ПРОГРАММИРОВАНИЕ, 2013, № 4, с. 47–65

ЯЗЫКИ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

УДК 681.3.06

ПОДХОД К РЕАЛИЗАЦИИ АСПЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ ДЛЯ ЯЗЫКА СИ *

© 2013 г. Е.М. Новиков

Институт системного программирования РАН

109004 Москва, ул. А.Солженицына, 25

E-mail: novikov@ispras.ru

Поступила в редакцию 12.12.2012

Данная статья описывает подход к реализации аспектно-ориентированного программирования (АОП) для языка программирования Си. Излагаются традиционные средства АОП для различных языков программирования, и показывается, как особенности языка Си и процесса сборки программ на данном языке влияют на реализацию АОП. Далее рассматриваются дополнительные требования, которые накладывает практическое применение реализации АОП для программ на языке Си. Приводится описание существующих решений и анализируется возможность их использования. В статье описывается инструмент, реализующий предложенный подход, и демонстрируются возможности его применения.

1. ВВЕДЕНИЕ

Аспектно-ориентированное программирование (АОП) возникло как ответ на вопрос о том, как "правильно" декомпозировать большую программу на модули. Одной из самых оригинальных и, вместе с тем, мощных идей о критериях декомпозиции была идея Д.Л. Парнаса выделять в отдельный модуль каждое проектное решение, особенно, если речь идет о сложных решениях, которые в будущем будут с высокой вероятностью пересматриваться [1]. Парнас показал, что традиционных средств обеспечения модульности часто достаточно, но иногда их явно не хватает – нужны дополнительные возможности. Среди отечественных специалистов примерно в то же время, в 70-х годах, этой же проблемой занимался А.Л. Фуксман, который говорил о необходимости ввести в практику программирования специальные конструкции для "сосредоточенного описания рассредоточенных действий" [2][1].

Распространенные в настоящее время языки программирования, как правило, не поддерживают такие конструкции. Например, те виды модулей, которые имеются в процедурных или в объектно-ориентированных языках, помогают декомпозировать программную систему по принципу группировки функционально связанных компонентов и/или данных. Но есть и другие способы декомпозиции, которые могут в большей степени зависеть не от структуры реализации основной функциональности, а от архитектурных решений, направленных, например, на выполнение технологических задач. Соответствующая подобным задачам "дополнительная" функциональность получила название **сквоз-**

*Работа поддержана ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы" (контракт № 07.514.11.4104).

[1]Заметим, что задача декомпозиции встает не только на фазе проектирования программы. В ходе жизненного цикла возникает потребность в "оснастке" проекта разработки некоторыми программными модулями, которые используются для целей отладки, мониторинга, оптимизации, анализа безопасности и т.д. Такие "модули", с одной стороны, не являются неотъемлемой частью программы, с другой стороны, общее решение задачи декомпозиции, должно включать в рассмотрение и эти модули тоже.

ной функциональности[2] (англ. *cross-cutting concerns*).

Описание сквозной функциональности приходится распределять по тексту программы, что запутывает его и приводит к разрастанию кода. Повышается вероятность появления ошибок в связи с некорректным внесением модификаций во фрагменты программной системы, связанные со сквозной функциональностью. Типичными примерами сквозной функциональности являются ведение журнала, трассировка, конфигурационное управление и обработка ошибок. Более сложные примеры сквозной функциональности встречаются при обеспечении безопасного доступа к системам, при работе с базами данных, при проведении транзакций и т.д.

В 90-е годы была предложена концепция аспектно-ориентированного программирования, которая предоставила специальный механизм для пополнения традиционных средств поддержки модульности в плане основной функциональности средствами поддержки модульности в плане сквозной функциональности. АОП расширяет возможности существующих языков программирования, органично дополняя их. Изначально АОП разрабатывалось для объектно-ориентированных языков программирования, в первую очередь, Java; однако, поддержка АОП может быть добавлена и для процедурных языков, таких как, например, Си.

Проект, в рамках которого выполнялась данная работа, посвящен разработке системы верификации драйверов операционной системы Linux LDV [10–13]. Возможности АОП рассматриваются в этом проекте, как основное средство для автоматического инструментирования программного кода драйверов перед выполнением собственно верификации. Драйверы операционных систем обычно пишутся на языке Си, при этом используется практически весь арсенал языковых возможностей. По этой причине было решено разрабатывать не узкоспециализированное средство инструментирования, а полноценную реализацию АОП, которая впоследствии может иметь и самостоятельную ценность. В связи с этим предлагаемые в данной статье возможности реализации АОП полезны для разнообразных Си-программ, но часть возможностей адресуется к специфике инструментирования кода компонентов ядра операционной системы Linux и драйверов этой системы, в частности. Соответственно, часть статьи посвящена АОП в целом и особенностям реализации АОП для языка Си, а другая часть посвящена деталям, обусловленным потребностями проекта LDV.

В разделе 2 данной статьи рассматриваются основные понятия аспектно-ориентированного программирования и приводятся поясняющие их примеры. Раздел 3 посвящен традиционным способам описания аспектов. В разделе 4 обсуждается влияние особенностей языка программирования Си на реализацию АОП. В разделе 5 рассматриваются требования к реализации АОП для языка Си, которые предъявляются проектом LDV. Раздел 6 описывает существующие реализации АОП для Си. Инструмент, который реализует предложенный подход к реализации АОП для языка программирования Си, представлен в разделе 7. Раздел 8 оценивает возможности практического применения разработанного инструмента и существующих реализаций АОП для Си. В разделе 9 подводятся итоги и рассматриваются направления дальнейшего развития.

2. ОСНОВНЫЕ ПОНЯТИЯ АОП

Одним из основных понятий аспектно-ориентированного программирования является **точка соединения** (англ. *join point*). Существуют различные определения данного понятия [3, 4, 8, 9, 14–16]. В данной статье под точкой соединения будет пониматься программная конструкция, которая может быть связана с описанием некоторой части сквозной функциональности программы. Типичными примерами точек соединения служат вызов функции и определение структуры.

Срез (англ. *pointcut*) – это описание набора точек соединения, логически объединенных по некоторому условию. Например, с помощью среза можно описать все вызовы функций выделения памяти (таких как *malloc, calloc* и т.д.).

[2]Более точно семантику данного понятия передает термин “пронизывающий”, поскольку описание данной функциональности рассредоточивается по модулям программы, пронизывая их реализацию. Однако, в русскоязычной литературе достаточно устойчивым является термин “сквозная”.

Еще одним понятием АОП является **рекомендация** (англ. *advice*). Посредством рекомендации задается набор действий, которые должны быть выполнены для точек соединения, описанных срезом[3]. Например, для такой точки соединения, как вызов функции, в рекомендации могут быть записаны инструкции по выводу в журнал сообщения, содержащего значение, которое возвращает данная функция.

Посредством срезов и рекомендаций АОП позволяет выделить описание части сквозной функциональности программы в отдельные модули, так называемые **аспекты**[4] (англ. *aspect*). Далее в данном разделе будет приведен пример аспекта и рассмотрено, каким образом можно задавать срезы и рекомендации.

Помимо возможности описания части сквозной функциональности в виде аспектов АОП предполагает средства для автоматического связывания аспектов с целевой программой. По сути, данный процесс заключается в том, что для некоторого представления кода[5] программы идет поиск соответствия потенциальных точек соединения заданным в аспекте срезам. При обнаружении такого соответствия и наличии рекомендации для данного среза происходит обрамление точки соединения тем кодом, который задан в рекомендации. Поскольку процесс связывания аспектов с целевой программой имеет определенное сходство со стандартной компоновкой (англ. *linkage*) программ мы будем называть данный процесс **аспектной компоновкой**[6].

Чтобы реализовать аспектно-ориентированный подход программирования для некоторого языка программирования необходимо определить способы описания аспектов и разработать аспектный компоновщик. В настоящее время существует достаточно большое количество реализаций АОП для различных языков программирования: AspectJ [4] для Java, AspectC++ [8] для Си++, Aspect.NET [9] для Microsoft.NET, ACC [14, 15], InterAspect [16] и SLIC [18] для Си и т.д. В качестве примера рассмотрим AspectJ, одну из самых передовых и известных реализаций АОП на сегодняшний день.

Для разработки аспектов AspectJ предлагает одноименное аспектно-ориентированное расширение языка программирования Java [5]. На рис. 1 приведен аспект *Logging,* посредством которого для графической системы выделена сквозная функциональность, ведение журнала [6]. Для этого в аспекте задается именованный срез[7] *move*. Точки соединения, описываемые срезом move, – это вызовы метода *setXY* класса *FigureElement* и вызовы методов *setX* и *setY* класса *Point*. Кроме того, в аспекте *Logging* задается рекомендация. Данная рекомендация говорит, что перед *(before)* выполнением точек соединения, описанных именованным срезом *move* (вызовом соответствующих методов), должно быть напечатано сообщение на экран. Аспектная компоновка в AspectJ выполняется на уровне байт-кода программы с помощью *ajc,* специального компилятора языка Java и его расширения AspectJ [7]. Скомпонованный байт-код может интерпретироваться средствами стандартных виртуальных машин Java.

При разработке подхода к реализации аспектно-ориентированного программирования для языка Си активно использовался опыт AspectJ. Также мы приняли во внимание опыт существующих реализаций АОП для Си и Си++. В следующем разделе представлены хорошо устоявшиеся и применимые для различных языков программирования способы описания аспектов.

3. ТРАДИЦИОННЫЕ СПОСОБЫ ОПИСАНИЯ АСПЕКТОВ

Большинство существующих реализаций АОП для описания аспектов предлагают расширения тех языков программирования,

[3]Точнее говоря, некоторым образом должно быть выполнено обрамление или “декорирование” соответствующих точек соединения соответствующим кодом, который может содержать как инструкции, так и некоторые описания, декларации.

[4]Здесь термин “аспект” имеет узкотехническое значение, которое используется только в литературе по АОП.

[5]Забегая вперед отметим, что различные виды обработки, связанные с аспектами, можно проводить с различными видами представления программы, например, с текстовым, промежуточным, бинарным и др.

[6]В англоязычной литературе данный процесс обозначают посредством “*aspect weaving*”, что дословно переводится, как “аспектное переплетение” или “аспектное соединение”.

[7]Формально понятие именованный срез рассмотрено в следующем разделе.

```
// Аспект состоит из именованного среза и рекомендации.
aspect Logging {
  // Именованный срез 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 печатает сообщение на экран до
  // выполнения соответствующей данному именованному
  // срезу точки соединения программы.
  before(): move() {
    System.out.println("about to move");
  }
}
```

Рис. 1. Пример аспекта AspectJ, с помощью которого для графической системы выделена сквозная функциональность ведения журнала.

на которых пишутся целевые программы. В AspectJ, AspectC++ и Aspect.NET, как вероятно и в других реализациях АОП для объектно-ориентированных языков программирования, аспекты во многом схожи с классами. У аспектов могут быть собственные поля и методы, для них поддерживаются механизмы, аналогичные наследованию и полиморфизму классов. Реализации АОП для языка Си не поддерживают подобную абстракцию. Вместо этого они предлагают описывать аспекты в виде отдельных файлов. В том, что касается других понятий, реализации АОП для различных языков программирования более схожи.

Традиционно для точек соединения предлагается разбиение на статические и динамические. **Статические точки соединения** представляют собой декларации программы, такие как структура, класс, функция, метод, переменная, поле структуры или класса и т.д. **Динамические точки соединения** соответствуют событиям, которые могут произойти при выполнении программы. Примеры – это вызов функции или метода, присваивание значения переменной или полю, инициализация структуры или класса и т.д.

Для задания среза, который описывает набор статических точек соединения, традиционно используется сигнатура соответствующей сущности. При записи сигнатуры допускается использовать так называемые **групповые символы** (англ. *wildcards*). Групповые символы позволяют задавать соответствие любому типу, части имени сущности или произвольному списку параметров. В примере, приведенном на рис. 1, для описания методов *setX* и *setY* класса Point в AspectJ можно использовать сигнатуру “**Point.set**(..)”.

Как правило, срез, описывающий набор динамических точек соединения, задается добавлением ключевого слова к срезу, который описывает набор статических точек соединения. Данное ключевое слово отражает суть соответствующего события. Традиционно реализации АОП поддерживают следующие срезы, описывающие динамические события[8]:

- **call** – вызов функции;
- **execution** – выполнение функции, которое происходит после перехода управления из некоторой функции, вызывающей данную;
- **set** – присваивание значения переменной или полю;

[8]Начиная с данного момента, в тексте текущего раздела мы не будем рассматривать ничего, что связано с точками соединения, которые имеют чисто объектно-ориентированную природу и не имеют процедурных аналогов, поскольку целью данной статьи является разработка подхода к реализации АОП для языка Си.

- **get** – использование переменной или поля.

Стоит отметить, что традиционно АОП предоставляет возможность неявного задания точек соединения по их контексту, например:

- **infile** – все точки соединения из некоторого файла;
- **infunc** – все точки соединения из некоторой функции;
- **cflow** – все точки соединения, которые встречаются в контексте выполнения точек соединения другого среза.

Рассмотренные срезы для описания точек соединения называются **примитивными срезами.** Комбинации срезов, которые можно получить с помощью операторов "и" – пересечение, "или" – объединение, "не" – исключение и оператора группировки (скобок), называются **составными срезами.** Особую роль составные срезы играют при уточнении описания набора точек соединения, например, путем комбинирования явного и неявного способов задания точек соединения. **Именованный срез** представляет собой примитивный или составной срез, который связан с некоторым именем, посредством которого данный срез можно использовать. В примере на рис. 1 срез *move* является именованным срезом, который представляет составной срез, объединяющий три примитивных среза.

Рекомендация в АОП традиционно состоит из объявления и тела. **Объявление рекомендации** содержит срез и в случае, когда срез описывает динамический набор точек соединения, одно из ключевых слов *before, after* или *around.* Срез определяет условие, когда нужно применять рекомендацию. Ключевые слова указывают на то, как это надо делать: *before* – до, *after* – после, *around* – вместо[9]. В теле рекомендации записывается тот код, которым должны быть обрамлены соответствующие точки соединения. Как уже отмечалось, рекомендация из примера на рис. 1 говорит, что перед выполнением точек соединения, описанных именованным срезом *move,* должно быть напечатано сообщение на экран.

Как правило, тело рекомендации записывается с помощью инструкций того же языка программирования, на котором пишется целевая программа. Наряду с этим реализации АОП позволяют использовать в теле рекомендации специальные инструкции. Например, с помощью инструкции *proceed* можно записать вызов функции, соответствие которому задано срезом рекомендации. Посредством специальных инструкций в теле рекомендации можно использовать так называемую **рефлективную информацию** (англ. *reflection information*), которая представляет собой информацию о соответствующей точке соединения и ее контексте. Традиционно для точек соединения поддерживается следующая рефлективная информация:

1. Для вызовов и выполнений функций:
   - имя вызываемой функции;
   - тип возвращаемого значения;
   - типы и количество аргументов.
2. Для вызовов функций (дополнительно):
   - фактические параметры;
   - возвращаемое значение;
   - файл и функция, в контексте которых происходит вызов.
3. Для присваиваний и использований переменных и полей:
   - имя переменной или поля;
   - присваиваемое или текущее значение переменной или поля;
   - файл и функция, в контексте которых происходит присваивание или использование.

Один из важных вопросов, которые традиционно возникают при описании аспектов, – это порядок применения нескольких рекомендаций в том случае, когда их срезы соответствуют одной

[9] Дословно "around" переводится, как "вокруг". Однако, большинство реализаций АОП рассматривают *around* рекомендации, в первую очередь, для того, чтобы отменить выполнение соответствующей динамической точки соединения. В соответствии с этим мы предложили более адекватное название, хотя будем пользоваться словом *around.*

и той же точке соединения. Различные реализации АОП предлагают незначительно отличающиеся решения. Например, AspectJ предлагает следующий алгоритм применения нескольких рекомендаций для одной и той же точки соединения. Среди рекомендаций одного типа (*before, after, around*) первой применяется та, которая в коде аспекта встречается раньше. В первую очередь применяются *around* рекомендации следующим образом:

1. Если в теле текущей применяемой рекомендации нет специальной инструкции *proceed,* то ее применение завершается.

2. Иначе применяется часть данной рекомендации до инструкции proceed; вместо этой инструкции применяется следующая *around* рекомендация, если таковая имеется, либо алгоритм:

   - применяет все *before* рекомендации;
   - выполняет непосредственно саму точку соединения;
   - применяет все *after* рекомендации.

   После чего применяется оставшаяся часть данной рекомендации.

Для более сложных случаев, например, когда программа связывается сразу с несколькими аспектами, традиционно считается, что поведение аспектного компоновщика неопределенно.

Как правило, рекомендации для статических точек соединения рассматриваются отдельно от рекомендаций для динамических точек соединения. Достаточно много реализаций АОП предлагают возможность добавить поля в определения составных типов данных, таких как структуры и объединения. Особое значение задание рекомендаций для статических точек соединения играет для объектно-ориентированных языков, поскольку для них это позволяет описывать в аспектах сквозную функциональность, связанную с инкапсуляцией, наследованием и полиморфизмом.

Мы рассмотрели основные традиционные способы описания аспектов. Данные способы в той или иной мере поддерживаются в реализациях АОП для различных языков программирования. В разделе 6 будет показано, что существующие реализации АОП для языка Си не выходят за рамки поддержки традиционных способов описания аспектов. Однако, язык Си и процесс сборки Си-программ обладают несколькими особенностями, которые необходимо принимать во внимание при разработке реализации АОП.

4. ВЛИЯНИЕ ОСОБЕННОСТЕЙ ЯЗЫКА ПРОГРАММИРОВАНИЯ СИ И ПРОЦЕССА СБОРКИ СИ-ПРОГРАММ НА РЕАЛИЗАЦИЮ АОП

Ключевой особенностью языка программирования Си является то, что в этом языке поддерживается адресная арифметика. Особый интерес для АОП представляют переменные или поля, которые имеют тип указатель, а также различные операции с ними. Все то, что поддерживается реализациями АОП для переменных и полей, автоматическим образом переносится на указатели. Однако в качестве динамических точек соединения для указателей необходимо дополнительно поддержать такие операции, как взятие адреса переменной или поля, разыменование указателя, присваивание и использование элемента массива.

Еще одной особенностью языка программирования Си является механизм сборки программ на этом языке. Язык Си является одним из немногих языков программирования, для которых перед компиляцией и компоновкой кода программы происходит его предварительное препроцессирование на основе конфигурации, заданной файлами сборки и пользователем. Препроцессирование – это неотъемлемая особенность языка Си (например, в Java этого нет), поэтому эту стадию обработки исходного кода программы нужно учитывать при реализации АОП для Си. Основные действия, которые происходят во время этого процесса – это выполнение условной компиляции, включение текста заголовочных файлов и выполнение макроподстановок последовательно для всех файлов программы.

Включение текста заголовочных файлов может быть связано с понятиями АОП следующим образом. Каждый включаемый в некоторый файл заголовочный файл и непосредственно сам файл могут рассматриваться, как статические

точки соединения. В качестве сигнатуры, с помощью которой можно описать набор таких точек соединения, можно рассматривать шаблон пути, в котором допускается использовать групповой символ, обозначающий последовательность символов произвольной длины. Кроме того, возможно использовать специальное обозначение для текущего препроцессируемого файла. Благодаря этому появляется возможность выделять в аспектные файлы сквозную функциональность, связанную с включением заголовочных файлов. Это может быть использовано, например, для того, чтобы добавить некоторые вспомогательные директивы препроцессора (тем самым повлиять на конфигурацию программы), прототипы функций и т.п.

Макросы во многом схожи с функциями. Например, у макроса есть именованное параметризованное определение, а макроподстановка заключается в том, чтобы заменить используемые в определении формальные параметры на фактические и подставить получившийся результат в код программы. В силу этой аналогии макросы естественно считать статическими точками соединения. Для описания набора данных точек соединения можно использовать сигнатуру с поддержкой групповых символов, как для задания части имени макроса, так и для задания списка параметров произвольной длины. Опять же в силу аналогии с функциями для макросов можно рассматривать такие динамические точки соединения, как “выполнение” макроса и “вызов” (подстановку) макроса. Естественным образом переносятся способ задания этих точек соединения через их контекст *(infile)* и использование в рекомендациях рефлективной информации, такой как имя макроса, фактические параметры и контекст. С помощью задания рекомендаций для макросов можно вместо (или до, или после) подставляемого кода поместить некоторый дополнительный код.

Мы рассмотрели характерные особенности языка программирования Си и процесса сборки программ на данном языке, которые влияют на реализацию АОП для Си. Не менее важно принять во внимание особенности практического применения реализации АОП для языка Си, чему посвящается следующий раздел статьи.

5. ОСОБЕННОСТИ ПРАКТИЧЕСКОГО ПРИМЕНЕНИЯ РЕАЛИЗАЦИИ АОП ДЛЯ ЯЗЫКА СИ

На процесс разработки подхода к реализации АОП для Си в рамках данной работы сильно повлиял проект верификации драйверов Linux LDV [10–13]. Цель проекта LDV – предоставить инструментарий, который позволит использовать различные инструменты статического анализа кода, чтобы проверять удовлетворяют ли драйверы операционной системы Linux некоторому набору правил корректности.

Оказалось, что подходящим способом формализовать правила корректности, независимым от используемого инструмента статического анализа образом, и затем инструментировать исходный код драйверов, которые будут проверяться, является использование реализации АОП для языка Си.

Данный вариант практического применения потребовал от реализации АОП для языка Си следующее:

- Поддерживать язык программирования Си со всеми расширениями компилятора GCC в качестве входного языка (это стандартный язык для написания драйверов операционной системы Linux), а также поддерживать стандартные и GCC опции сборки программ на Си.

- Предлагать большой набор средств АОП, позволяющих выделять в аспекты сквозную функциональность программ на языке Си. Это требуется ввиду многообразия проверяемых правил корректности. В частности, для формализации правил корректности требуется использовать модельное состояние и модельные функции, которые по сути аналогичны собственным полям и методам аспектов реализаций АОП для объектно-ориентированных языков программирования (пример можно увидеть в разделе 8). При этом важно, чтобы использовать предлагаемые средства АОП для Си для разработки аспектов было достаточно удобно и интуитивно понятно.

- Аспектный компоновщик должен выдавать на выходе корректную программу на Си, эк-

```
// Рекомендация печатает сообщение и возвращаемое значение
// функции foo2 после ее вызова.
after (int res):
  call(int foo2(int)) && result(res) {
  printf(" after call foo2, return %d\n", res);
}
```

Рис. 2. Пример аспекта ACC.

вивалентную исходной с тем лишь исключением, что она расширена описанием соответствующей сквозной функциональности. Это требуется для последующего применения инструментов статического анализа кода.

- Реализация АОП должна быть достаточно легко поддерживаемой и расширяемой новыми возможностями. Это требование пришло из практики. Например, для формализации новых правил корректности иногда бывает необходима поддержка дополнительных видов точек соединения и рефлективной информации.

Нельзя не отметить, что сформулированные выше специфические требования можно рассматривать с точки зрения выделения сквозной функциональности для любой программы, написанной на языке Си, с поправкой, что это делается для Linux. Например, большинство программ под Linux требуют поддержки языка Си с расширениями GCC. Для эффективного выделения сквозной функциональности необходим достаточно большой набор средств АОП, а использовать реализацию АОП должно быть удобно. Требование того, что выход аспектного компоновщика должен быть программой на Си, также полезно для целей отладки работы реализации АОП.

6. СУЩЕСТВУЮЩИЕ РЕАЛИЗАЦИИ АОП ДЛЯ СИ

Методы реализации аспектно-ориентированного подхода для языка программирования Си разработаны в меньшей степени по сравнению с таковыми для Java. В настоящее время, самой передовой реализацией АОП для Си является ACC (AspeCt-oriented C) [14]. Расширение ACC для языка Си [15], используемое для разработки аспектов, похоже на расширение для Java, сделанное в AspectJ. На рис. 2 представлен пример аспекта, написанного с помощью данного расширения Си. Аспект состоит из одной рекомендации, которая говорит, что после всех вызовов функции *foo2* на экран должно быть напечатано сообщение, содержащее возвращаемое значение данной функции. Подход к аспектной компоновке ACC отличается от подхода, принятого в AspectJ. Для каждого подаваемого на вход препроцессированного файла специальный компилятор *acc* генерирует файл на Си, который расширен описанием соответствующей сквозной функциональности.

ACC поддерживает достаточно большой набор средств АОП для выделения сквозной функциональности. Из традиционных средств АОП инструмент не поддерживает динамические точки соединения для присваиваний и использований полей, а также порядок применения рекомендаций для одной и той же точки соединения в том случае, когда должны быть применены одновременно и *around,* и *before/after* рекомендации.

Что касается специфичных средств АОП для языка Си, ACC не позволяет влиять на процесс препроцессирования, поскольку он принимает на вход уже препроцессированный исходный код. Также ACC не предлагает способа описания в срезах операций с указателями.

При использовании ACC задавать модельное состояние и модельные функции необходимо вручную, поскольку инструмент не предоставляет таких возможностей. Инструмент выпускается под лицензией GNU General Public License версии 2, что позволяет дорабатывать его. При этом ACC использует парсер языка Си и его аспектно-ориентированного расширения,

```
static void instrument_malloc_calls() {
  /* Создание примитивного среза, задающего соответствие
       вызовам функции void *malloc(unsigned int). */
  struct aop_pointcut *pc =
    aop_match_function_call();
  aop_filter_call_pc_by_name(pc, "malloc");
  aop_filter_call_pc_by_param_type
    (pc, 0, aop_t_all_unsigned());
  aop_filter_call_pc_by_return_type
    (pc, aop_t_all_pointer());
  aop_join_on(pc, malloc_callback, NULL);
}
```

Рис. 3. Пример задания среза InterAspect.

который сгенерирован на основе закрытой грамматики. Данный парсер не может обработать некоторые расширения GCC. Поэтому, например, ACC не может быть применен для инструментирования исходного кода драйверов последних версий ядра ОС Linux. Для больших и сложных программ ACC генерирует на выходе Си-код, который не эквивалентен исходной программе, а в некоторых случаях является некорректным. Поддержка инструмента его разработчиками в настоящее время не ведется, а так как доработка используемого им парсера затруднена, данную реализацию АОП тяжело дорабатывать.

InterAspect – одна из самых новых реализаций АОП для языка программирования Си [16]. Инструмент интересен в первую очередь тем, что он основан на плагинах GCC [17], благодаря чему он поддерживает все расширения GCC. Для разработки аспектов вместо расширения языка Си InterAspect предоставляет специальную библиотеку АОП для Си. На рис. 3 показано, как с помощью данной библиотеки можно определить примитивный срез, описывающий вызовы функции *malloc.* Для этого необходимо создать соответствующий срез *(aop_match_function_call)* и задать ограничения на имя вызываемых функций *(aop_filter_call_pc_by_name),* на типы их аргументов *(aop_filter_call_pc_by_param_type)* и на тип возвращаемого ими значения *(aop_filter_call_pc_by_return_type),* что достаточно трудоемко.

InterAspect поддерживает значительное количество традиционных средств АОП. К наиболее существенным недостаткам инструмента относится отсутствие поддержки *around* рекомендаций. Также InterAspect не позволяет задавать точки соединения по их контексту, не поддерживает описание составных срезов, предоставляет недостаточно большое количество рефлективной информации о точке соединения и не позволяет задавать рекомендации для статических точек соединения.

Так же, как и ACC, InterAspect не поддерживает средств АОП, связанных с особенностями языка Си и сборки программ на данном языке.

InterAspect работает на уровне низкоуровневого внутреннего представления компилятора GCC, расширяя его описанием сквозной функциональности. Выход инструмента, объектный или бинарный код, генерируется GCC, что не позволяет использовать InterAspect напрямую для статического анализа, а также неудобно проводить отладку инструмента. Разработка InterAspect ведется неактивно. Инструмент выпускается под лицензией GNU General Public License версии 3, что позволяет расширять его функциональность.

Один из наиболее перспективных с точки зрения инструментирования подходов реализован в языке SLIC (язык спецификаций для проверки интерфейсов) и его препроцессоре [18]. SLIC представляет собой Си подобный язык, с помощью которого разрабатываются спецификации. Спецификация SLIC, на самом деле, есть

не что иное, как аспект. В телах рекомендаций SLIC допускается использовать условные выражения, простые присваивания, ссылки на параметры функций и их возвращаемое значение, а также специальные инструкции для инструмента статического анализа кода. SLIC поддерживает задание модельного состояния. Приведенная на рис. 4 спецификация SLIC демонстрирует возможность проверки ошибок в программах с помощью АОП. В данном примере ошибочной является ситуация, когда в очереди больше четырех нулей. В спецификации определяется модельное состояние программы, *zero_ cnt,* которое служит для подсчета количества нулей в очереди. Задается рекомендация для входа в функцию *put,* которая проверяет количество нулей в очереди и увеличивает *zero_ cnt* в том случае, если не произошла ошибка. Аналогичным образом задается рекомендация для выхода из функции *get.*

```
state { int zero_cnt = 0; }
put.entry {
  if ($1 == 0) {
    if (zero_cnt == 4)
      abort "Queue has 4 zeroes!";
    else
      zero_cnt = zero_cnt + 1;
  }
}
get.exit {
  if ($return == 0)
    zero_cnt = zero_cnt - 1;
}
```

Рис. 4. Пример спецификации SLIC.

SLIC используется в процессе верификации драйверов операционной системы Microsoft Windows. Препроцессор SLIC инструментирует исходный код драйверов на основе спецификации. На выходе препроцессор генерирует эквивалентную оригинальной программу на Си, которая дополнена описанием требуемых проверок. В последствие данная программа проверяется с помощью инструмента статического анализа кода.

В силу специфики своей области применения SLIC никогда не был нацелен на то, чтобы быть полноценной реализацией АОП для Си. В инструменте реализована поддержка всего нескольких динамических точек соединения (на самом деле, только вызов и выполнение функции). SLIC не позволяет задавать точки соединения по их контексту, не поддерживает именованные срезы, а из составных срезов поддерживает только “или”. Инструмент не предлагает *around* рекомендации, а в телах рекомендаций позволяет использовать небольшое количество рефлективной информации. SLIC не позволяет задавать несколько рекомендаций для одной и той же точки соединения и не поддерживает рекомендации для статических точек соединения.

Подобно ACC и InterAspect, SLIC не поддерживает средств АОП, связанных с особенностями языка Си и сборки программ на данном языке.

Исходный код препроцессора SLIC закрыт в отличие от InterAspect и ACC, что, в частности, не позволяет расширить область его возможного применения для выделения сквозной функциональности программ на Си с расширениями GCC. Как уже было отмечено, инструмент позволяет задавать модельное состояние. Модельные функции не поддерживаются. Поскольку инструмент используется достаточно активно для решения важных индустриальных задач, можно сделать предположение, что поддержка SLIC ведется на достаточно высоком уровне.

В данной статье не рассматриваются другие реализации АОП для Си, например, C4, Aspicere2, Xweaver, WeaveC, поскольку они обладают меньшим количеством полезных особенностей по сравнению с подходами, описанными выше. Также в обзоре не представлены инструменты АОП для языка программирования Си++. Данные инструменты могут быть приспособлены для выделения сквозной функциональности программ на языке Си, но для разработки аспектов и выполнения аспектной компоновки в них активно используются понятия и средства объектно-ориентированного программирования.

Таким образом, существующие реализации АОП для языка Си предоставляют далеко не все из рассмотренных возможностей АОП. Для большей части реализаций имеются принципиальные ограничения на то, чтобы добавить в них поддержку некоторых возможностей. Этот вывод привел нас к необходимости разработки собственной реализации АОП для языка Си.

Таблица 1. Сравнение C Instrumentation Framework с существующими реализациями АОП для языка программирования Си.

Характеристики реализации АОП		ACC	InterAspect	SLIC	CIF
Поддержка традиционных средств АОП	Срезы для статических точек соединения, в том числе поддержка групповых символов	+	+	±	+
	Срезы для динамических точек соединения	±	±	±	+
	Срезы для описания точек соединения по их контексту	+	−	−	+
	Составные и именованные срезы	+	±	±	+
	Рекомендации *before*, *after* и *around* для динамических точек соединения	+	±	±	+
	Произвольный корректный Си-код в телах рекомендаций	+	+	±	+
	Специальные инструкции в теле рекомендации, в том числе рефлективная информация	+	±	±	+
	Порядок применения рекомендаций	±	±	−	+
	Рекомендации для статических точек соединения	+	−	−	+
Поддержка средств АОП для Си	Срезы на включение заголовочных файлов	−	−	−	+
	Срезы на „выполнение“ и подстановки макросов	−	−	−	+
	Срезы на операции с указателями	−	−	−	−
Возможности практического применения	Си с расширениями GCC, в том числе GCC опции сборки	±	+	±	+
	Модельное состояние и модельные функции	−	+	±	+
	Выход аспектного компоновщика на Си	±	−	±	+
	Расширение возможностей реализации	±	+	−	+
	Поддержка реализации	−	±	+	+

7. C INSTRUMENTATION FRAMEWORK – РЕАЛИЗАЦИЯ АОП ДЛЯ ЯЗЫКА СИ

Предложенный в рамках данной работы подход к реализации аспектно-ориентированного программирования для языка Си был реализован в инструменте C Instrumentation Framework (CIF). Для разработки аспектов CIF предлагает расширение языка Си наподобие того, как это делают AspectJ, AspectC++, ACC, SLIC и множество других реализаций АОП для различных языков программирования. Аспекты сохраняются в виде файлов отдельно от исходного кода программы. Пример аспектного файла CIF можно найти в следующем разделе.

CIF поддерживает большую часть средств АОП для Си, выделенных в предложенном в данной статье подходе. Поддерживаются все традиционные средства АОП, а также возможность задания срезов на включение заголовочных файлов, на “выполнения” и подстановки макросов. На текущий момент так же, как и в других реализациях АОП для Си, инструмент не поддерживает срезы на операции с указателями.

Большое внимание при разработке CIF было уделено возможности его практического применения, в особенности, для инструментирования исходного кода драйверов операционной

системы Linux перед проведением статического анализа кода. Благодаря предложенной архитектуре инструмент может принять на вход исходный код программы на Си со всеми расширениями GCC, а также поддерживает стандартные и GCC опции сборки программ на Си. После выполнения аспектной компоновки CIF может сгенерировать на выходе как Си код, который, например, можно передать инструменту статического анализа кода, так и те представления кода программы, которые поддерживает компилятор GCC.

В табл. 1 сравниваются поддерживаемые возможности CIF и существующих реализаций АОП для Си, которые были описаны в разделе 6. Как видно, CIF поддерживает практически все перечисленные возможности.

Рассмотрим архитектуру C Instrumentation Framework более подробно. Первоначально архитектура основывалась на инфраструктуре LLVM [20]. Оказалось, что такой подход имеет ряд ограничений, в особенности, с точки зрения практического применения. Поэтому в данной работе была предложена новая архитектура.

На вход CIF подаются: аспектный файл, непрепроцессированный файл программы на Си и набор опций препроцессирования и компиляции[10]. Дальнейшая работа инструмента по сути аналогична работе стандартных препроцессора и компилятора языка Си, за исключением того, что CIF дополнительно выполняет аспектную компоновку исходного кода. Аспектная компоновка осуществляется автоматическим образом в течение пяти этапов. На каждом из этапов CIF вызывает с соответствующими опциями модифицированную версию GCC 4.6.1, который включает в себя препроцессор и компилятор языка Си [21]. В модифицированную версию GCC были встроены:

- парсер аспектов, который проверяет лексическую, синтаксическую и семантическую корректность аспектных файлов и преобразует их во внутреннее представление CIF (тела рекомендаций разбираются только с целью определения в них специальных инструкций);
- компонент, перехватывающий обработанные препроцессором и компилятором GCC конструкции (точки соединения программы) и преобразующий их во внутреннее представление CIF;
- компонент, который сопоставляет точки соединения и срезы заданных в аспекте рекомендаций;
- компонент, выполняющий обрамление сопоставленных точек соединения на основе требований рекомендаций;
- компонент, позволяющий напечатать Си-код на основе высокоуровневого внутреннего представления GCC.

Рассмотрим подробнее, каким образом происходит аспектная компоновка на каждом из пяти этапов и как при этом используются входные данные CIF. Для удобства обозначим подаваемые на вход аспектный файл, как *a.aspect,* а непрепроцессированный файл программы на Си, как *a.c.*

Первый этап является вспомогательным. На этом этапе выполняется препроцессирование аспектного файла *a.aspect* средствами препроцессора GCC. При этом интерпретируются директивы, начинающиеся с символа “@”, а не стандартного символа “#”. Задать опции аспектного препроцессирования можно с помощью опции CIF *--aspect-preprocessing-opts.* Получаемый на выходе файл обозначим, как *a.aspect.i.* Препроцессирование аспектных файлов можно использовать, например, для разработки библиотеки аспектных файлов, что является актуальной задачей при широком практическом применении.

На втором этапе CIF вставляет дополнительный текст до или посрепроцессора. Обозначим выходной файл, как *a.prepared.*

На третьем этапе CIF запускает стандартный препроцессор GCC, которому передаются на вход файл *a.prepared* и опции препроцессирования. По мере выполнения препроцессирование происходит аспектная компоновка “выполнений” и подстановок макросов в соответствии

[10]Непрепроцессированный файл программы и набор опций препроцессирования и компиляции могут быть получены, например, на основе файлов, описывающих сборку программы.

с рекомендациями аспектного файла *a.aspect.i.* При этом на основе полученной информации о точке соединения в телах рекомендаций заменяются специальные инструкции. Остальной код тел рекомендаций используется как есть. Получаемый на выходе препроцессированный файл обозначим *a.macroweaved.*

На четвертом этапе CIF подает на вход стандартному компилятору GCC файл *a.macroweaved* и опции компиляции. На этом этапе компилятор выполняет разбор поданного на вход файла и переводит его в свое внутреннее высокоуровневое представление. При этом специальный компонент GCC перехватывает обработанные определения и вызовы функций, присваивания и использования переменных и полей, а также декларации составных типов данных. В случае обнаружения соответствия между ними и заданными в аспектном файле *a.aspect.i* рекомендациями, составные типы данных дополняются полями на уровне текстового представления файла *a.macroweaved,* а для функций, переменных и полей создаются вспомогательные функции, которые дописываются в конец файла *a.macroweaved.* Телами вспомогательных функций становятся тела соответствующих рекомендаций. Специальные инструкции в коде тел рекомендации заменяются на их значения на основе получаемой информации о точке соединения. Стоит отметить, что в телах рекомендаций допускается использовать произвольный корректный код на языке программирования Си с расширениями GCC. Разбор и проверка этого кода компилятором GCC происходит на следующей стадии работы CIF. Обозначим получаемый в результате работы на данной стадии файл, как *a.weaved.*

На пятом заключительном этапе CIF подает на вход стандартному компилятору GCC файл *a.weaved* и опции компиляции. Дальнейшая работа в целом аналогична работе на четвертом этапе за исключением того, что декларации составных типов данных не перехватываются, а для функций, переменных и полей происходит связывание соответствующих динамических точек соединений и вспомогательных функций на уровне высокоуровневого внутреннего представления GCC. Если CIF запускается с опцией *--back-end=src,* то по мере того, как компилятор завершает разбор конструкций, их высокоуровневое представление преобразуется в Си-код, которые печатается в указанный с помощью опции выходной файл. Следует отметить, что при этом поддерживается взаимосвязь с оригинальным исходным кодом программы путем добавления директив line. Если значение опции CIF *--back-end* одно из *asm, obj* или *bin,* компилятор GCC продолжает свою работу стандартным образом и выдает на выход одно из представлений кода программы, которые поддерживает компилятор GCC (ассемблерный, объектный или бинарный код соответственно). Выходной файл обозначим, как *a.o.*

На рис. 5 показана схема работы C Instrumentation Framework, которая демонстрирует наглядным образом, как используются и преобразуются подаваемые на вход данные. Прямоугольниками с пунктирной границей на схеме выделены внешние данные.

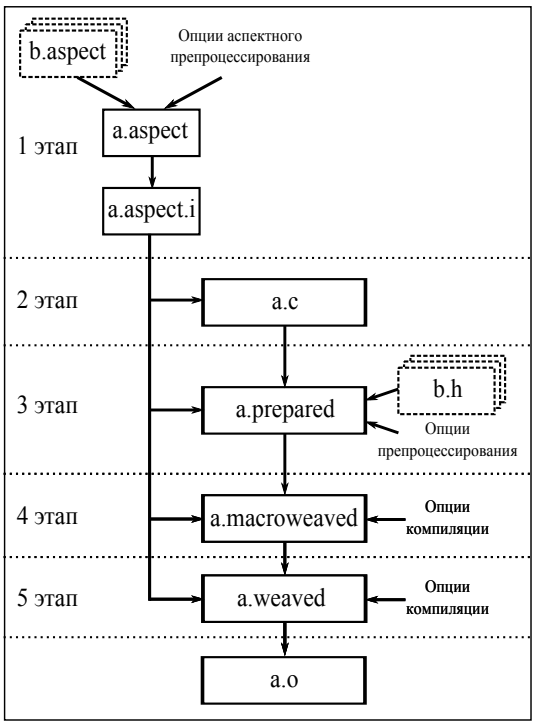

Рис. 5. Схема работы C Instrumentation Framework.

8. ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ ПРЕДЛОЖЕННОЙ РЕАЛИЗАЦИИ АОП ДЛЯ СИ

Предложенная реализация АОП для язы-

ка Си C Instrumentation Framework входит в инструментарий проекта верификации драйверов операционной системы Linux LDV [10–13]. Правила LDV, которые описывают корректное использование интерфейса ядра Linux драйверами, вручную формализуются в виде аспектов следующим образом:

1. Анализируется та часть интерфейса ядра Linux для драйверов, которая имеет отношение к данному правилу. В большинстве случаев при этом выделяются макросы и функции, корректное использование которых описывает правило.

2. В аспекте описываются модельное состояние и модельные функции, в которых выполняются требуемые правилом проверки и соответствующие изменения модельного состояния.

3. С помощью рекомендаций в аспекте описывается привязка модельных функций к тем местам, где используется соответствующий им интерфейс ядра Linux.

4. В различных частях аспекта описываются вспомогательные конструкции. CIF используется для автоматического инструментирования исходного кода драйверов с целью его последующей проверки с помощью инструментов статического анализа кода.

В качестве примера рассмотрим одно из правил LDV, которое описывает корректную регистрацию устройства универсальной последовательной шины класса Gadget. При несоблюдении данного правила может произойти крах всей системы [22]. Правило говорит следующее:

1. В первую очередь необходимо в произвольном порядке зарегистрировать класс устройств и диапазон номеров устройств, а затем зарегистрировать устройство класса Gadget. Повторная регистрация ресурсов запрещается.

2. После завершения работы с устройством нужно сначала дерегистрировать устройство класса Gadget, а затем в произвольном порядке – класс устройств и диапазон номеров устройств.

При формализации правила нужно учесть, что функции регистрации могут не срабатывать по тем или иным причинам. В таком случае также необходимо дерегистрировать зарегистрированные ресурсы. Например, если не удается зарегистрировать устройство класса Gadget, нужно в произвольном порядке дерегистрировать класс устройств и диапазон номеров устройств.

В результате анализа интерфейса ядра Linux были выявлены следующие макросы и функции, которые имеют отношение к рассматриваемому правилу:

1. Макрос “$class\_create(owner, name)$” – создание и регистрация класса устройств.

2. Функция “$void\ \ class\_destroy(struct\ class\ *)$” – дерегистрация и уничтожение класса устройств.

3. Макрос “$class\_register(class)$” – регистрация класса устройств.

4. Функция “$void\ class\_unregister(struct\ class\ *)$” – дерегистрация класса устройств.

5. Функции “$int\ alloc\_chrdev\_region(dev\_t\ *,\ unsigned,\ unsigned,\ const\ char\ *)$” и “$int\ register\_chrdev\_region(dev\_t,\ unsigned,\ const\ char\ *)$” – регистрация диапазона номеров устройств.

6. Функция “$void\ unregister\_chrdev\_region\ (dev\_t,\ unsigned)$” – дерегистрация диапазона номеров устройств.

7. Функция “$int\ usb\_gadget\_register\_driver\ (struct\ usb\_gadget\_driver\ *driver)$” – регистрация устройства класса Gadget (до версии ядра Linux 2.6.37).

8. Функция “$int\ usb\_gadget\_probe\_driver\ (struct\ usb\_gadget\_driver\ *\ driver,\ int\ (*bind)\ (struct\ usb\_gadget\ *))$” – регистрация устройства класса Gadget (начиная с версии ядра Linux 2.6.37).

9. Функция “$int\ usb\_gadget\_unregister\_driver\ (struct\ usb\_gadget\_driver\ *\ driver)$” – дерегистрация устройства класса Gadget.

```
32  new: file(LDV_COMMON_MODEL) {
33  #include <verifier/rcv.h>
34  #include <kernel-model/err.h>
35  enum {
36    LDV_CLASS_UNREGISTERED,
37    LDV_CLASS_REGISTERED
38  };
39  // Аналогично для диапазона номеров устройств и устройства
    // класса Gadget.
40  static int ldv_class = LDV_CLASS_UNREGISTERED;
41  static int ldv_region = LDV_REGION_UNREGISTERED;
42  static int ldv_usb_gadget = LDV_USB_GADGET_UNREGISTERED;
43  void *ldv_create_class(void)
44  {
45    void *is_got;
46    is_got = ldv_undef_ptr();
47    if (is_got <= LDV_PTR_MAX)
48    {
49      ldv_assert(ldv_usb_gadget == LDV_USB_GADGET_UNREGISTERED);
50      ldv_assert(ldv_class == LDV_CLASS_UNREGISTERED);
51      ldv_class = LDV_CLASS_REGISTERED;
52    }
53    return is_got;
54  }
55  int ldv_register_class(void)
56  {
57    int is_reg;
58    is_reg = ldv_undef_int_nonpositive();
59    if (!is_reg)
60    {
61      ldv_assert(ldv_usb_gadget == LDV_USB_GADGET_UNREGISTERED);
62      ldv_assert(ldv_class == LDV_CLASS_UNREGISTERED);
63      ldv_class = LDV_CLASS_REGISTERED;
64    }
65    return is_reg;
66  }
67  // Аналогично для ldv_register_region и
    // ldv_register_usb_gadget.
68  void ldv_unregister_class(void)
69  {
70    ldv_assert(ldv_usb_gadget == LDV_USB_GADGET_UNREGISTERED);
71    ldv_assert(ldv_class == LDV_CLASS_REGISTERED);
72    ldv_class = LDV_CLASS_UNREGISTERED;
73  }
74  // Аналогично для ldv_unregister_region и
    // ldv_unregister_usb_gadget.
75  void ldv_check_final_state(void)
76  {
77    ldv_assert(ldv_class == LDV_CLASS_UNREGISTERED);
78    ldv_assert(ldv_region == LDV_REGION_UNREGISTERED);
79    ldv_assert(ldv_usb_gadget == LDV_USB_GADGET_UNREGISTERED);
80  }
81  }
```

Рис. 6. Модельное состояние и модельные функции для правила LDV, которое описывает корректную регистрацию устройства универсальной последовательной шины класса Gadget.

На рис. 6 представлена часть аспектного файла, в которой задаются модельное состояние и модельные функции (опущены комментарии и фрагменты аналогичного кода). Модельное состояния и модельные функции выносятся в отдельный файл с помощью специальной рекомендации *new* (32 строка) для того, чтобы они не повторялись для нескольких файлов с Си-кодом, которые могут быть скомпонованы вместе. В данном случае этот файл задается посредством переменной окружения LDV_COMMON_MODEL.

В строке 33 включается вспомогательный заголовочный файл, в котором определяется интерфейс инструмента статического анализа кода:

- Функции *ldv_under_ptr* (используется в строке 46) и *ldv_undef_int_nonpositive* (используется в строке 58) возвращают недетерминированные указатель и неположительное целое число соответственно. Данные функции необходимы для того, чтобы моделировать возможные сбои регистрации ресурсов.

- Макрос *ldv_assert* (используется в строках 49, 50 и т.д.) представляет собой проверку условия, которое передается ему в виде параметра. В том случае, если инструмент статического анализа кода обнаружит нарушение данного условия, он сообщит о возможной ошибке.

В строке 34 включается вспомогательный заголовочный файл, в котором определяются модельные функции для обработки ошибок, передающихся посредством возвращаемых значений функций типа указатель. Например, в строке 47 используется специальное макроопределение *LDV_PTR_MAX*, с помощью которого задается максимально возможное значение указателя, не несущего информацию об ошибке.

В строках 35–38 задаются возможные значения модельной переменной *ldv_class*: *LDV_CLASS_UNREGISTER* соответствует состоянию, при котором класс устройств не зарегистрирован; *LDV_CLASS_REGISTER* – зарегистрирован.

В строках 40–42 определяются и инициализируются переменные, которые представляют модельное состояние. В начале выполнения программы все ресурсы находятся в незарегистрированном состоянии.

В строках 43–54 определяется модельная функция *ldv_create_class*, которая соответствует макросу интерфейса ядра Linux *class_create*, отвечающему за создание и регистрацию класса устройств. *ldv_create_class* моделирует возможный сбой в регистрации класса устройств (строки 45–47, 53) и проверяет, что устройство класса Gadget не было зарегистрировано до регистрации класса устройств (строка 49) и что регистрация класса устройств не проходит повторно (строка 50). В том случае, если проверки проходят, происходит изменение модельного состояния, а именно переменная *ldv_class* устанавливается в значение *LDV_CLASS_REGISTER*.

Макросу *class_register* соответствует модельная функция *ldv_register_class* (строки 55–66). В целом данная функция аналогична *ldv_create_class* за исключением того, что возможный сбой в регистрации класса устройств моделируется с помощью целых чисел, а не указателей (строки 57–59, 65).

В строках 68–73 определяется модельная функция *ldv_unregister_class*, которая соответствует функциям интерфейса ядра Linux *class_destroy* и *class_unregister*. В *ldv_unregister_class* проверяется, что устройство класса Gadget было дерегистрировано и что класс устройств находится в зарегистрированном состоянии. После этого происходит дерегистрация класса устройств.

В строках 75–80 определяется специальная модельная функция, которая выполняется при завершении работы драйвера. В данной функции проверяется, что в конечном состоянии все ресурсы находятся в дерегистрированном состоянии.

На рис. 7 представлена часть аспектного файла, описывающая привязку модельных функций (строки 12, 15 и т.д.) к “выполнениям” макросов (строки 11 и 14) и вызовам функций (строки 17, 20 и т.д.). Для данного правила привязку к выполнению функций осуществить нельзя, поскольку реализации этих функций недоступны

```
11  around: define(class_create(owner, name)) {
12    ldv_create_class()
13  }
14  around: define(class_register(class)) {
15    ldv_register_class()
16  }
17  around: call(void class_destroy(..)) ||
            call(void class_unregister(..)) {
18    ldv_unregister_class();
19  }
20  around: call(int alloc_chrdev_region(..)) ||
            call(int register_chrdev_region(..)) {
21    return ldv_register_region();
22  }
23  around: call(void unregister_chrdev_region(..)) {
24    ldv_unregister_region();
25  }
26  around: call(int usb_gadget_register_driver(..)) ||
            call(int usb_gadget_probe_driver(..)) {
27    return ldv_register_usb_gadget();
28  }
29  around: call(int usb_gadget_unregister_driver(..)) {
30    ldv_unregister_usb_gadget();
31  }
```

Рис. 7. Привязка модельных функций к “выполнениям” макросов и вызовам функций интерфейса ядра Linux.

```
01  @include <kernel-model/err.aspect>
02  before: file("$this") {
03  void *ldv_create_class(void);
04  int ldv_register_class(void);
05  void ldv_unregister_class(void);
06  int ldv_register_region(void);
07  void ldv_unregister_region(void);
08  int ldv_register_usb_gadget(void);
09  void ldv_unregister_usb_gadget(void);
10  }
```

Рис. 8. Вспомогательные включение аспектного заголовочного файла и определение прототипов модельных функций.

при статическом анализе исходного кода драйвера.

Отметим, что посредством использования составного среза, который описывает вызовы функций *class_destroy* и *class_unregister* (строка 17), вместо них можно вызвать одну и ту же модельную функцию *ldv_unregister_class* (строка 18). Аналогичный прием используется для вызовов функций *alloc_chrdev_region* и *register_chrdev_region*, а также *usb_gadget_register_driver* и *usb_gadget_probe_driver*. Последнее, в частности, позволяет использовать данный аспектный файл для инструментирования исходного кода драйверов ядра операционной системы Linux различных версий (как до 2.6.37, так и после).

На рис. 8 представлена вспомогательная часть аспектного файла. В строке 01 включается специальный аспектный заголовочный файл, в котором описана привязка модельных функций для обработки ошибок, передающихся посредством возвращаемых значений функций типа указатель, к выполнениям соответствующих функций интерфейса ядра Linux. В

строке 02 говорится, что перед кодом инструментируемого драйвера необходимо поместить прототипы модельных функций (строки 03–09). Это требуется, потому что перед использованием функций они должны быть определены либо для них должны быть предоставлены прототипы. Поскольку модельные функции определяются в отдельном файле, для них предоставляются прототипы. Помимо этого в разных частях аспектного файла, приведенных на рис. 6–8, записываются вспомогательные модельные комментарии, которые опущены для краткости.

На текущий момент в виде аспектов было формализовано 44 правила проекта LDV. На основе данных аспектов с помощью C Instrumentation Framework был успешно инструментирован исходный код более 95% драйверов (всего их порядка 2–4 тысяч в зависимости от версии ядра) ядра операционной системы Linux версий от 2.6.31.6 до 3.7-rc4 [23]. Оставшиеся драйверы не были обработаны из-за технических проблем в компоненте GCC, который печатает Си-код на основе высокоуровневого внутреннего представления компилятора. Поскольку данный компонент разрабатывается в рамках данной работы, потенциально эти проблемы можно устранить.

Инструментированный код проверялся инструментами статической верификации BLAST [24] и CPAchecker [25]. В результате было выявлено 75 реальных ошибок в драйверах операционной системы Linux, о которых было сообщено разработчикам ядра [26]. Все эти ошибки были исправлены.

9. ЗАКЛЮЧЕНИЕ

Аспектно-ориентированное программирование предлагает специальный механизм для пополнения средств поддержки модульности, которые есть в существующих языках программирования. Изначально АОП разрабатывалось для объектно-ориентированных языков программирования, в первую очередь, Java. В статье исследуются подходы к реализации аспектно-ориентированного программирования для языка Си. Для этого рассматриваются традиционные способы описания аспектов для различных языков программирования, а также то, как особенности Си и процесса сборки программ на данном языке влияют на реализацию АОП. Особое внимание уделяется возможности применения реализации АОП для языка Си на практике, в особенности, для инструментирования исходного кода драйверов операционной системы Linux перед проведением статического анализа. В статье описываются возможности и недостатки существующих решений. Предлагается новая реализация АОП для Си C Instrumentation Framework, которая превосходит существующие реализации по всем показателям.

Дальнейшее развитие предложенной реализации АОП для языка программирования Си будет нацелено на улучшение поддержки средств АОП специфичных для Си, в особенности, операций с указателями; а также расширение области практического применения.

Информацию об актуальной версии C Instrumentation Framework можно получить на сайте проекта [19].

СПИСОК ЛИТЕРАТУРЫ

1. *Parnas D.L.* On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM. December, 1972. V. 15. № 12. P. 1053–1058.

2. *Фуксман А.Л.* Технологические аспекты создания программных систем. М.: Статистика, 1979.

3. Определения понятий аспектно-ориентированного программирования. http://www.aosd.net/wiki/ /index.php?title=Glossary

4. *Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W.G.* An Overview of AspectJ. In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01). 2001. P. 327–353.

5. AspectJ: аспектно-ориентированное расширение к языку программирования Java. http://www.eclipse.org/aspectj/doc/released/ /progguide/language.html

6. Введение в AspectJ. http://eclipse.org/aspectj/doc/released/progguide/ /starting-aspectj.html

7. ajc: компилятор языка программирования Java и его расширения AspectJ. http://www.eclipse.org/aspectj/doc/released/ /devguide/ajc-ref.html

8. *Spinczyk O., Lohmann D., Urban M.* AspectC++: an AOP extension for C++ // Software Developer's J. May, 2005. P. 68–76.

9. *Сафонов В.О.* Aspect.NET – инструмент аспектно-ориентированного программирования для разработки надежных и безопасных программ. Компьютерные инструменты в образовании. СПб.: Изд-во ЦПО "Информатизация образования". 2007. № 5. С. 3–13.

10. *Khoroshilov A., Mutilin V., Shcherbina V., Strikov O., Vinogradov S., Zakharov V.* How to cook an automated system for Linux driver verication. 2nd Spring Young Researchers' Colloquium on Software Engineering, St. Petersburg, 2008. V. 2. P. 10–14.

11. *Khoroshilov A., Mutilin V., Petrenko A., Zakharov V.* Establishing Linux driver verification process. Perspectives of Systems Informatics, Novosibirsk, 2010. V. 5947. P. 165–176.

12. *Мутилин В.С., Новиков Е.М., Страх А.В., Хорошилов А.В., Швед П.Е.* Архитектура Linux Driver Verification. Труды Института системного программирования РАН. 2011. Т. 20. С. 163–187.

13. *Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A.* Towards an open framework for C verification tools benchmarking. Proceedings of PSI, 2011. P. 82–91.

14. *Gong M., Zhang C., Jacobsen H.-A.* AspeCt-oriented C. Technology Showcase, CASCON 2007, Markahm, Ontario, 2007.

15. *Gong W., Jacobsen H.-A.* AspeCt-oriented C Language Spefication Version 0.8. University of Toronto, 2008.

16. *Seyster J., Dixit K., Huang X., Grosul R., Havelund K., Smolka S.A., Stoller S.D., Zadok E.* Aspect-Oriented Instrumentation with GCC. First International Conference on Runtime Verification, Malta, 2010. V. 6418. P. 405–420.

17. Плагины GCC. http://gcc.gnu.org/wiki/plugins

18. *Ball T., Rajamani S.K.* SLIC: a Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2002.

19. CIF: реализация аспектно-ориентированного программирования для языка Си. http://forge.ispras.ru/projects/cif

20. *Novikov E.* One Approach to Aspect-Oriented Programming Implementation for the C programming language. Proceedings of the 5th Spring/Summer Young Researchers' Colloquium on Software Engineering, 12–13 May, 2011, Yekaterinburg. P. 74–81. item GNU Compiler Collection 4.6.1. http://gcc.gnu.org/onlinedocs/gcc-4.6.1/gcc/

21. Исправление ошибки в драйвере устройства универсальной последовательной шины класса Gadget. http://marc.info/?l=linux-usbm=129649764609408

22. Ядро операционной системы Linux. http://kernel.org

23. *Beyer D., Henzinger T.A., Jhala R., Majumdar R.* The software model checker Blast: Applications to software engineering. Int. J. Softw. 2007. V. 9(5). P. 505–525.

24. *Beyer D., Keremoglu M.E.* CPAchecker: A Tool for Configurable Software Verification / G. Gopalakrishnan, S. Qadeer (eds.). CAV 2011, LNCS. Springer, Heidelberg, 2011. V. 6806. P. 184–190.

25. Список ошибок в драйверах опера ионной системы Linux, обнаруженных в рамках проекта LDV. http://linuxtesting.org/results/ldv