считается истинным, если его сопоставление прошло успешно. Если хотя бы одно из ограничений не выполняется, сопоставление признаётся неуспешным. Если все ограничения истинны и $Тип_i$ – последний в цепочке, исходное сопоставление считается успешным. Если же ограничения истинны, но данный тип не последний, происходит переход к следующему узлу согласно пункту 4.
4. Следующий текущий узел определяется согласно квалификатору $Квалификатор_i$. Если это имя дочерней ссылки, происходит переход по соответствующей ссылке. Если это какая-то из осей, происходит переход к одной из вершин, соответствующих данной оси. Таковые вершины будут перебираться до тех пор, пока это возможно или пока не будет успешно сопоставлена оставшаяся часть шаблона. Если текущий узел найден, сопоставление продолжается согласно пункту 2, иначе оно считается неуспешным.

Следует отметить, что при нахождении удовлетворяющего шаблону узла поиск не прекращается и ищутся остальные аналогичные дефекты в данном исходном коде.

Чтобы не ограничивать пользователя жёсткими рамками декларативного шаблона, была реализована возможность определения пользовательских функций, которые пишутся на процедурном API для доступа к синтаксическим деревьям. Это делает выразительную мощность нашего декларативного языка сопоставимой с выразительной мощностью API.

5. Реализация

Анализ существующих систем с открытым исходным кодом, поддерживающих **XPath** ([13, 14]), показывает, что обычно в них используется следующий алгоритм поиска на деревьях:

1. xml-документ рассматривается как множество узлов, каждый из которых имеет не более одной ссылки на отца и некоторое количество ссылок на дочерние узлы. В процессе работы анализатор оперирует неким подмножеством этого множества, которое мы будем называть *активным множеством*. На начальном этапе активное множество полагается равным всему множеству узлов документа.
2. Когда анализатору передаётся запрос, он вычисляет подмножество активного множества, которое соответствует первому элементу запроса. После этого от запроса отрезается его первый элемент.

3. Если вычисленное подмножество непусто и оставшаяся часть запроса также непуста, активное множество полагается равным найденному подмножеству и вычисление повторяется согласно пункту 2.
4. Если вычисленное подмножество пусто, делается вывод, что узлов, соответствующих данному запросу, в документе нет.
5. Если вычисленное подмножество непусто, а оставшаяся часть запроса пуста, найденное подмножество и есть результат применения данного запроса к данному документу.

Кроме того, ввиду ограниченного объёма xml-документов для реализации анализаторов XPath нередко используются функциональные языки программирования ([15]). Размеры же синтаксических деревьев нередко бывают чрезвычайно велики (порядка миллионов узлов), а число шаблонов поиска может достигать сотен. Это делает описанные выше манипуляции с множествами вершин неэффективными, а применение рассмотренного алгоритма и функциональных языков для решения нашей задачи, как следствие, - неудобным и нецелесообразным. Более логичным представляется другой подход, основанный на планомерном обходе дерева сверху вниз и слева направо:

1. Обход начинается с корня дерева, каковым обычно является узел, представляющий единицу трансляции (исходный файл, с включёнными заголовочными файлами в случае C/C++).
2. В каждой вершине происходит попытка применить все правила, для которых данный тип вершин является стартовым. Если сопоставление правила прошло успешно, этот факт фиксируется.
3. Аналогично обходятся все дочерние ссылки данной вершины.

Если обозначить размер дерева через A, а количество шаблонов через p, то, очевидно, сложность такого алгоритма можно будет оценить снизу величиной $O(Ap)$. Данная оценка будет достигаться, если ни для какого из шаблонов не нашлось вершин, тип которых являлся бы для этого шаблона стартовым. Если же вершина с нужным типом в дереве всё же нашлась и реальное сопоставление шаблона с поддеревом началось, следует учитывать свойства самого шаблона, которые мы назовём его *сложностью*.

На первый взгляд кажется, что сложность определяется главным образом максимальным количеством вершин, которые придётся обойти при сопоставлении с данным шаблоном. Однако опыт написания реальных шаблонов показывает, что многие из них ограничены весьма небольшими поддеревьями (не более 4-5 вершин), поэтому наибольший вклад в величину сложности вносят как раз условия, не требующие обхода поддерева. Так или иначе, сложность шаблона, очевидно, может быть оценена количественно, и поэтому, имея некоторый их набор, мы можем определить максимальную сложность шаблона l. Тогда сложность нашего алгоритма можно будет оценить сверху величиной $O(Apl)$.

Стоит, однако, отметить, что сложность шаблонов практически может быть оценена сверху некоторым константным значением, не зависящим от их конкретного набора. СД-дефекты обычно таковы, что шаблоны для их описания требуются сравнительно небольшие. Дефект, содержащий в своём описании очень большое количество ограничений, перестаёт быть некоторой универсальной категорией и становится экзотическим случаем. Соответствующий чекер с большой вероятностью будет исключительно труден для понимания и сопровождения. Поэтому для большинства случаев верхнюю оценку сложности нашего алгоритма можно также записать как *O(Ap)*.

6. Пример работы

Проиллюстрируем использование языка KAST следующим примером (для языка C++):

Найти все публичные члены класса, имеющие тип int.

Сама по себе эта задача, по-видимому, не имеет большого практического значения, но может являться подзадачей при обнаружении некоторого действительно значимого дефекта. Мы будем пользоваться программным инструментом под названием *Checker Studio*, входящим в состав продуктов компании Klocwork. Данная программа графически изображает синтаксическое дерево для заданного фрагмента исходного кода и подсвечивает фрагменты кода и узлы дерева, соответствующие заданному KAST-чекеру. Код, на котором мы будем искать наш дефект, выглядит так:

```
class Zoo {
    private:
    int deers;
};
class Forest {
    public:
    int bears;
};
```

Итак, поиск членов класса начнём с поиска самих определений классов:

```
// ClassType
```

Этому шаблону, естественно, соответствуют оба определения класса из нашего примера (рис. 1):

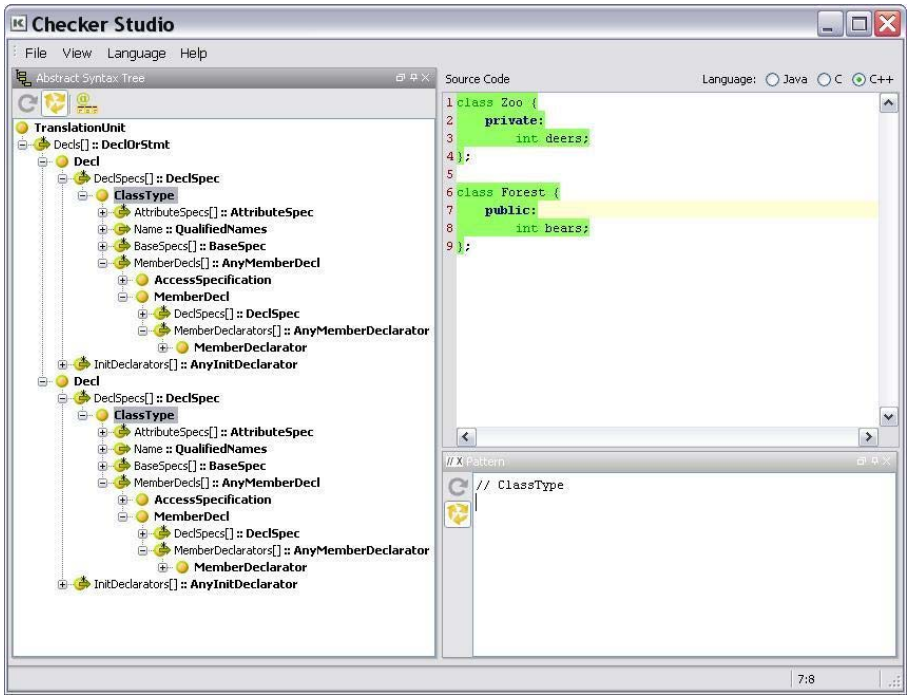

Рис. 1.

Но нас интересуют не сами классы, а их члены-переменные:

```
// ClassType / MemberDecls[*]::MemberDecl
```

Звёздочка означает, что данный член может находиться на произвольном месте в списке членов класса. Если бы нам было важно, чтобы он был, например, третьим, мы бы вместо звёздочки указали число 2 (нумерация элементов списка начинается с 0).

Для данного шаблона также нашлось два соответствия (рис. 2):

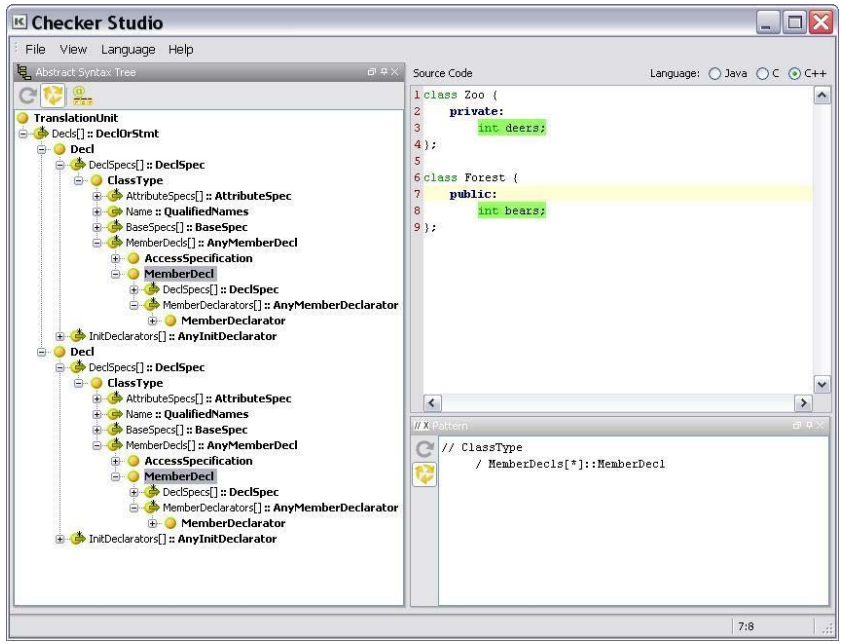

Рис. 2.

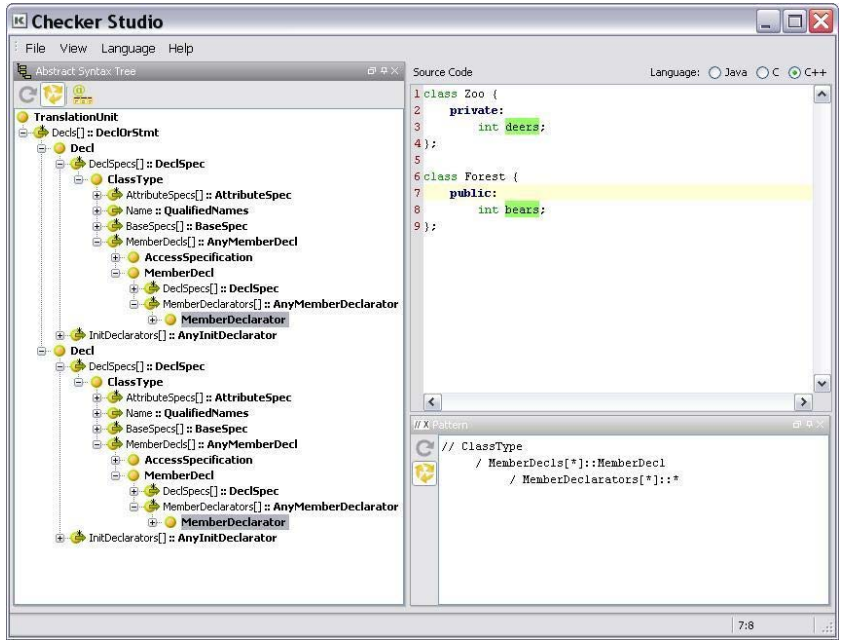

Рис. 3.

Далее мы перейдём непосредственно к декларации конкретного члена:

*// ClassType / MemberDecls[*]::MemberDecl / MemberDeclarators[*]::**

Как видно, для нас не важна не только позиция данного члена в данной декларации (а их там может быть несколько, например *int x, *y;*), но и тип соответствующего узла дерева. В данном случае отсутствие проверки типа скомпенсируется ограничениями, которые будут наложены на данный узел далее. Результат поиска по данному шаблону выглядит так (рис. 3):

Осталось среди найденных членов класса выбрать искомые. Во-первых, они должны иметь тип int:

// ClassType / MemberDecls[*]::MemberDecl / MemberDeclarators[*]:: [getTypeName() = 'int']*

Этому ограничению по-прежнему удовлетворяют два члена класса и два узла дерева (рис. 4):

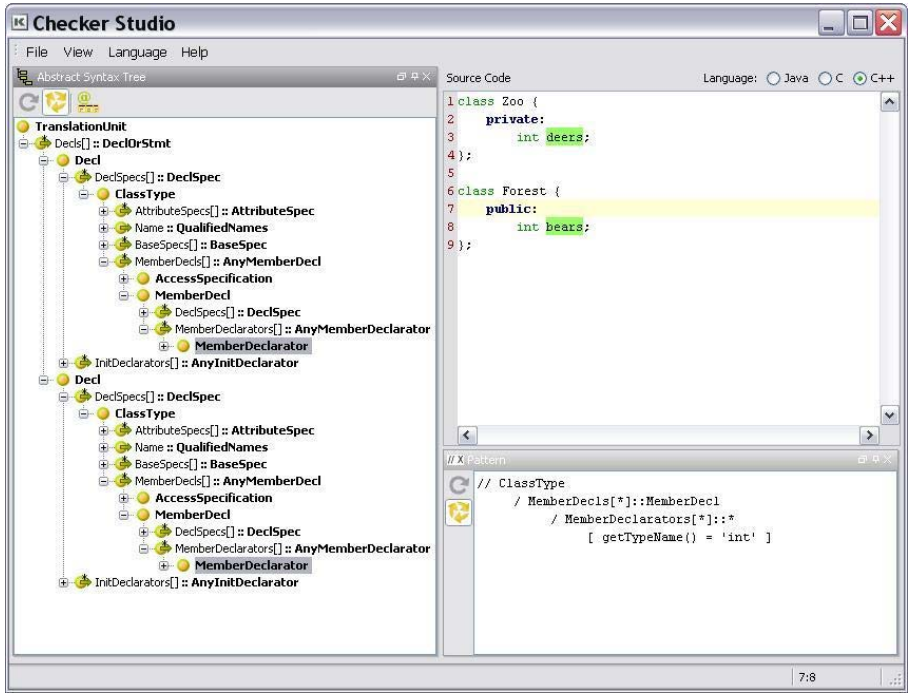

Рис. 4.

Во-вторых, они должны быть публичны:

// ClassType / MemberDecls[*]::MemberDecl / MemberDeclarators[*]:: [getTypeName() = 'int'] [isPublic()]*

Это ограничение позволяет отсеять один из первоначально найденных членов (рис. 5):

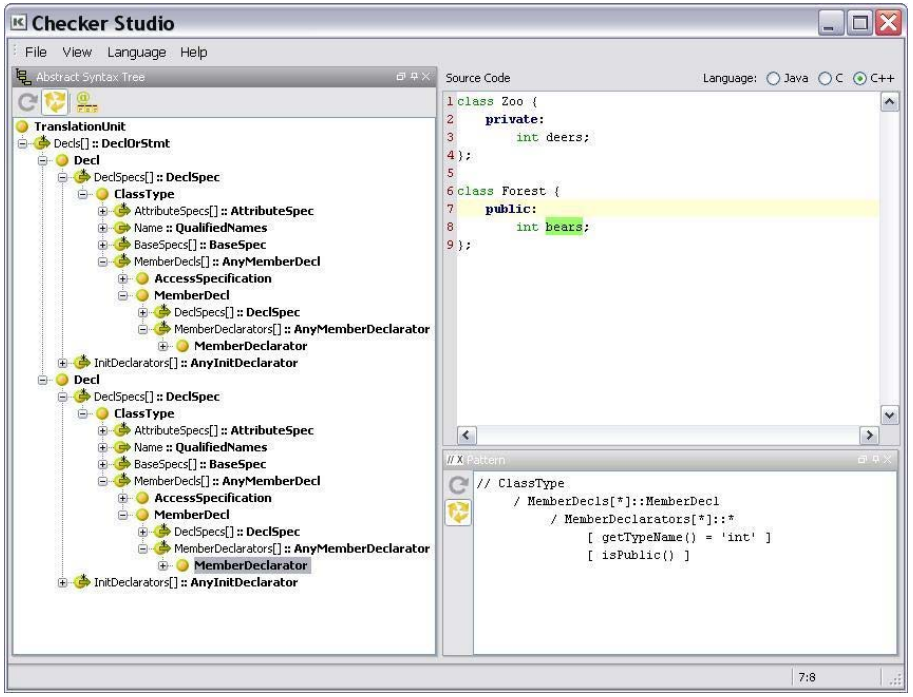

Рис. 5.

Подобно тому, как мы последовательно строили наш шаблон, работает и сам сопоставитель шаблонов. Для данного шаблона сначала ищутся определения классов (рис. 1); найдя такое определение, сопоставитель проверяет, выполняются ли наложенные на это определение ограничения (в данном случае их нет) и в соответствии со спецификацией (/ *MemberDecls[*]::*) переходит к следующему узлу (рис. 2). Аналогично, после проверки ограничений второго узла, происходит переход к третьему узлу (рис. 3) и проверяются его ограничения (рис. 4, 5).

7. Результаты

В настоящее время поддержка KAST реализована (с некоторым достаточным набором встроенных функций) для языков программирования C/C++ и Java; в значительной степени также поддержан язык C#. В частности, на первоначальном этапе для языков C и C++ в среде Unix было написано порядка 25 чекеров, как простых, ориентированных преимущественно на выявление несущественных дефектов кода, так и более сложных. При помощи этих чекеров были проанализированы исходные коды проектов

Mozilla Firefox 3.5, Apache http server (httpd) 2.2.14 и postgresql 8.4.2. Результаты анализа представлены в таблице 1.

	Всего потенциальных дефектов	**Из них реальных**	**Доля реальных дефектов**
Firefox	393	226	58%
httpd	7	3	43%
postgresql	33	21	64%
Всего	433	250	58%

Табл. 1. Результаты работы KAST-чекеров на некоторых проектах.

При этом нужно отдавать себе отчёт, что процент ложных срабатываний для простых и (часто, но не всегда) не очень важных дефектов был заметно ниже, чем для более сложных. Также следует отметить, что для 50% всех имеющихся чекеров не было отмечено ни одного срабатывания, истинного либо ложного, что объясняется главным образом спецификой анализируемых проектов.

При общем времени работы программы над данным проектом порядка 50 минут (оно включает в себя, главным образом, время, затрачиваемое на построение синтаксических деревьев, работу синтаксического анализатора и работу не-KAST чекеров) время работы собственно KAST-анализатора составило всего несколько минут.

8. Заключение

Разработанный нами язык оказался удобным средством описания несложных дефектов кода, находимых по синтаксическому дереву. Он широко используется в компании Klocwork при написании поставляемых конечным клиентам чекеров. Более того, практика показывает, что и сами клиенты активно пользуются предоставленной возможностью по разработке собственных обнаружителей дефектов.

Основная возможность для усовершенствования нашего языка шаблонов — это добавление новых встроенных функций, при этом исключительно полезным оказывается анализ отзывов и пожеланий конечных пользователей, которые в процессе анализа собственного кода сталкиваются с отсутствием той или иной функциональности в языке KAST. Кроме того, ввиду

относительной новизны описанной технологии и, как следствие, отсутствия некоего канонического варианта синтаксиса языка, возможно добавление новых осей. Не все оси, имеющиеся в стандартном Xpath, на данный момент поддерживаются в KAST, и возможно, наличие некоторых из них было бы полезно при поиске дефектов в синтаксических деревьях.

Перспективы расширения KAST за счёт поддержки новых языков программирования представляются достаточно туманными, поскольку поддержка для основных коммерчески используемых процедурных языков (C/C++, Java, C#) уже реализована.

Приложение.

Примеры чекеров на KAST

Приведём в качестве примеров чекеры, написанные на KAST и предназначенные для поиска дефектов, описанных во введении.

Пример 1'. Переменная неявно объявлена как целочисленная

```
// Decl [ not InitDeclarators::Null ]
        [ not DeclSpecs[*]::AnyTypeOf ]
        [ not DeclSpecs[*]::ReservedTypeSpec ]
        [ not DeclSpecs[*]::ClassType ]
        [ not DeclSpecs[*]::EnumType ]
        [ not DeclSpecs[*]::TypeName ]
```

Пример 2'. Слишком большие структуры данных передаются по значению в качестве аргументов функции.

```
// FuncDeclarator  [ ancestor::FuncDef ]
                  [ $1 := getName() ]
                 / Params[*]::Decl [ InitDeclarators[*]::InitDeclarator
                                   [ $type := getInitialDefinedType() ]
                                     [ not $type.isArray() ]
                                     [ not $type.isPointer() ]
                                     [ not $type.isReference() ]
                                     [ $type.getTypeSize() > 128 ] ]
```

Пример 3'. В декларации класса определяется конструктор копирования, но не определяется оператор присваивания.


```
// ClassType
    [ MemberDecls[*]::MemberDecl
        / MemberDeclarators[*]::MemberDeclarator
                                [ isCopyConstructor() ]
    | MemberDecls[*]::MemberFunc / FuncDef::FuncDef
        / Declarator::Declarator [ isCopyConstructor() ] ]
    [ not MemberDecls[*]::MemberDecl
        / MemberDeclarators[*]::MemberDeclarator
                                [ isAssignmentOperator() ] ]
    [ not MemberDecls[*]::MemberFunc / FuncDef::FuncDef
        / Declarator::Declarator [ isAssignmentOperator() ] ]
```

Литература

[1] Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Т. 1. Синтаксический анализ. М., 1978. 613 с.

[2] S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997. 888 с.

[3] A. Belevantsev, O. Malikov. Using data flow analysis for detecting security vulnerabilities. Сборник трудов Института системного программирования РАН. Под ред. чл.-корр. РАН Иванникова В.П. Т. 11. М., ИСП РАН, 2006. 128 с., с. 83-98.

[4] http://www.zvon.org/xxl/XPathTutorial/General/examples.html

[5] http://www.w3schools.com/XPath/default.asp

[6] http://www.mmsindia.com/jstyle.html

[7] http://www.fortify.com/security-resources/rats.jsp

[8] http://www.dwheeler.com/flawfinder

[9] http://www.coverity.com

[10] http://www.appperfect.com/products/java-code-test.html

[11] http://pmd.sourceforge.net

[12] http://checkstyle.sourceforge.net

[13] http://tinyxpath.sourceforge.net

[14] http://xerces.apache.org

[15] Д. А. Лизоркин. Оптимизация вычисления обратных осей языка XML Path при его реализации функциональными методами. Сборник трудов Института системного программирования РАН. Под ред. чл.-корр. РАН Иванникова В.П. Т. 8, ч. 2. М., ИСП РАН, 2004. 214 с., с. 93-119.

[16] R. Cole, R Hariharan, P. Indyk. Tree pattern matching and subset matching in deterministic O(n log3m) time. Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, MD, 1999, pp. 245-254.