УДК 004.052

Проверка корректности освобождения ресурсов, локальных для функции на языке С

© А.В. Медников, В.А. Крищенко

МГТУ им. Н.Э. Баумана, Москва, 105005, Россия

Если перемененные, хранящие идентификатор полученного ресурса, являются локальными для некоторой функции, то ресурс должен быть свободен до момента ее завершения, иначе его освобождение произойдет только при уничтожении процесса. Оно также должно осуществляться в соответствии с жизненным циклом идентификатора ресурса. Для проверки корректности освобождения локальных ресурсов предложен алгоритм, основанный на статическом анализе ГПУ функции. Он позволяет обнаруживать утечку таких локальных для функции ресурсов, а также попытку повторного освобождения, использования после освобождения и попытку освобождения невыделенного ресурса. Программная реализация предложенного алгоритма использует ГПУ функции на языке С, полученный компилятором GCC. Разработанное программное обеспечение может обнаруживать ошибки, связанные с освобождением локальных ресурсов.

Ключевые слова: *обнаружение утечки ресурсов, ошибки использования ресурсов, обработка ошибок, статический анализ, С, GCC, ГПУ.*

Если процесс не освобождает своевременно ресурс, выделенный ему операционной системой, то момент освобождения такого ресурса откладывается до завершения процесса. Для системных служб данная ситуация не является приемлемой: типичная сетевая служба функционирует от момента запуска сервера до его выключения.

Язык программирования С является одним из наиболее популярных при создании сетевых служб. При этом он не предоставляет программисту ни аналога подхода RAII [1] для автоматического освобождения ресурсов, локальных для функции, ни синтаксиса для обработки исключений. Обработка ошибок на языке С выполняется при помощи условного оператора, часто в связи с оператором перехода как ближайшего аналога блока **try ... finally** в других языках программирования. В результате, освобождение ресурсов в программе на языке С возложено на программиста, и с ростом числа используемых в функции ресурсов возрастает и количество потенциальных ошибок.

Если все переменные, хранящие идентификатор такого ресурса, локальны для функции, то можно утверждать, что ресурс должен быть освобожден к моменту ее завершения, иначе наблюдается ситуация утечки ресурса. Автоматизация проверки этого критерия позволит повысить качество создаваемого ПО. В настоящий момент суще-

ствуют многочисленные средства, обнаруживающие утечки памяти как на основе динамического (например, [2]), так и статического анализа (например, CppCheck). Поскольку эти средства ограничены поиском утечек памяти, то проблема обнаружения утечек иных ресурсов (например, файловых дескрипторов) является к настоящему моменту нерешенной.

Таким образом, задача создания программного средства для проверки освобождения локальных ресурсов в программах на языке С и разработки необходимых для этого алгоритмов является актуальной.

Жизненный цикл идентификатора ресурса. Рассмотрим цикл использования ресурса, такого как открытый файл или объект синхронизации, с позиции прикладного процесса. Допустим, в данный момент времени выполняется процесс, которому необходимо использовать некоторый ресурс. Для этого процесс должен завести некоторый контейнер для хранения идентификатора ресурса, причем в дальнейшем значение одного контейнера может копироваться в другой. Разрабатываемый алгоритм ограничен проверкой контейнеров, являющихся локальными переменными, что типично для ресурсов, время использования которых ограничено некоторой функцией.

После выделения контейнера процесс должен вызвать в терминологии стандартов POSIX системную функцию на выделение ему ресурса операционной системой и записать ее результат в контейнер. В случае успеха процесс получает корректный идентификатор выделенного ресурса, в противном случае системная функция обычно возвращает процессу значение идентификатора, считающееся некорректным (часто это значение «–1»). Использование и освобождение успешно выделенного ресурса осуществляется затем с помощью полученного идентификатора.

Любое действие, совершаемое процессом над ресурсом, использует идентификатор ресурса, поэтому с точки зрения прикладной программы выделенный ресурс эквивалентен его идентификатору, который хранится в некотором контейнере или контейнерах. Жизненный цикл идентификатора ресурса на протяжении всего времени выполнения программы можно описать конечным автоматом, представленным на рис. 1.

По приведенному автомату можно определить множество событий, приводящих к ошибкам при использовании ресурса.

Следующие ситуации рассматриваются как ошибочные:

1) попытка использования или освобождения ресурса с применением неинициализированного контейнера;

2) ресурс был корректно выделен, но идентификатор ресурса затем утерян процессом, т. е. произошла утечка ресурса;

3) попытка использования или освобождения ресурса с помощью контейнера, не содержащего корректного значение идентификатора ресурса;

4) попытка использования ресурса после его освобождения;

5) попытка повторного освобождения ресурса.

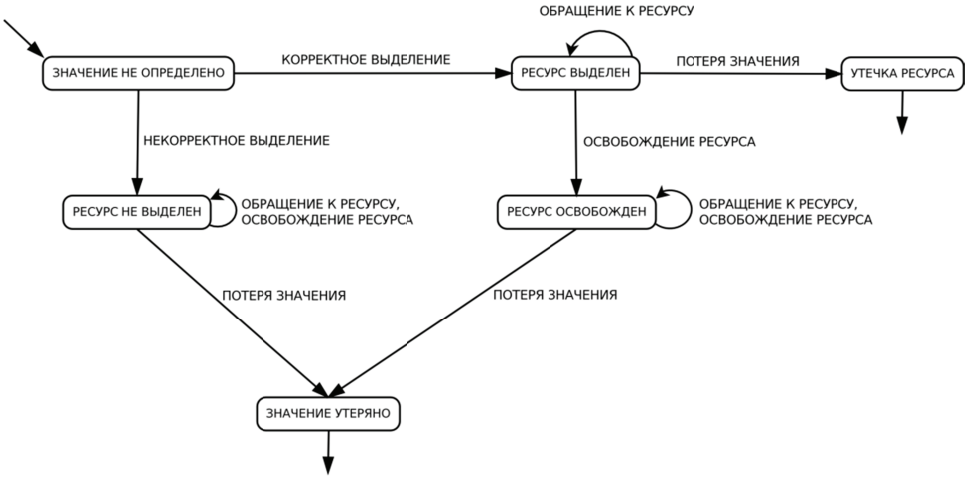

Рис. 1. Жизненный цикл идентификатора ресурса

Для поиска перечисленных ошибок можно использовать как статический, так и динамический подходы к проверке корректности ПО. Для реализации динамического подхода необходимо подменить функции выделения ресурсов, чтобы они в ходе тестирования возвращали и отрицательные результаты — такой поход использует, например, система Dmalloc.

При динамическом подходе необходимо также отслеживать копирование содержимого одного контейнера в другой, для чего следует использовать инструментирование исполняемого кода, аналогично тому, как это делают системы поиска гонок при работе с памятью [3]. В ходе работы динамического метода необходимо будет перебрать все возможные комбинации успешного/неуспешного выделения всех ресурсов в пределах функции для полного покрытия ее кода.

С другой стороны, выделение и освобождение рассматриваемых ресурсов происходит в пределах одной функции, что упрощает применение для решения поставленной задачи и статического анализа. Преимуществом статического подхода является также переносимость созданного программного решения между различными операционными системами. Учитывая эти соображения был выбран статический подход на основе анализа ГПУ функции, выделяющей и освобождающей проверяемые ресурсы [4].

Представление графа потока управления. Граф потока управления является ориентированным графом, вершины которого содержат блоки инструкций.

Сущности ГПУ показаны на рис. 2, каждый ГПУ имеет единственный входной и единственный выходной блок инструкций.

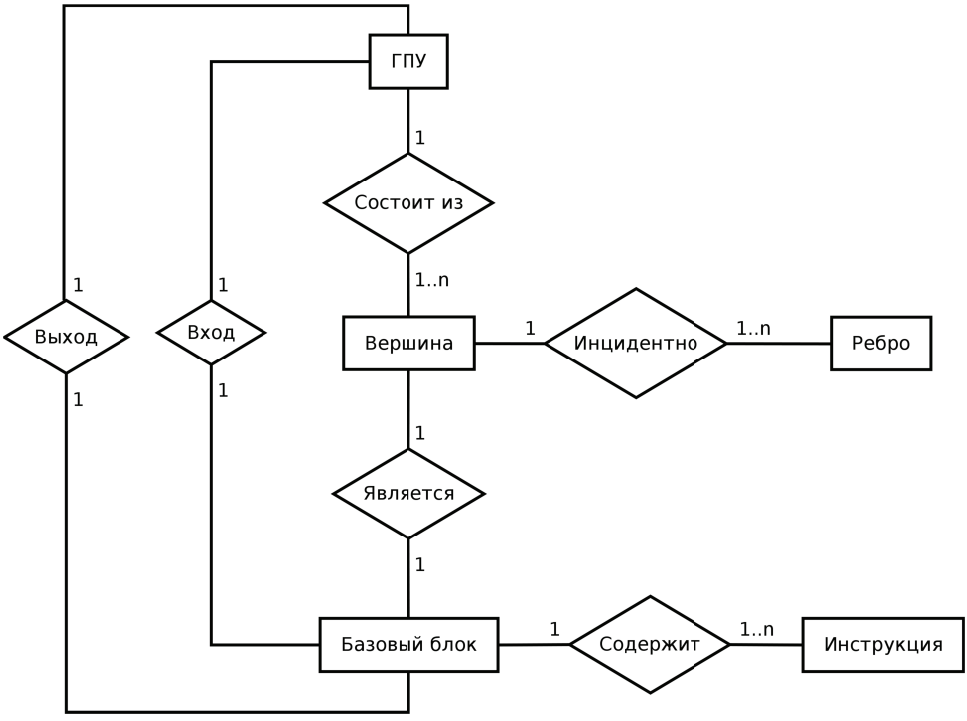

Рис. 2. Сущности графа потока управления

На множестве блоков можно определить отношение доминирования управления [4]:

1) блок A доминирует над блоком B — любой путь, идущий к выходному блоку через B, должен предварительно пройти через A. В данном случае A называется доминатором узла B. Входной блок доминирует над всеми блоками;

2) блок A называется непосредственным доминатором блока B, если A является последним доминатором в любых путях от входного блока к B;

3) блок B постдоминирует над блоком A — любой путь, идущий к выходному блоку через B, должен впоследствии пройти через A. Выходной блок постдоминирует над всеми блоками;

4) блок B называется непосредственным постдоминатором блока A, если B является первым постдоминатором в любых путях от A к выходному блоку.

Компилятор GCC позволяет получить текстовое промежуточное представление кода, включающее информацию о ГПУ: начало и конец каждого блока, доминаторы, постдоминаторы и соответствующие им дуги. Инструкции в таком представлении имеют трехадресную арифметику, а операторы выбора, цикла и ветвления выражаются с помощью условной инструкции перехода. Ниже приведен пример функции, ГПУ которой представлен на рис. 3.

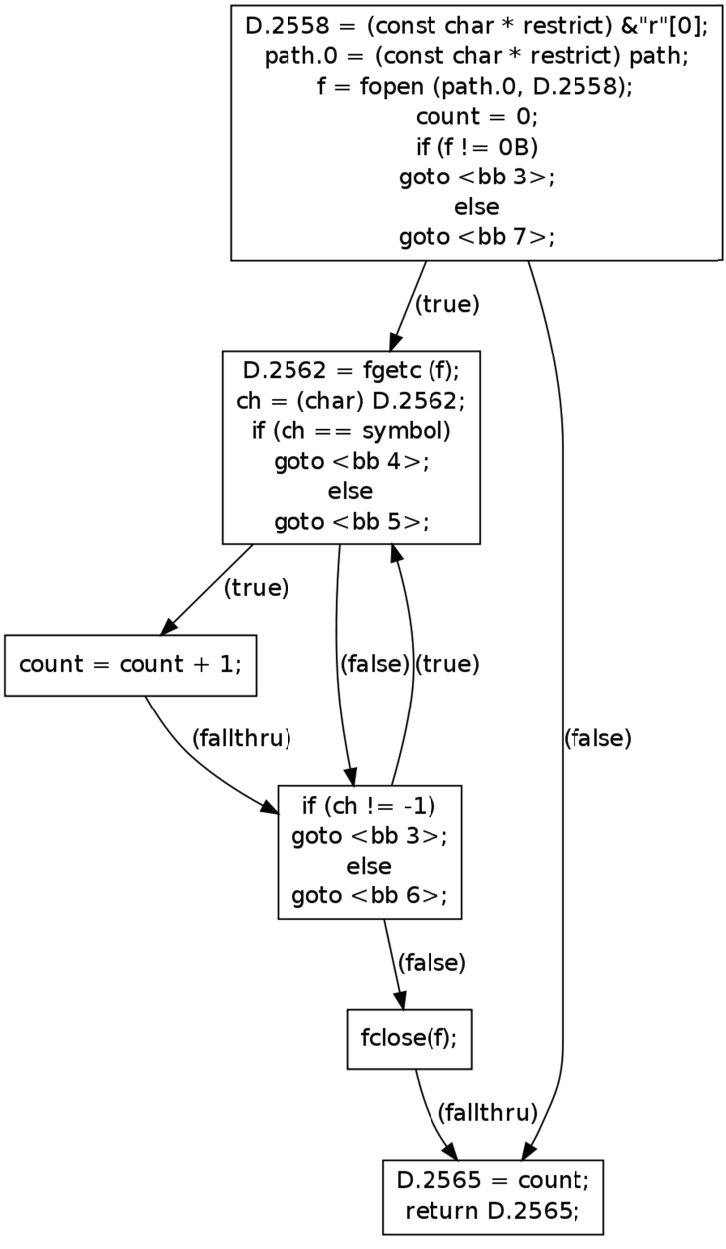

Рис. 3. Визуальное представление ГПУ, полученного с помощью GCC

```
int CountSymbol (char *path, char symbol)
{
   int count = 0;
   FILE *f = fopen(path, "r");
   if (f) {
       char ch;
       do {
          if ((ch = fgetc(f)) == symbol)
               count++;
       } while (ch != EOF);
       fclose(f);
   }
   return count;
}
```

Алгоритм обнаружения утечки локального ресурса. Алгоритм оперирует представлением ГПУ в виде списков смежности, которое получено разбором текстового промежуточного представления кода, включающего информацию о ГПУ. Для этого разбора достаточно использовать регулярные выражения.

Наиболее существенным ограничением разрабатываемого алгоритма является борьба с зацикливанием простейшим способом, когда анализируются только те базовые блоки, анализ которых еще не был проведен. В этом случае итерации цикла, отличные от первой, не будут проанализированы.

Для определения потока управления, соответствующего текущему состоянию ресурса, необходимо выполнить обход графа с учетом значений обрабатываемых локальных переменных. В качестве стартового следует выбрать блок, содержащий инструкцию выделения ресурса. До выполнения данной инструкции состояния локальных переменных не могут зависеть от идентификатора, поэтому не влияют на последовательность выполнения команд использования ресурса.

При выделении ресурса возможен один из двух исходов, поэтому для каждого выделения анализ должен быть выполнен дважды — для корректного и некорректного выделения ресурса соответственно. В пределах одной функции может быть несколько инструкций выделения ресурса, поэтому анализ его использования должен быть произведен для каждого выделения.

Входными данными алгоритма анализа выделения ресурса являются: объект ГПУ, идентификатор базового блока, в котором ресурс был выделен, и индекс инструкции выделения ресурса в списке инструкций базового блока. В результате работы алгоритма должен быть совершен обход всех путей исполнения программы, при которых выполняется выделение ресурса. На рис. 4 представлена схема алгоритма.

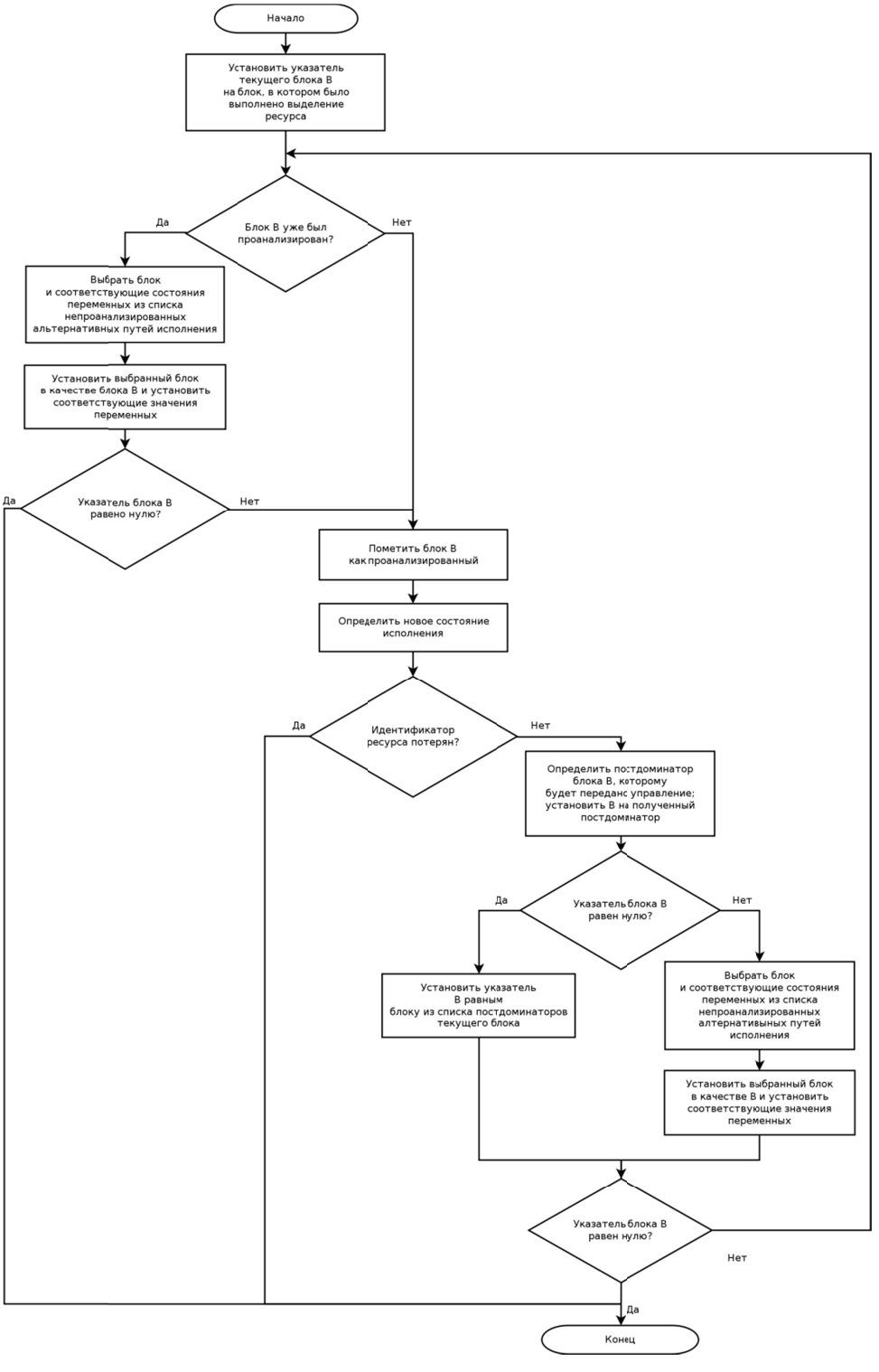

Рис. 4. Алгоритм анализа выделения ресурса

Анализ потока ГПУ начинается с блока, в котором было выполнено выделение ресурса. Во избежание потенциального зацикливания необходимо анализировать каждый блок не более одного раза. Анализ заключается в обработке тех инструкций блока, которые могут потенциально вызвать ошибку использования ресурса. К данным инструкциям относятся все инструкции блока, кроме инструкций передачи управления. В результате обработки инструкции изменяются состояния отслеживаемых локальных переменных, и выполняется запись информации о выявленных ошибках в журнал. Схема выполнения проведения анализа блока приведена на рис. 5.

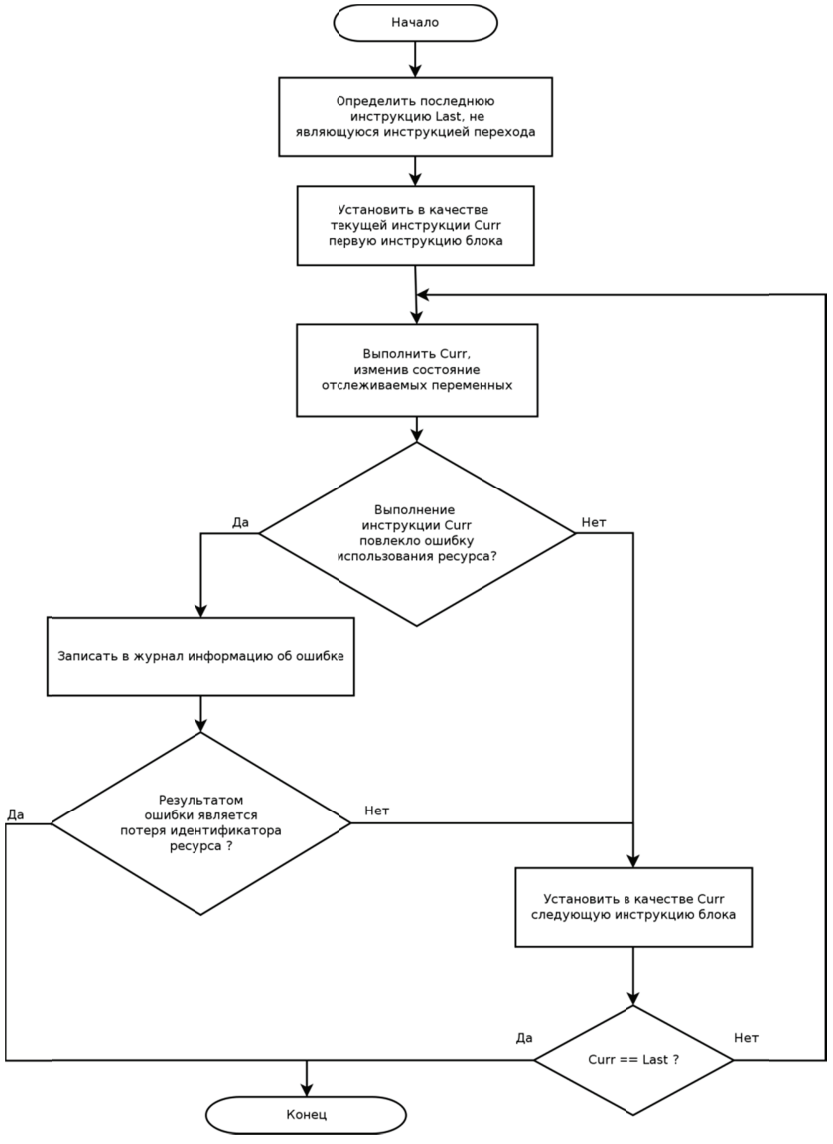

Рис. 5. Алгоритм анализа инструкций блока

После обработки инструкций блока необходимо определить постдоминатор, на который будет передано управление и который будет проанализирован далее. В силу ограниченности множества обрабатываемых инструкций преобразованного представления ГПУ возможна ситуация, при которой постдоминатор не может быть выявлен. В этом случае необходимо выбрать для анализа первый блок из списка постдоминаторов, а остальные сохранить в список непроанализированых альтернативных путей исполнения. Вместе с каждым сохраненным блоком должны быть сохранены текущие значения локальных переменных. Сохраненные блоки будут выбираться для анализа, если список постдоминаторов текущего блока пуст. Анализ использования ресурса имеет смысл до тех пор, пока идентификатор ресурса не утерян, поэтому при потере идентификатора ресурса анализ прекращается.

После анализа инструкций базового блока необходимо определить, какому из его непосредственных постдоминаторов было бы передано управление при выполнении программы. Данный постдоминатор должен быть проанализирован после текущего блока. Для определения следующего блока необходимо проанализировать результат выполнения инструкций передачи управления в зависимости от состояния локальных переменных.

Рассмотрим используемые инструкции передачи управления из блока (см. рис. 3):

1) безусловная передача управления с помощью инструкции *return*. Данная инструкция может находится только в выходном блоке, список постдоминаторов которого пуст, поэтому и блока ГПУ, на который управление было бы передано, нет;

2) безусловная передача управления с помощью инструкции *goto*. В данном случае список непосредственных постдоминаторов содержит единственный элемент, который и является искомым блоком;

3) условная передача управления с помощью инструкций *if* и *switch*. Поскольку в рамках обработки инструкций реализована обработка операторов сравнения языка С, то становится возможным определить тот блок, на который будет передано управление, если известно значение выражений условия перехода или выбора.

Таким образом, для каждого базового блока определен список его непосредственных постдоминаторов.

Программная реализация и результаты применения. Программный комплекс, реализующий описанный алгоритм, основан на языке С++ и состоит из следующих модулей: взаимодействия с GCC, создания внутреннего представления ГПУ, поиска ошибок согласования выделения и освобождения ресурса, пользовательского интерфейса. Схема взаимодействия модулей представлена на рис. 6.

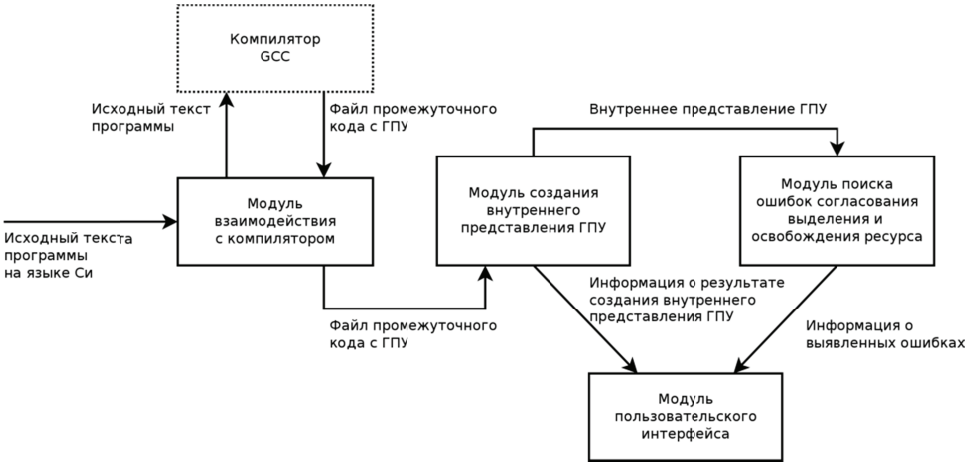

Рис. 6. Схема взаимодействия модулей разработанного ПО

При проведении опытов на вход анализатору будет подано несколько файлов исходных текстов, содержащих ошибки использования ресурса. Множество этих файлов должно представлять все типы ошибок, выявляемых разработанным алгоритмом. Ниже приведены минимальные примеры для всех видов ошибок.

Пример 1. Утечка указателя на выделенную память при выделении. Случай обрабатывается корректно.

```
int main()
{
        int *ptr = malloc(sizeof(int));
}
```

Пример 2. Утечка ресурса (сокета) в процессе работы функции, попытки использования и освобождения ресурса по контейнеру с неверным содержимым. Все три ошибки обнаруживаются верно.

```
int send_msg()
{
     int fd = socket(AF_INET, SOCK_STREAM, 0);
     if (fd == -1) {
        fd = 1;
        close(fd);
     } else {
        char msg[] = "Hello on the other side!";
        fd = 2;
        send(fd, msg, strlen(msg), 0);
     }
     return fd;
}
```

Пример 3. Использование ресурса (переменной типа **FILE ***) после его освобождения и попытка повторного освобождения. Обе ошибки обнаруживаются корректно.

```
int read_file(char *path)
{
     FILE *fd = fopen(path, "r");
     int a = 1;
     fclose(f);
     char first = fgetc(f);
     if (fd) {
         a = 2;
         fclose(fd);
     }
}
```

Пример 4. Использование и освобождение ресурса (номера файлового дескриптора) в случае ошибочного выделения. Обе ошибки обнаруживаются корректно.

```
int read_file(char *path)
{
    int fd = open(path, O_RDONLY);
    int a = 1;
    if (fd == -1) {
        a = 2;
        char ch;
        read(fd, &ch, 1);
        switch(a) {
            case 3: break;
            case 2: close(fd); break;
            case 1: break;
        }
    }
}
```

Пример 5. Использование ресурса после освобождения на итерации цикла, отличной от первой. Такая ошибка не может быть обнаружена в силу ограниченной обработки циклов.

```
void CannotHandle()
{
    int *ptr = malloc(sizeof(int));
    int c = 2;
    while (c < 4) {
         if (c == 3) {
            free(ptr);
            *ptr = 3;
         }
         c++;
    }
}
```

Реализация алгоритма работает в соответствии с его ограничениями: наиболее существенной проблемой является отсутствие обработки циклов и появление вследствие этого ошибки первого рода. Несмотря на этот недостаток, созданные алгоритмы и разработанное программное обеспечение позволяют обнаруживать часть ошибок, связанных с получением и освобождением процессом ресурсов операционной системы, и могут быть полезны при поиске ошибок в исходном коде ПО.

Работа выполнена при частичной поддержке Российского фонда фундаментальных исследований (грант № 13-07-00918).

ЛИТЕРАТУРА

[1] Stroustrup B. *The design and evolution of C++ — Adisson — Wesley*, 1994, 480 p.

[2] Nethercote N., Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. *Proceedings of the 2007 Programming Language Design and Implementation Conference*, 2007, vol. 26, no. 6, pp. 89–100.

[3] Ковега Д.Н., Крищенко В.А. Использование системы LLVM при динамическом поиске состояний гонок в программах. *Инженерный журнал: наука и инновации*, 2013, вып. 2(14). URL: http://engjournal.ru/catalog/it/hidden/549.html

[4] Lowry E.S., Medlock C.W. Object code optimization. *Common*. *ACM*, 1969, vol. 12, no. 1, pp. 13–22.

Статья поступила в редакцию 10.06.2013

Ссылку на эту статью просим оформлять следующим образом:

Медников А.В., Крищенко В.А. Проверка корректности освобождения ресурсов, локальных для функции на языке С. *Инженерный журнал: аука и инновации*, 2013, вып. 6. URL: http://engjournal.ru/catalog/it/hidden/1098.html

Медников Антон Владимирович родился в 1991 г., окончил бакалавриат МГТУ им. Н.Э. Баумана в 2012 г. Магистрант кафедры «Программное обеспечение ЭВМ и информационные технологии». Научные интересы: анализ и верификация программ. e-mail: vk.mrvk@gmail.com

Крищенко Всеволод Александрович родился в 1975 г., окончил магистратуру МГТУ им Н.Э. Баумана в 1998 г. Канд. техн. наук, доцент кафедры «Программное обеспечение ЭВМ и информационные технологии». Автор более 15 научных трудов, научные интересы - статический и динамический анализ и верификации программного обеспечения и сетевых протоколов. e-mail: kva@bmstu.ru