JAVACHECKER: СТАТИЧЕСКИЙ АНАЛИЗ ПРОГРАМНЫХ КОМПЛЕКСОВ С ПОМОЩЬЮ ПЕРЕПИСЫВАЮЩИХ ПРАВИЛ.

Руслан Шевченко
rssh@gradsoft.kiev.ua

Институт Програмных Систем НАНУ.
Просп. Глушкова 40, корп. 5, Киев 03187, Украина

У роботі описано один із підходів до аналізу програмних комплексів, що грунтується на методех символьної обробки інформації. Особливістю підходу є застосування формального аналізу не до обчислювальної семантики програмних систем, а до окремих характеристик. Це дозволяє уникнути складнощів повної формалізації задачі, і у той-же час дозволяє проводити прагматично корисні висновки щодо характеристик програмної системи (такі, як наявнісьт помилок визначенного классу) за допомогою порівняльно невеликого об'єму обчислень. Описується статичний аналізатор програм на мові Java (JavaCheker), що реалізує даний підхід і побудован у середовищі алгебраїчного програмування TermWare.

This work describei an approach to software system analysis, based on the methods of symbolic comutations. The approach is characterized by using light-weight formal model based not on full-fledged computation semantics, but on particular properties of underlying software system. Such approach eliminate need in full formal analysis of software system and allows to receive some pragmatically usefull results about software (such as existence of some types of errors) without complex comutations. Static analyzer for Java source code (JavaChecker) build on top of TermWare framework is described.

1. Введение

В этой статье описывается один из подходов к анализу программных комплексов, основанный на методах символьной обработки.

Одной из особенностей этого подхода является применение формального анализа не к вычислительной семантике программных систем, а к отдельным характеристикам, не связанным напрямую с формальной семантикой. Это позволяет избежать трудностей полной формализации задачи, и в то-же время позволяет производить прагматически полезные выводы о свойствах программной системы (такие как наличие определенного класса ошибок), с помощью сравнительного небольшого объема вычислений.

Описывается статический анализатор программ для Java, (JavaChecker),
реализирующий данный подход и построенный в среде алгебраического программирования TermWare, которая разрабатывается в ООО "Град-Софт",

2. TermWare: Основные сведения

TermWare представляет собой систему переписывающих правил, основанной на формализме логики со взаимодействиями: основные объекты с которыми мы имеем дело это термальные системы, состоящие из БД фактов, набора переписывающих правил и стратегии их применения.

Основное отличие TermWare от традиционных систем переписывания в том, что в окружение переписывающих правил внедрены операции взаимодействия со базой фактов (которая представляет для системы внешнюю среду).

Правило переписывания это четверка *(x, ein, y, eout)*, где *x* и *y* - это входной и выходной терм, соответственно, как и в традиционных системах, а *ein* и *eout* - операции ввода и вывода информации из БД фактов. *ein* и *eout* часто называют условием и действием, соответственно.

Термальная система описывается выражением типа

- *system(name, facts, ruleset, strategy),* где:
- *name* - имя системы.
- *facts* - БД фактов, в которую инкапсулирована внешняя среда. С помощью специального терма *javaFacts* мы можем провести соответствие между Java-классом и БД фактов -- условия и действия будут редуцироваться в вызовы методов данного класса.
- *ruleset* - наборы правил, описанный выше. Также наборы правил могут включать в себя спецификации правил из других именованных систем.

- *strategy* - стратегия применения набора правил. Понятие стратегии совпадает с принятым в [7], [4].

Семантика TermWare подробно описана в [8]. Для более легкого введения см. Также [1], [3].Сама система оформлена как библиотека классов, предоставляющая открытое Java API: конечный пользователь может определять стратегию переписывания, интерфейс процедурных действий и запроса окружения на языке Java, использующая TermWare как встраиваемый модуль; с другой стороны внутренний язык системы позволяет непосредственной обращение к Java объектам из оператора взаимодействия с окружением, используя API рефлексии.

Для применения к задачам программной инженерии TermWare предоставляет интерфейсы, позволяющие ее интеграцию с синтаксическим анализаторами следующим образом: дерево разбора представляется в виде терма, который потом можно преобразовывать с помощью обычных логических средств.

Подробнее:

Пусть *L* - формальный язык, алфавит которого не пересекается с логическим алфавитом **TermWare**. В таком случае можно представить дерево разбора как логический терм:

1. Каждому нетерминалу соответствует выделенный атом *li*.
2. Каждому выражению, построенному из правила порождения *r : t -> t1 . . . tn* соответствует терм *Rt(Rt1 . . .Rtn)*

Таким образом мы построили некоторый симбиоз **AST** деревьев и алгебраических термов. Что это нам дает - можно выполнять точно-определенные операции преобразования на довольно богатой алгебре термов, представляя их в декларативном виде переписывающих правил.

К примеру, следующее выражение на языке Java:

```
class X
{
    int x() { return 1; }
}
```

В виде терма TermWare может быть представлено следующим образом:

```
java_class_declaration(
        java_identifier("X"),
        java_empty_type_properties,
        java_name([java_identifier("java"),
                java_identifier("lang"),
                java_identifier("Object")]),
                empty_list,
                [java_method_declaration(java_identifier("x"),empty_list,java_int,empty_list,empty_list,
                [java_return(java_integer_literal("1"))]])
         )
)
```

Декларативные правила дают нам некоторое преимущество в наглядности, по сравнению с системами статического анализа, основанными на паттернах обхода синтаксического дерева.

К примеру, сравним правило, которое находит в исходном тексте программы условные предложения с пустой основной ветвью в PMD [2]:

```
// Extend AbstractRule to enable the Visitor pattern
public class EmptyIfStmtRule extends AbstractRule implements Rule {

    // This method gets called when there's a Block in the source code
    public Object visit(ASTBlock node, Object data){

        // If the parent node is an If statement and there isn't anything
        // inside the block
        if ((node.jjtGetParent().jjtGetParent() instanceof ASTIfStatement) && node.jjtGetNumChildren()==0)
{
                // then there's a problem, so add a RuleViolation to the Report
                RuleContext ctx = (RuleContext)data;
                ctx.getReport().addRuleViolation(createRuleViolation(ctx,
                node.getBeginLine()));
        }

        // Now move on to the next node in the tree
```

```
        return super.visit(node, data);
    }
}
```

И в TermWare:

```
java_if($x1,java_empty_block,$x2) -> PROBLEM
                                        // emptyIfStatementDiscovered($x1)
```

Более того, как мы увидим позже, представление программы как логического терма позволяет нам производить логический анализ кода, основанный на частичной модели программы.

3. Типология программных погрешностей

Исторически, начиная с Lint [6] утилиты статического анализа основывались на определении статических образцов в коде, которые указывают на дефекты кодирования либо проектирования.
Большинство существующих анализаторов [2], [5] используют сопоставление с образцами как основную технику обнаружения дефектов. С точки зрения переписывающих правил, сопоставление с образцом это просто унификация определенного терма со свободными переменными с каким-либо субтермом программы.

В индустрии Java-программирования уже выделились основные типы ошибок, определяемые подобными методами:

3.1 Обработка исключений

Пустой блок catch. Пример:

```
package testpackages.je;
import java.io.*;
/**
*This is demo class, which have one problem - empty catch block.
* TermWare will automatically find this problem for us.
*/
public class JET1 {
  public static void main(String[] args)
  {
     String fname="qqq";
     File f=new File(fname);
     try {
       InputStream input = new FileInputStream(f);
       input.close();
      }catch(IOException ex){
       /* ignore - this must be catched */
      }
  }
}
```

Это является серьезным дефектом, так как игнорирование исключений, без генерации предупреждающих сообщений, делает невозможной эффективную отладку программ.

Соответствующее выражение на языке TermWare, определяющее образец:

```
java_catch($formal_parameter, java_empty_block )
                                    -> PROBLEM // emptyCatchClauseDiscovered($formal_parameter)");
```

- Перехватывание исключения общего вида, т. е. нахождение шаблона типа:

```
try {
...
}catch(Exception $x){
...
}
```

Это является дефектом в том случае, если класс использует обратные вызовы: клиенты могут передавать информацию в другие части программы с помощью своих непроверяемых исключений.

- Генерация исключений общего вида, т.е. нахождения шаблона:

method(x1...xn) throws Exception

Это является дефектом, так как вынуждает пользователей этого класса перехватывать исключения общего вида.

3.2 Нарушение контракта API

Наиболее часто описываемый случай нарушения контракта - это перегрузка метода *equals* без модификации метода *hashCode*, что приводит к неопределенному поведению программы когда объекты, модифицированные таким образом, используются со стандартными коллекциями *HashSet* и *HashMap*.

Покажем как построить систему переписывающих правил, проверяющих тот факт, что если перегружен метод **equals**, то необходимо перегрузить и метод **hashCode**.

Система переписывающих правил будет иметь вид:

```
system(OverloadedEquals,
    javaFacts("ProxyJavaFacts",   "ua.kiev.gradsoft.JavaChecker.ProxyJavaFacts"),
        ruleset(
            import(general,apply),
            import(general,logical_and),
```

Собственно правила, которые мы составим позже.

```
    ),
    FirstTop
);
```

Здесь в самом начале мы видим заголовок системы и то, что она использует БД Фактов, реализуемую классом **ProxyJavaFacts**. При редуцировании термов мы в условиях и действиях можем использовать методы класса **ProxyJavaFacts**.

Далее, мы импортируем из обычную семантику двух стандартных преобразований: вызова другой системы и логического "и".

Стратегия FirstTop означает, что редуцироваться будет всегда "самый верхний терм" выражения.

Перейдем к логике: алгоритм нахождения двух методов в дереве программы можно представить как сжимающее отображение - отбрасываем все, кроме определений классов, а в определениях классов редуцируем все, кроме интересующих нас методов.

Начинаем мы работы с терма *java_compilation_unit($package,$imports,$classes)*, соответственно первое выражение будет иметь вид:

```
java_compilation_unit($x,$y,$z) -> checkType($z),
```

Преобразовываем операции над списком типов к операциям над одним типом:

```
checkType([$x,$y]) -> checkType($x) && checkType($y),
checkType([$x]) -> checkType($x),
```

Теперь, если мы нашли класс, то:

```
checkType(java_class_declaration($name,$properties,
$extend_classes,$implements_classes,$body)) -> checkClassBody($body,$name),
```

Интерфейсы нас не интересуют.

```
checkType(java_interface_declaration($x,$y,$z,$w)) -> true,
```

Теперь понятно, почему мы импортировали логическое умножение - **true && X** сводится к **X**, таким образом мы сокращаем целевой терм.

А в терме для тела класса проводим аналогичные сокращения:

```
checkClassBody([$x1,$x2] ,$classname) ->
                                checkClassBody($x1,$classname) && checkClassBody($x2,$classname),
```

Пока не доходим до целевого тема, вида:

```
checkClassBody(
   java_method_declaration(
            java_identifier("equals"),
            $attributes,
            $result_type,
            $formal_parameters,
            $exceptions,
            $method_body),
            $classname) -> CHECK_EQUALS($classname,  checkEqualsParams($formal_parameters)),

CHECK_EQUALS($classname, true) -> FOUND_EQUALS($classname),
```

То же самое преобразование проводим и с **hashCode**, все остальные методы – редуцируем к **true** с

помощью следующего правила:

```
checkClassBody(
    java_method_declaration(
                    $name,
                    $attributes,
                    $result_type,
                    $formal_parameters,
                    $exceptions,
                    $method_body),
     $classname)  [| $name != java_identifier("equals") &&
                   $name != java_identifier("hashCode")  ] -> true,
```

Для вложенных дела раций классов повторяем ту же операцию:

```
checkClassBody(
        java_class_declaration($name,$properties,$extend_classes,
                               $implements_classes,$body),
        $classname
       )                ->      checkClassBody($body,subclass($classname,".",$name)) ,
```

В коце-концов, наш терм сведется к последовательности типа:
FOUND_EQUALS($x) && FOUND_HASHCODE($x) && FOUND_EQUALS($y)

Если у нас определены как **equals**, так и **hashCode**, значит в полученном терме будет выражение типа *FOUND_EQUALS(x)&&FOUND_HASHCODE(x)*.

Их мы можем средуцировать с помощью следующих двух правил:

```
FOUND_EQUALS($classname) && FOUND_HASHCODE($classname) -> true,
FOUND_HASHCODE($classname) && FOUND_EQUALS($classname) -> true,
```

Теперь все. Программа, которая не содержит погрешности отредуцирована в истину.

4. Определения нарушений синхронизации

Следующая часто встречающаяся проблема - обработка многопоточности.

Рассмотрим следующий пример:

```
01 package testpackages.sv;
02
03
04 public class Sv1
05 {
06
07        public synchronized void setMyVariable(int x)
08        {
09           myVariable_=x;
10         }
11
12         public void incrementMyVariable()
13        {
14           ++myVariable_;
15         }
16
17        private int myVariable_=0;
18 }
```

Очевидно, что класс *Sv1* содержит погрешность проектирования - судя по методу *setMyVariable*, к нему предусмотрен многопоточный доступ, но в *incrementMyVariable* программист об этом забывает.

Погрешности подобного рода приводят к тому, что программа нормально работает при небольшой нагрузке, а при промышленной эксплуатации начинаются таинственные сбои.

Соответствующее правило программирования можно сформулировать следующим образом:

Если класс предоставляет многопоточный доступ, и в каждый момент времени объект должен предоставлять непротиворечивое виденье переменной, то все использование этой переменной должно происходить либо в синхронизированных методах, либо внутри блока synhronized относительно объекта.

Заметим, что этот дефект нельзя определить, пользуясь только шаблонами сопоставления, так как нам требуется провести анализ зависимости синхронизирующего объекта и защищенной переменной.

Однако с помощью правил переписывания это возможно.

Неформально, алгоритм можно описать следующим образом:

- Определить множество переменных-членов класса, у которых синхронизированы функции доступа.
- Для каждой пары *(v,s)*, где *v* - терм переменной, *s* - синхронизирующий объект, применять следующие правила:

  - Для каждого несинхронизированного метода класса *m* с модификатором доступа *public* найти все вхождения переменной *v* в *m* за пределами блока *synchronized(s){..}*.

  - Для каждого вложенного класса *c* убедиться, что либо *v* не используется в методах *c*, либо *v* находится в блоке *synchronized(super.this).....*

  - Если одно из этих условий нарушено - значит у нас есть потенциальная опасность при многопоточном доступе к переменной.

Пользуясь частичной моделью программы JavaChecker определяет пары *(v,s)*, а затем для каждой пары и для каждого несонхринизированного публичного метода *s* редуцирует терм *C(m,v,java_this)* с помощью следующих правил:

C([$x],$var,$synchronizer) -> C($x,$var,$synchronizer),
C([$x,$y],$var,$synchronizer) -> C($x,$var,$synchronizer) && C($y,$var,$synchronizer),
C(java_switch($expr,$statements),$v,$s) -> C($expr,$v,$s) && C($statements,$v,$s),
C(java_switch_label($label,$statement),$v,$s) -> C($statement,$v,$s),
C(java_if($t1,$t2,$t3),$v,$s) -> C($t1,$v,$s) && C($t2,$v,$s) && C($t3,$v,$s),
....
C(java_assign($t1,$t2),$v,$s) -> C($t1,$v,$s) && C($t2,$v,$s),
.....

и так для каждой конструкции языка Java, кроме имен, синхронизаторов и вложенных определений классов.

В конце концов, разбирая выражения мы доберемся либо до переменной:

C($v,$v,$s) -> false // synchronizeViolationDiscovered($v,$s),
C(java_name([$v]),$v,$s) -> false // synchronizeViolationDiscovered($v,$s),
C(java_identifier($x),$v,$s) [| java_identifier($x) != $v |] -> true,

либо до синхронизатора:

C(java_synchronized($s,$x),$v,$s) -> true,
C(java_synchronized($s1,$x),$v,$s) [| $s1 != $s |] -> C($x,$v,$s),

либо до конструктора вложенного анонимного класса.

C(java_dot(java_new($t1,$t2,$t3)),$v,$s) -> C($t2,$v,$s) && CB(t3,$v,$s),

сокращая определения вложенного класса методом, аналогичным предыдущему примеру, мы приходим к анализу методов вложенных классов. Единственное отличие - если раньше синхронизирующей переменной была *this*, то во вложенном классе – *this.super*

```
CB1(
        java_method_declaration(
                                $name,
                                $attributes,
                                $result_type,
                                $formal_parameters,
                                $exceptions,
                               $method_body),
                                        $v, $s) -> C($x,$v,FromSuperClass.$s),
```

Где система FromSuperClass состоит из единственного преобразования:

```
java_this -> java_primary_experssion(java_this,java_dot(java_super))
                                                        // setCurrentStopFlag(true),
```

Таким образом мы построили систему переписывающих правил, которые редуцируют java терм в true, если в программе есть дефекты синхронизированных переменных, и false в противном случае.

Заметим, что в построенном алгоритме есть одно упрощение - мы не учитываем случай, когда имя формального параметра метода, либо имя локальной переменной скрывает *v*, как в следующем блоке

кода:

```
public synchronized void setMyVariable(int x)
{
  myVariable_=x;
}
public void incrementMyVariable()
{
 int myVariable_=0;
  ++myVariable_;
}
```

Сокрытие имени члена класса формальным параметром либо процедурой является дефектом стиля и определяется с помощью отдельной процедуры проверки.

5. Дефекты стиля

Стиль программирования является важным элементом поддержки качества программного проекта - программы, соответствующие определенному стилю легче читать и модифицировать; с помощью соглашений стиля в наименованиях переменных кодируется дополнительная информация, такая как имена паттернов проектирования, которая облегчает анализ и разработку кода. К примеру, если у нас в программе есть класс, с наименованием *XxxFactory*, то мы с определенной долей вероятности полагаем, что это объект-фабрика для классов типа *Xxx*, которые создаются с помощью метода *create*. Если класс типа *XxxFactory* не обладает методом *create*, то с проектированием этого класса что-то не то.

Существует множество стандартов соблюдения стиля Java программ, в основном это специализации стандарта кодирования Sun на языке Java [9]. В JavaChecker определен настраиваемый интерфейс проверки стиля программирования проектов, включающий в себя:

- Регулярные выражения для разных классов символов: (Имен классов, имен методов, имен полей классов)
- Проверка дефектов сокрытия имен.
- Проверка соблюдения правил инкапсуляции объектов. (таких, например, как неиспользование публичных классов-данных)

В дальнейшем планируется дополнить подсистему проверки стиля проверкой использования наименований широко известных шаблонов проектирования.

6. Анализ применения

Нами был проведен анализ нескольких широко известных свободно распространяемых пакетов. Результаты приведены в следующей таблице.

	Jetty - 4.2.9	Jetty - 4.2.10pre0	Mckoi - 1.0.2	JBoss - 3.2.3	dbcp - 1.1
empty catch clauses :	15	24	39	653	18
generic exception specs :	64	103	2	3906	107
generic exception catch :	224	393	6	2448	48
ov. equals or hashCode	1	3	9	27	0
synchronize violations :	6	10	0	14	3
Files :	344	528	344	4056	57

Результаты, признаться, немного шокируют - даже те дефекты, о которых рассказывается в каждом учебнике присутствуют в четырех из пяти программных комплексах. Видно, что вопросы качества ПО довольно актуальны в настоящее время.

Также следует заметить, что в абсолютно всех анализируемых пакетах есть обращение к исключением общего вида. Это означает, что Java-модель исключений слишком сложна для применения

в разработке больших программных комплексов.

Пристальное рассмотрение обнаруженных дефектов показывает, что:

- Обработка исключение общего вида оправданно в нескольких довольно редких случаях. Однако большинство обнаруженных вхождений представляли собой дефекты.
- Истинность дефекта синхронизации связанна с методикой использования многопоточного доступа. Так, например, в **commons-dbcp-1.1** с помощью synchronized эмулируется не семантика исключительного доступа, а исключительный доступ на запись при параллельном доступе на чтение. Отсутствие взаимных блокировок записи обеспечивается с помощью синхронизатора, отсутствие блокировок записи и чтения обеспечивается атомарностью присваивания.

Обнаруженные факты дают нам направление дальнейшего развития - так как текст программы не определяет однозначно методику использования той или иной конструкции, необходимо добавить возможность для программиста указывать необходимость и тип проверки в комментариях.

Еще один вывод, который можно сделать их результатов анализа -- стандарты кодирования, применяемые в каждой из этих проверок различны -- стандарты кодирования Sun в целом соблюдаются, но в каждом из рассматриваемых проектов соблюдался также более специализированный вариант, несовместимый с остальными.

7. Вывод

Таким образом мы показали возможность применения методов символьных преобразований для анализа архитектуры программных комплексов на примере автоматического определения потенциальных стандартных ошибок.

Дальнейшая работа связана, в первую очередь с откликом от использования JavaChecker в индустрии.

В первую очередь, необходимо:

- интегрировать процесс использования JavaChecker в процесс разработки ПО в пилотном проекте.
- определить разность изменения метрик и наличие экономического эффекта.
- определить влияние использования средств автоматического анализа на скорость обучения программиста.
- расширить типологию дефектов программирования и встроить в JavaChecker интерфейс подключения внешних описаний правил.

В связи с этим заметим, что сам JavaChecker и полные тексты всех правил проверки доступны в Internet, с сайта http://www.gradsoft.kiev.ua. Было бы интересно сотрудничать с организациями, разрабатывающие ПО для апробации средств использования статического анализа.

Весьма интересными представляются также исследования проблемно-ориентированных контрактов использования API, когда вместе с проблемно-ориентированной библиотекой поставляется список правил, определяющий контракты ее использования, и разработка подобных систем для других языков - в первую очередь **C#** и **IDL**.

Библиография

1. А. Дорошенко. Р. Шевченко. Система символьных вычислений для программирования динамических приложений. *Проблемы программирования., 4.2003.*
2. Tom Copeland. Static analysis with pmd. *OnJava.com*, 2003. http://www.onjava.com/pub/a/onjava/2003/02/12/static_analysis.html.
3. Ruslan Shevchenko. Anatoliy Doroshenko. Managing business logic with symbolic computations. *Lecture Notes in Informatics V. 30 - Proceeding of Information Systems Technology and its Applications 2003*, pages 143–152, 2003.
4. Visser E. Stratego: A language for program transformation based on rewriting strategies. system description of stratego 0.5. *Rewriting Techniques and Applications (RTA 01)*, 2051.:357–361, 2001.
5. David Hovemeyer and William Pugh. Finding bugs is easy. 2003. http://www.cs.umd.edu/˜pugh/java/bugs.
6. S. C. Johnson. *Lint, a c program checker*, 1985. in Unix Programmer Supplementary Documents, vol. 1.
7. J.V.Kapitonova A.A.Letichevsky M.S.L'vov and V.A.Volkov. Tools for solving problems in the scope of algebraic programming. *LNCS*, 958, 1995.
8. Ruslan Shevchenko. Termware semantics,. 2004. http://www.gradsoft.kiev.ua/Products/TermWare/Semantics_eng/.
9. Sum Microsystems. Code Conventions for the Java Programming Language. April 1999, http://java.sun.com/docs/codeconv/