Использование аспектно-ориентированного программирования для выполнения запросов по исходному коду программ[1]

Новиков Е.М., Хорошилов А.В.
joker@ispras.ru, khoroshilov@ispras.ru

Аннотация. Запросы по исходному коду программ помогают разработчикам обнаруживать искомые фрагменты кода и определять их взаимоотношения друг с другом. Для выполнения запросов по исходному коду автоматизированным образом существуют различные подходы от достаточно простых, основывающихся на текстовом поиске по шаблонам, до более интеллектуальных, осуществляющих поиск на основе формального представления программ и позволяющих использовать для запросов естественные языки. В статье предлагается подход к выполнению запросов по исходному коду программ на основе аспектно-ориентированного программирования, рассматриваются достоинства и недостатки такого подхода.

Ключевые слова: разработка программы; поддержка программы; запрос по исходному коду программы; формальное представление программы; аспектно-ориентированное программирование.

1. Введение

При разработке и поддержке программ важными задачами являются обнаружение искомых фрагментов кода и определение их взаимоотношений друг с другом. Данные задачи возникают, например, когда разработчики пытаются понять на уровне исходного кода, как в программе реализована та или иная функциональность; при рефакторинге программ; при обратном проектировании; во время отладки программ и при вычислении различных метрик исходного кода. С целью обнаружения искомых фрагментов кода и определения их взаимоотношений друг с другом используются запросы по исходному коду программ.

[1] Работа поддержана ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы" (контракт N 11.519.11.4006)

Все возможные запросы по исходному коду программ могут быть условно отнесены к одному из трех типов [1]:

- Запросы по общей структуре программ. Например, о составе и связи файлов, функций, переменных и типов, составляющих программу.
- Запросы по структуре выражений. Например, найти все вызовы функции, использования поля структуры, циклы и т.д.
- Запросы по потоку управления и потоку данных. Например, найти последовательности выражений, в которых участвует некоторая переменная.

Делать запросы по исходному коду вручную для больших программ достаточно трудоемко, поскольку объем исходного кода очень велик, а взаимоотношения между различными его частями бывают очень сложными. С целью автоматизации процесса были предложены различные подходы от достаточно простых, основывающихся на текстовом поиске по шаблонам, до более интеллектуальных, осуществляющих поиск на основе формального представления программ и позволяющих использовать для запросов естественные языки. В статье предложен подход к написанию и выполнению запросов по исходному коду программ на основе аспектно-ориентированного программирования (АОП).

В разделе 2 данной статьи приведено описание основных понятий АОП. В разделе 3 кратко рассматриваются возможности C Instrumentation Framework, одной из реализаций АОП для языка программирования Си. В разделе 4 обсуждается возможность использования АОП для выполнения запросов по исходному коду программ. Показывается, как с этой целью был доработан C Instrumentation Framework. В разделе 5 предложенный подход сравнивается с существующими подходами. В заключении подводятся итоги работы и рассматриваются направления дальнейшего развития.

2. Основные понятия аспектно-ориентированного программирования

Аспектно-ориентированное программирование (АОП) предлагает специальные средства для поддержки модульности программ в тех случаях, когда существующие средства языков программирования не позволяют выделить определенную функциональность в отдельные модули. Подобная функциональность получила название *сквозной функциональности*. Примерами сквозной функциональности являются ведение журнала, трассировка и обработка ошибок.

В АОП сквозная функциональность программы выделяется в отдельные модули, так называемые *аспекты*, с помощью срезов и рекомендаций. В данной статье под *срезом* будет пониматься описание множества фрагментов исходного кода (*точек соединения*) программы, удовлетворяющих

некоторому логическому условию. Посредством среза можно задать, например, соответствие всем вызовам функций выделения памяти (таким как *malloc*, *calloc* и т.д.).

Рекомендация указывает, каким образом необходимо преобразовать точки соединения программы, соответствие которым задается некоторым срезом. Как правило, эти преобразования заключаются в добавлении набора инструкций до, после или вместо соответствующих точек соединения. Инструкции записываются с помощью того же языка программирования, на котором пишется целевая программа. Помимо этого АОП предоставляет возможность использовать информацию о соответствующей точке соединения, например, имя вызываемой функции, типы ее аргументов и т.п.

```
// Аспект Logging состоит из среза и рекомендации.
aspect Logging {
   // Срез move задает соответствие точкам соединения программы,
   // вызовам методов.
   pointcut move():
     call(void FigureElement.setXY(int,int)) ||
     call(void Point.setX(int))              ||
     call(void Point.setY(int));
   // Рекомендация говорит, что перед выполнением точек соединения
   // среза на экран должно быть напечатано сообщение.
   before(): move() {
     System.out.println("about to move");
   }
```

Рис. 1. Пример аспекта AspectJ, выделяющего сквозную функциональность ведения журнала для графической системы.

На Рис.1 показан пример аспекта AspectJ, посредством которого для графической системы выделена сквозная функциональность, ведение журнала [7]. Для этого в аспекте задается срез *move*, задающий соответствие точкам соединения, вызовам методов *setXY* класса *FigureElement* и вызовам методов *setX* и *setY* класса *Point*. Кроме того, в аспекте *Logging* задается рекомендация, которая говорит, что перед выполнением точек соединения среза *move* (вызовом соответствующих методов) на экран должно быть напечатано сообщение.

Процесс применения аспектов к целевой программе, при котором для соответствующих срезам точек соединения выполняются указанные рекомендациями преобразования, называется *инструментированием*. Предполагается, что инструментирование программ должно выполняться автоматически.

Чтобы реализовать АОП для некоторого языка программирования необходимо определить механизм описания аспектов и разработать средства инструментирования программ. В настоящее время существует достаточно большое количество реализаций АОП для различных языков программирования. В следующем разделе приведено краткое описание инструмента C Instrumentation Framework, который является одной из реализаций АОП для языка программирования Си.

3. C Instrumentation Framework - реализация АОП для языка программирования Си

Реализация АОП для языка программирования Си C Instrumentation Framework (CIF) [5,6] была разработана в рамках проекта верификации драйверов Linux Driver Verification [2-4]. В этом проекте возможности CIF используются, во-первых, для формализации правил корректного поведения драйвера в виде аспектов, а во-вторых, для подготовки исходного кода драйверов к верификации путем его инструментирования на основе данных аспектов. Далее в статье подробнее рассмотрен процесс инструментирования CIF. Одновременно с этим рассматриваются возможности CIF для разработки аспектов.

CIF выполняет инструментирование исходного кода программы в течении 4 этапов:

- На 1-м этапе CIF позволяет дописывать текст до и после инструментируемого исходного кода. Благодаря этому, например, в начало файла можно добавить прототипы функций, необходимые перед их первым использованием, и включения дополнительных заголовочных файлов.
- На 2-м этапе выполняется стандартное препроцессирование исходного кода программы. На этом этапе CIF позволяет инструментировать соответствующие срезам макроопределения, макрофункции и их подстановки. В случае макроопределения и макрофункции в рекомендации предоставляется возможность задать собственный текст для подстановки (для макрофункции в данном тексте можно ссылаться на формальные параметры). Для подстановок макрофункций CIF позволяет использовать фактические параметры.
- На 3-м этапе CIF позволяет инструментировать вызовы и определения функций, простые использования и присваивания переменных и параметров. При этом в рекомендации возможно использовать произвольный корректный код на языке Си и ссылаться на формальные параметры, на их типы, на тип возвращаемого значения, на имя функции, переменной или параметра в зависимости от того, соответствие какой точке соединения задает срез. Данный код реализуется в виде вспомогательных функций. Также на 3-м этапе

можно расширить определения составных типов данных (структур, объединений и перечислений), дописав в начало или в конец дополнительные поля или константы.

- На 4-м заключительном этапе CIF связывает исходные конструкции с вспомогательными функциями, после чего выдает на вывод либо исходный код, либо продолжает работать дальше, как стандартный компилятор.

На каждой из 4 стадий CIF вызывает инструмент Aspectator, который представляет собой модифицированную версию компилятора GCC 4.6.1 [8]. Благодаря этому CIF может обрабатывать исходный код на языке Си с расширениями GNU и выдавать на выход те представления программы, которые поддерживаются GCC. Дополнительно к этому поддерживается вывод в виде исходного кода.

4. Использование реализации АОП для выполнения запросов по исходному коду программ

С точки зрения выполнения запросов по исходному коду реализацию АОП можно использовать следующим образом. Запросы естественным образом писать с помощью аспектов. Поскольку для выполнения запросов инструментирование кода как таковое не требуется[2], достаточно задействовать возможности АОП по заданию срезов, поиску соответствующих точек соединения в коде программы и получению информации касательно данных точек соединения. Таким образом, получается, что единственное дополнительное требование к реализации АОП, необходимое для выполнения запросов по исходному коду программ, заключается в возможности вывода полученной информации о точках соединения, соответствующих заданным в аспектах срезам.

Для того чтобы использовать C Instrumentation Framework, реализацию АОП для языка программирования Си, для выполнения запросов по исходному коду в инструменте была добавлена поддержка специальной конструкции *$fprintf*. Данная конструкция позволяет напечатать в указанный файл информацию о соответствующей точке соединения на этапе выполнения инструментирования программы. Далее в статье рассмотрены несколько примеров запросов по исходному коду, которые можно сделать с помощью доработанной версии инструмента.

[2] В статье не рассматриваются запросы к исходному коду программ, результаты которых анализируются по выводу, генерируемому в процессе выполнения данных программ.

4.1. Получение фактического параметра макрофункции

```
before: expand(module_init(init))
{
  $fprintf<out_file_name,$arg_val1>
}
```

Рис. 2. Задание функции загрузки драйвера устройства электронной защиты Parallax LiteLink..

```
linux/drivers/net/irda/litelink-sir.c:
static int __init litelink_sir_init(void)
{
        return irda_register_dongle(&litelink);
}
module_init(litelink_sir_init);
```

Рис. 3. Аспект для получения фактического параметра макрофункции module_init.

Запросы на получение фактического параметра макрофункции возникают при построении окружения для драйверов в проекте верификации драйверов Linux Driver Verification [2-4]. Имена функций загрузки драйверов, необходимые для построения окружения, передаются в качестве параметров макрофункции *module_init* (Рис. 2). Задать запрос по исходному коду драйвера можно с помощью аспекта, представленного на Рис. 3. Данный аспект требует перед выполнением подстановки макрофункции *module_init* с одним параметром напечатать в файл *out_file_name* ее первый фактический параметр. В результате применения аспекта к файлу драйвера *linux/drivers/net/irda/litelink-sir.c* в данный файл будет напечатано имя функции загрузки драйвера *litelink_sir_init*.

На Рис. 4 представлен способ задания функции загрузки драйвера ОС Linux через несколько последовательных подстановок макрофункций. При препроцессировании сначала *module_usb_driver(karma_driver)* будет заменена на *module_driver(karma_driver, usb_register, usb_deregister)*, а эта подстановка в свою очередь – на определение функции загрузки драйвера и *module_init(karma_driver_init)*. Для выполнения запроса в случае

использования данного способа задания функции загрузки аспект, представленный на Рис. 3, также применим. В результате в файл *out_file_name* будет напечатано *karma_driver_init*.

```
include/linux/device.h:
#define module_driver(__driver, __register, __unregister, ...) \
static int __init __driver##_init(void) \
{ \
        return __register(&(__driver) , ##__VA_ARGS__); \
} \
module_init(__driver##_init);
...
include/linux/usb.h:
#define module_usb_driver(__usb_driver) \
        module_driver(__usb_driver, usb_register, \
                      usb_deregister)
drivers/usb/storage/karma.c:
module_usb_driver(karma_driver);
```

Рис. 2. Задание функции загрузки драйвера цифрового аудио проигрывателя Rio Karma..

Получение списка вызываемых функций, у которых тип одного из параметров является указателем

Для того чтобы задать запрос на получение списка вызываемых функций, у которых тип одного из параметров является указателем на структуру *mutex*, можно использовать аспект, приведенный на Рис. 6. При его применении к фрагменту исходного кода, представленному на Рис. 5, в файл *out_file_name* будет напечатано *mutex_lock, mutex_lock_nested, mutex_unlock, mutex_unlock*.


```
drivers/scsi/libfc/fc_npiv.c
void fc_vport_setlink(struct fc_lport *vn_port)
{
  ...
  mutex_lock(&n_port->lp_mutex);
  mutex_lock_nested(&vn_port->lp_mutex,
LPORT_MUTEX_VN_PORT);
  __fc_vport_setlink(n_port, vn_port);
  mutex_unlock(&vn_port->lp_mutex);
  mutex_unlock(&n_port->lp_mutex);
}
```

Рис. 5. Вспомогательная функция для библиотеки функций волоконно-оптического канала, вызывающая функции, у которых тип одного из параметров является указателем на структуру mutex.

```
before: call($ $(.., struct mutex *, ..))
{
  $fprintf<out_file_name,$func_name>
}
```

Рис. 6. Аспект для получения списка вызываемых функций, у которых тип одного из параметров является указателем на структуру mutex.

4.2. Получение списка функций, в которых изменяется глобальная переменная

Предположим, что перед разработчиком стоит задача определить те функции реализации уровня записи пакетов для устройств CD-RW, DVD+RW, DVD-RW и DVD-RAM, в которых изменяется глобальная переменная *pkt_debugfs_root* (Рис. 7). С этой целью он может использовать запрос по исходному коду в виде аспекта, приведенного на Рис. 8. В результате в файл *out_file_name* будет напечатано только имя функции *pkt_debugfs_init*, поскольку в функции *pkt_debugfs_dev_new* переменная *pkt_debugfs_root* не изменяется.

```
drivers/block/pktcdvd.c
static struct dentry *pkt_debugfs_root;
static void pkt_debugfs_dev_new(struct pktcdvd_device *pd) {
  if (!pkt_debugfs_root)
    return;
  ...
}
static void pkt_debugfs_init(void)
{
  pkt_debugfs_root = debugfs_create_dir(DRIVER_NAME, NULL);
  ...
}
```

Рис. 7. Функции реализации уровня записи пакетов для устройств CD-RW, DVD+RW, DVD-RW и DVD-RAM, в которых встречается глобальная переменная pkt_debugfs_root.

```
before: set(static struct dentry *pkt_debugfs_root)
{
  $fprintf<out_file_name,$scope_func_name>
}
```

Рис. 8. Аспект для получения списка вызываемых функций, у которых тип одного из параметров является указателем на структуру mutex.

4.3. Поиск файла, в котором определяется тип данных

В качестве последнего примера рассмотрим типичную задачу поиска файла, в котором определяется тип данных. Например, для того, чтобы найти заголовочный файл в ядре операционной системы Linux, в котором определяется структура *device*, можно написать запрос в виде аспекта, представленного на Рис. 9. В результате применения данного аспекта к исходному коду ядра в файл *out_file_name* будет выведено имя заголовочного файла *include/linux/device.h*.


```
before: introduce(struct device)
{
  $fprintf<out_file_name,$scope_file_name>
}
```

Рисунок 9. Аспект для поиска заголовочного файла в ядре операционной системы Linux, в котором определяется структура device.

5. Существующие подходы

Для выполнения запросов по исходному коду программ существует много разнообразных подходов. Данные подходы продолжают развиваться по мере роста объема и сложности исходного кода. Один из первых подходов для выполнения запросов заключался в использовании регулярных выражений для поиска соответствующих текстовых фрагментов в программах. Данный подход обладает следующими преимуществами:

- высокая скорость работы;
- независимость языка запросов от языка программирования;
- отсутствие необходимости предварительного преобразования исходного кода программ в какое-либо формальное представление;
- терпимость к ошибкам в коде;
- простота использования.

Благодаря этим свойствам регулярные выражения используются для выполнения запросов по исходному коду программ и в настоящее время. Например, данный подход используется в проекте Linux Driver Verification для построения окружения драйверов. Опыт использования регулярных выражений в данном и других проектах показал, что некоторые запросы, в особенности, касающиеся сложных выражений и их взаимоотношений, сформулировать достаточно сложно. Зачастую при этом используются эвристики, которые не могут гарантировать ни то, что результаты запросов правильные, ни то, что вся необходимая информация извлекается.

Другие подходы для выполнения запросов по исходному коду программ предполагают предварительное преобразование кода к какому-либо формальному представлению, например абстрактному синтаксическому дереву или графу потока управления. Как правило, это, во-первых, позволяет существенно облегчить формулировку сложных запросов, а во-вторых, результаты выполнения данных запросов более точны по сравнению с использованием регулярных выражений. Основные отличия подходов заключаются:

- в языке написания запросов;
- в способе выполнения запросов;
- в способе хранения формального представления исходного кода программ.

Один из первых подходов, использующих предварительное преобразование исходного кода программ к формальному представлению, был реализован в инструменте SCRUPLE [9]. В качестве языка запросов в SCRUPLE используется расширение того языка, на котором написана программа. Для выполнения запросов, они преобразуются в конечные автоматы, после чего данные автоматы применяются к абстрактному синтаксическому дереву программы. В инструменте tawk [10] для написания запросов был предложен абстрактный язык, не привязанный к конкретному языку программирования.

С целью повышения эффективности хранения формального представления исходного кода программы и выполнения запросов было предложено использовать возможности реляционных баз данных. Один из первых инструментов, в котором был реализован данный подход, - это OMEGA [11]. OMEGA хранил формальное представление исходного кода и выполнял запросы неэффективно. Эти ограничения были частично устранены, например, в инструменте CodeQuest [12]. Запросы по исходному коду программ в инструменте CodeQuest делаются с помощью языка Datalog, который является подмножеством языка Prolog и используется специально для написания запросов к базам данных.

Позднее было замечено, что выполнение одного запроса по исходному коду не всегда предоставляет разработчику сразу всю необходимую информацию. Часто приходится выполнять несколько запросов, результаты которых не связывались друг с другом при использовании существующих инструментов. С целью решения данной проблемы были разработаны инструменты FEAT и JQuery [13, 14]. Отличие этих инструментов заключается в том, что JQuery позволяет привязать результаты выполнения запросов к иерархическому представлению исходного кода (например, к иерархии классов или к схеме вызываемых функций), а FEAT позволяет итеративно уточнять представление реализации сквозной функциональности в программе. FEAT и JQuery предназначены для выполнения запросов по исходному коду на языке программирования Java. Инструменты являются плагинами к интегрированной среде разработки Eclipse [15].

Для выполнения запросов к программам на языке Си формальное представление строится либо до препроцессирования кода (например, в инструменте Coccinelle [16]), либо после. Использование в качестве основы непрепроцессированного кода позволяет, во-первых, осуществлять поиск по всему исходному коду программы, а во-вторых, возвращать результаты в терминах оригинального представления программы. Однако, результаты

выполнения запросов могут оказаться менее точными, поскольку формальное представление в данном случае строится с использованием эвристик.

Одной из последних разработок в области выполнения запросов по исходному коду программ является поддержка естественных языков для написания запросов [17]. Использование данного подхода значительно упрощает процедуру написания запросов, поскольку от разработчиков не требуется знание специфичных языков запросов. Однако, результаты практического применения показали, что подход не является надежным (распознается около 80% запросов).

Предложенный в данной статье подход целесообразно сравнивать с существующими подходами, использующими предварительное преобразование исходного кода программ к формальному представлению. Данные подходы потенциально нацелены на выполнение произвольных запросов по исходному коду. По сравнению с ними предложенный подход не является универсальным, поскольку АОП поддерживает относительно простые точки соединения, что позволяет делать запросы по общей структуре программ и по структуре выражений, но не позволяет делать запросы по потоку управления и потоку данных.

К преимуществам предложенного подхода можно отнести то, что для его реализации в достаточно высокой степени переиспользуется существующая реализация АОП для соответствующего языка программирования. Для написания запросов используются практически те же средства АОП, что и для выделения сквозной функциональности в виде аспектов. Это позволяет разработчикам, которые знают или используют АОП, задействовать возможности предложенного подход для выполнения запросов по исходному коду без значительных усилий.

Ввиду того, что предложенная реализация основана на C Instrumentation Framework, который использует компилятор GCC для представления исходного кода программы в виде дерева разбора, она может быть использована для любых программ на языке Си с расширениями GNU. Для других подходов подобное зачастую является существенным ограничением, поскольку они используют собственные парсеры языка. Благодаря выполнению запросов на основе дерева разбора предложенный подход позволяет получать более точные результаты по сравнению с результатами, получаемыми на основе формального представления непрепроцессированного кода. Также можно отметить, что с помощью предложенного подхода можно делать запросы не только на основе дерева разбора программ, но и к конструкциям препроцессора, в том числе тем, которые возникают только в ходе выполнения препроцессирования. Насколько известно авторам статьи, последнее не поддерживается другими инструментами.

6. Заключение

В статье рассмотрена задача выполнения запросов по исходному коду автоматизированным образом. Данная задача остро встает перед программистами в процессе разработки и поддержки больших программ. Для решения задачи было предложено множество различных подходов. Самые первые подходы использовали для запросов по исходному коду программ регулярные выражения. Благодаря своей простоте и эффективности данные подходы используются до сих пор. Однако, с помощью регулярных выражений сложно написать запросы для получения информации о сложно взаимосвязанных фрагментах исходного кода. Поэтому в более поздних подходах было предложено осуществлять поиск по какому-либо формальному представлению исходного кода программ, например, абстрактному синтаксическому дереву. В дальнейшем было сделано много исследований по упрощению языка написания запросов (использование абстрактных и даже естественных языков) и по повышению эффективности их выполнения (использование реляционных баз данных для хранения формального представления исходного кода программы и выполнения запросов по нему).

В статье для выполнения запросов по исходному коду программ предложено использовать подход аспектно-ориентированного программирования. Для этого рассматриваются основные понятия АОП и особенности инструмента C Instrumentation Framework, который является одной из реализаций АОП для языка программирования Си. Показывается, что незначительная доработка инструмента позволяет добавить в CIF поддержку требуемой функциональности. В статье демонстрируются несколько примеров использования предложенного подхода, в том числе при построении окружения для драйверов в проекте верификации драйверов Linux Driver Verification.

Предложенный подход уступает по возможностям специализированным инструментам, поскольку он не позволяет делать запросы по потоку управления и потоку данных. К преимуществам подхода относится то, что его реализация в достаточно большой степени переиспользует реализацию АОП для соответствующего языка программирования и что подход позволяет задействовать традиционные средства АОП, используемые для выделения сквозной функциональности, для написания запросов по исходному коду программ.

Актуальная версия инструмента CIF с поддержкой выполнения запросов по исходному коду программ на языке Си доступна под лицензией GPLv3 [6].

Литература

[1] S. Paul and A. Prakash. Querying Source Code using an Algebraic Query Language. In Proceedings of the International Conference on Software Maintenance, pp. 127-136, 1994.

[2] В.С. Мутилин, Е.М. Новиков, А.В. Страх, А.В. Хорошилов, П.Е. Швед. Архитектура Linux Driver Verification. Труды Института системного программирования РАН, том 20, стр. 163-187, 2011.

[3] A. Khoroshilov, V. Mutilin, E. Novikov, P. Shved, A. Strakh. Towards an Open Framework for C Verification Tools Benchmarking. Proceedings of the Eighth International Andrei Ershov Memorial Conference «Perspectives of Systems Informatics» (PSI 2011), pp. 82-91, 2011.

[4] Проект верификации драйверов Linux Driver Verification. http://forge.ispras.ru/projects/ldv.

[5] E. Novikov. One Approach to Aspect-Oriented Programming Implementation for the C programming language. roceedings of the 5th Spring/Summer Young Researchers' Colloquium on Software Engineering, Yekaterinburg, pp. 74-81, 12-13 May, 2011.

[6] Реализация аспектно-ориентированного программирования для языка Си C Instrumentation Framework. http://forge.ispras.ru/projects/cif.

[7] Реализация аспектно-ориентированного программирования для языка Java AspectJ. http://www.eclipse.org/aspectj.

[8] GNU Compiler Collection. http://gcc.gnu.org.

[9] S. Paul, A. Prakash. A framework for source code search using program patterns. In IEEE Transactions on Software Engineering, pp. 463–475, 1994.

[10] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In WPC'96: Proceedings of the 4th International Workshop on Program Comprehension (WPC'96), Washington, DC, USA, p. 144, 1996.

[11] J. Ebert, B. Kullbach, A. Winter. Querying as an Enabling Technology in Software Reengineering, Proceedings of the Third European Conference on Software Maintenance and Reengineering, p.42, March 03-05, 1999.

[12] E. Hajiyev, M. Verbaere, O. de Moor. CodeQuest: scalable source code queries with datalog. Proceedings of the 20th European conference on Object-Oriented Programming, Nantes, France, July 03-07, 2006.

[13] M. P. Robillard, G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. Proceedings of the 24th International Conference on Software Engineering, May 19-25, Orlando, Florida, 2002.

[14] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD '03), 178-187, 2003.

[15] Интегрированная среда разработки Eclipse. http://www.eclipse.org.

[16] H. Stuart. Hunting bugs with Coccinelle. Masters Thesis, University of Copenhagen, August, 2008.

[17] M. Kimmig, M. Monperrus, and M. Mezini. Querying source code with natural language. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11), IEEE Computer Society, Washington, DC, USA, 376-379, 2011.