Реализация средств взаимодействия между обнаружителями дефектов в шаблонном языке поиска для синтаксических деревьев

С. В. Сыромятников, ИСП РАН

Поиск дефектов на синтаксических деревьях

- Сравнительная простота написания
- Сравнительно высокая скорость работы
- Меньшая по сравнению с анализом потока управления, но зачастую достаточная выразительная мощность
- Пользователю просто писать свои обнаружители дефектов

Интерфейсы для создания обнаружителей дефектов

- API
  - Высокая выразительная мощность
  - Существенные накладные расходы, особенно для простых правил
- Декларативный интерфейс
  - Удобство реализации простых правил
  - Недостаточная выразительная мощность

KAST

- Декларативный язык на основе XPath
- Развивается с 2007 года
- Активно используется в продукте компании Klocwork
- Использовался для реализации большинства правил MISRA

KAST

- //BinaryExpr [@Op = KTC_OPCODE_GE] [Left.isUnsigned()] [Right.getIntValue() = 0]

  Позволяет находить всегда истинные сравнения беззнаковой переменной с 0 на “**>=**”

```
for (unsigned int i = 100; i >= 0; i--) sum += a[i];
```

- // ConditionalExpr / Cond::* [isConstant()]

  Находит константные условия в тернарных выражениях

```
const int max_len = 256;
const char *message = max_len > 255 ? "good" : "bad";
```

KAST: преимущества и проблемы

Преимущества:

- Простота освоения
- Удобство написания небольших обнаружителей дефектов

Проблемы:

- Локальность:
  - для нахождения дефекта часто требуется анализировать синтаксически не связанные друг с другом фрагменты дерева
  - при анализе каждого поддерева может потребоваться информация, накопленная в процессе анализа других поддеревьев
- Как следствие, недостаточная по сравнению с API выразительная мощность

Примеры проблемных правил

- *В каждом семействе виртуальных функций в иерархии наследования классов должно быть не более одного определения (при неограниченном числе деклараций)*
- *В коде на языке C не должно быть неиспользуемых статических функций*

Для реализации данных правил пришлось использовать AST API

Улучшаем KAST: глобальные переменные

- Новый псевдотип Global
- Типичный случай использования коллекций:

  ```
  //Global [ $$storage := {} ]
  ```

- Типичный сценарий:
  - проверка в шаблоне уже имеющейся в хранилище информации и соотнесение её с характеристиками сопоставляемого в данный момент поддерева
  - в зависимости от результатов - выдача/невыдача дефекта, пополнение хранилища информацией о текущем поддереве
- Иногда этого мало: требуется анализ дерева целиком

Реализация первого проблемного правила

В каждом семействе виртуальных функций в иерархии наследования классов должно быть не более одного определения (при неограниченном числе деклараций)

```
<pattern>
   //Global [ $$vfuncs := { } ]
</pattern>
<pattern>
   //MemberFunc
        [ isVirtual() ]
        [ not isPureVirtual() ]
        [ not isDestructor() ]
        [ $sema := getSemanticInfo() ]
        [ if ( for ($i, $$vfuncs, (not overloads($sema,$i)) & (not overloads($i,$sema))),
              $$vfuncs.add-element($sema) & false(), $$vfuncs.add-element($sema)) ]
</pattern>
```

Улучшаем KAST: финальные блоки

- Новый псевдотип Finally
- Действия над глобальными переменными
- Требуется явная функция сообщения о дефекте (report)
- Типичный сценарий:
  - обычные шаблоны (сопоставляемые с небольшими поддеревьями синтаксического дерева) лишь накапливают информацию
  - в финальном блоке происходит выдача сообщений о найденных дефектах на основе анализа всего глобального хранилища

Реализация второго проблемного правила

В коде на языке C не должно быть неиспользуемых статических функций

```
<pattern>
  // Global [ $$fdecls := { } ] [ $$fcalls := { } ]
</pattern>
<pattern>
  // FuncDeclarator [  isCLanguage() ] [ isGlobal() ] [  isStatic()) ] [ $sema := getSemanticInfo() ]
    [ if (for($i,$$fdecls,getSemanticInfo() != getSemanticInfo($i)), $$fdecls.add-element(this()) & false(), false()) ]
</pattern>
<pattern>
  // CallExpr [ $sema := getSemanticInfo() ] [ $sema.isGlobal() ] [ $sema.isStatic() ]
    [ if ( for ($i, $$fcalls, $sema != getSemanticInfo($i)), $$fcalls.add-element($sema) & false(), false()) ]
</pattern>
<pattern>
   // Finally [ for ($i, $$fdecls, $$fcalls.member(getSemanticInfo($i)) | report($i)) ]
</pattern>
```

Дальнейшие исследования

- Возможность переопределения переменных
- Поддержка новых типов данных (например, отображений)
- Новые встроенные функции

Спасибо за внимание!