ИСП Институт Системного Программирования Российской Академии наук

ISSN 2079-8156 (Print)
ISSN 2220-6426 (Online)

Труды Института Системного Программирования РАН

Proceedings of the Institute for System Programming of the RAS

Том 27, выпуск 2

Volume 27, issue 2

Москва 2015

ISSN 2079-8156 (Print), ISSN 2220-6426 (Online)

Труды Института системного программирования РАН

Proceedings of the Institute for System Programming of the RAS

Труды ИСП РАН – это издание с двойной анонимной системой рецензирования, публикующее научные статьи, относящиеся ко всем областям системного программирования, технологий программирования и вычислительной техники. Целью издания является формирование научно-информационной среды в этих областях путем публикации высококачественных статей в открытом доступе.

Издание предназначено для исследователей, студентов и аспирантов, а также практиков. Оно охватывает широкий спектр тем, включая, в частности, следующие:

- операционные системы;
- компиляторные технологии;
- базы данных и информационные системы;
- параллельные и распределенные системы;
- автоматизированная разработка программ;
- верификация, валидация и тестирование;
- статический и динамический анализ;
- защита и обеспечение безопасности ПО;
- компьютерные алгоритмы;
- искусственный интеллект.

Журнал издается по одному тому в год, шесть выпусков в каждом томе.

Поддерживается открытый доступ к содержанию издания, обеспечивая доступность результатов исследований для общественности и поддерживая глобальный обмен знаниями.

Труды ИСП РАН реферируются и/или индексируются в:

Proceedings of ISP RAS are a double-blind peer-reviewed journal publishing scientific articles in the areas of system programming, software engineering, and computer science. The journal's goal is to develop a respected network of knowledge in the mentioned above areas by publishing high quality articles on open access.

The journal is intended for researchers, students, and practitioners. It covers a wide variety of topics including (but not limited to):

- Operating Systems.
- Compiler Technology.
- Databases and Information Systems.
- Parallel and Distributed Systems.
- Software Engineering.
- Software Modeling and Design Tools.
- Verification, Validation, and Testing.
- Static and Dynamic Analysis.
- Software Safety and Security.
- Computer Algorithms.
- Artificial Intelligence.

The journal is published one volume per year, six issues in each volume.

Open access to the journal content allows to provide public access to the research results and to support global exchange of knowledge.

Proceedings of ISP RAS is abstracted and/or indexed in:

УДК004.45

Адрес: 109004, г. Москва, ул. А. Солженицына, дом 25.
Телефон: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Сайт: http://www.ispras.ru/proceedings/

Address: 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Tel: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Web: http://www.ispras.ru/en/proceedings/

Содержание

Table of Contents

Использование многопоточных процессов в среде ParJava

М.С. Акопян <manuk@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. В работе описывается подход, применяемый в ParJava по разработке многопроцессно-многопоточных (МПМП) программ. Разработан API и поддерживающая его библиотека, которая позволяет писать параллельные МПМП приложения на языке Java оставаясь в рамках стандарта MPI. Использование потоков в программе позволяет лучше утилизировать ресурсы многоядерного процессора. В рамках работы реализована МПМП программа быстрого преобразования Фурье на языке Java. Проведенные эксперименты показали, что МПМП программа работает быстрее, чем многопроцессная программа.

Ключевые слова: параллельные вычисления; многоядерностые процессоры; параллельные по данным программы MPI, многопоточные Java программы, многопроцессно-многопоточные программы.

DOI: 10.15514/ISPRAS-2015-27(2)-1

Для цитирования: Акопян М.С. Использование многопоточных процессов в среде ParJava. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 5-23. DOI: 10.15514/ISPRAS-2015-27(2)-1.

1. Введение

В настоящее время процессоры, предлагаемые на рынке, базируются в основном на многоядерной архитектуре. Это повышает продуктивность процессоров - уменьшая стоимость процессора (по сравнению с многопроцессорной системой с аналогичным количеством выполняемых модулей) и повышая его производительность. Так же это дает прикладным программистам возможность использовать преимущество общей памяти.

В многопроцессной программе каждый процесс имеет свою область памяти, и если одному процессу при вычислении необходимы данные соседнего процесса, то необходимо провести обмен данными между процессами. Однако если в рамках каждого процесса использовать несколько потоков, которые имеют доступ к общим данным процесса, то это позволит уменьшить накладные расходы по обращению к данным программы. Преимущество общей памяти сопровождаются ее недостатками - необходимы меры по

синхронизации и своевременному доступу к общим данным, во избежание состояний гонок.

При использовании многоядерных процессоров на узле кластера для разработки параллельных программ можно использовать следующие подходы:

1. Многопроцессная программа. nx1 – на узле запускается n процессов по одному потоку в каждом.
2. МПМП программа. 1xn – на узле запускается один процесс, в котором используются n потоков.

Когда в процессе используются много потоков, то внутри каждого процесса обычно производится приватизация с целью уменьшения критических секций. Однако в случае отсутствия критических секций приватизация не нужна (как в случае с МПМП версией БПФ).

В данной статье рассматривается подход, применяемый в ParJava [1] при разработке многопроцессно-многопоточных (МПМП) программ. Среда программирования ParJava позволяет разрабатывать параллельные приложения на современном языке Java, оставаясь в рамках промышленного стандарта MPI. Был разработан API [2] и поддерживающая его библиотека, которая позволяет разрабатывать многопоточные MPI программы на основе стандартного библиотеки java.util.concurrent [3].

В статье приводится описание МПМП программы быстрого преобразования Фурье на языке Java. Проведены серии экспериментов на вычислительном кластере. Представлены графики сравнения многопроцессной и МПМП версий программы.

2. Модель выполнения параллельной многопоточной программы.

Среда программирования ParJava позволяет разрабатывать параллельные приложения на современном языке Java, оставаясь в рамках промышленного стандарта MPI. Коммуникационная библиотека mpiJava.mpi [4] реализующая MPI основана на библиотеке mpiJava[5], который представляет собой привязку языка Java через интерфейс JNI к существующей реализации MPI (MPICH,LAM …). В библиотеке реализованы оберточные функции для стандарта MPI1.1 [6].

Как показывает опыт, использование потоков в рамках одного узла на современных кластерах с многоядерными процессорами может увеличить производительность параллельной программы за счет использования общей памяти и уменьшения накладных расходов.

Многопоточное программирование посредством низкоуровневого интерфейса java.lang.Thread [7] является устаревшим и не считается лучшим решением в достижении параллелизма из-за проблем с производительностью. Начиная с версии 1.5 в Java введена библиотека java.util.concurrent предоставляющая высокоуровневый интерфейс к потокам в Java. Разработанная библиотека

mpi.threads использует и расширяет эффективные инструменты предоставляемые библиотекой java.util.concurrent.

Однако библиотека java.util.concurrent была разработана не для высокопроизводительных вычислений, поэтому в реализации mpi.threads некоторые методы оригинальной библиотеки были закрыты, некоторые адаптированы для применения в рамках модели SPMD. Библиотека mpi.threads не является реализацией OpenMP [8], но многие подходы оттуда используются при работе с библиотекой: модель выполнения потоков, распараллеливаемые инструкции оформляются в виде пользовательских заданий, локальные в рамках задания переменные, разделяемые переменные, операция редукции при формировании распараллеливаемого фрагмента, критические секции, атомарные конструкции, барьеры и.т.д.

Модель выполнения потоков в библиотеке mpi.threads аналогична модели выполнения потоков в OpenMP. На рис. 1 приведена модель выполнения МПМП Java–программы, где N – количество процессов программы (на каждом узле запускается один процесс), n – количество ядер на каждом узле. Рассмотрим поведение параллельной многопоточной программы в рамках одного MPI процесса в среде ParJava.

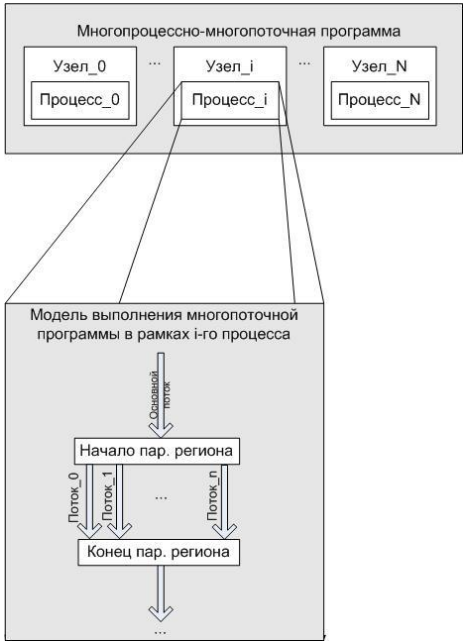

Рис. 1. Модель выполнения МПМП Java–программы.

В отличие от MPI и PGAS [9] моделей, где изначально работают N

В отличие от MPI и PGAS [9] моделей, где изначально работают N вычислительных модулей (процессы в случае MPI и потоки в случае PGAS), при применении библиотеки mpi.threads в начале программы выполняется один поток (основной поток), который последовательно выполняет инструкции программы. В определенных точках программы, где необходимо выполнить параллельно на нескольких потоках некий набор инструкций Ins (параллельный регион), пользователь формирует задания, содержащие инструкции Ins, и передает ссылки на задании библиотеке времени выполнения. Полученные пользовательские задания выполняются каждый в отдельном потоке. Для обеспечения функциональности барьера после формирования заданий пользователь должен явно вызвать соответствующую библиотечную функцию. В результате основной поток блокируется и ждет завершения выполнения сформированных ранее заданий выполняющихся в параллельных потоках. Синхронизация между потоками производится посредством критических секций, семафоров, атомарными операциями над переменными и.т.д.

В целях повышения производительности библиотека mpi.threads предоставляет возможность пользователю в рамках MPI процесса создавать пул потоков. При создании пула создаются и инициализируются фиксированное количество рабочих потоков. После чего рабочие потоки блокируются в ожидании пользовательских заданий. Также в пуле создается очередь заданий. Распараллеливаемый фрагмент последовательной программы должен быть преобразован (заменен) в группу заданий. Во время выполнения программы пользователь формирует новые задания и передает ссылки на них в очередь заданий, откуда задание передается первому свободному потоку, в котором и выполняется данное задание. Использование пула потоков позволяет избежать накладных расходов при создании и запуска новых потоков в ходе выполнения программы.

Пользовательское задание в ParJava представляет собой объект пользовательского класса наследника от системного класса mpiJava.threads.PJTask. В пользовательском классе должен быть реализован метод run(), содержащий исходный код задания. Обычно метод run() содержит цикл (гнездо циклов) исходной программы, который нужно распараллелить между потоками. Параметры задания устанавливаются с помощью метода setParams(…). При распараллеливании гнезда циклов пользователь должен распределить итерации цикла по заданиям с помощью параметров задания setParams(loop_start,loop_end,…) (ниже будет приведен пример распараллеливания цикла).

При работе с параллельными потоками используется родная модель памяти Java. Вся память, выделяемая в рамках задания, доступна потоку, который выполняет это задание. При формировании новых заданий память основного потока не наследуется автоматически. При необходимости использования переменных (или значений переменных) основного потока в задании,

пользователь при формировании задания должен передать в метод setParams(…) задания формальную переменную по ссылке (или по значению). При передаче переменных по ссылке, они становятся разделяемыми (shared) переменными и доступ к таким переменным в избегании состязаний (race condition) нужно осуществлять осторожно (необходимо пользоваться семафорами, критическими секциями, если хотя бы один из обращений к данным переменным является операцией записи).

Пример. Приведем пример (см. рис.2) использования потоков посредством

```
//инициализация массива A
…
//инициализация массива B
…
for(i=1; i <N; ++i)
{
  //тело цикла
  A[i] = B[i] + …;
  …
}
//печать массива A
…
```

Рис. 2.Последовательная версия исходного кода

библиотеки mpi.threads при распараллеливании гнезда циклов. Пусть имеется гнездо циклов с независимыми итерациями, которое нужно выполнить параллельно в нескольких потоках. Пусть количество потоков в пуле равно M. Итерации цикла распределяются равномерно по заданиям. В этом случае пространство итераций цикла [1,N] разбивается на M равных частей и формируется группа из M заданий (задание представляет собой диапазон итераций основного цикла, которые будут выполняться в одном потоке). На рис.3 приведен исходный код задания.

```
class LoopTask extends PJTask{
private int loop_start,loop_end;
private double[] refA;
private double[] refB;
```

```
public void setParams(int loopStart, int loopEnd,double[] A,double[] B)
{
   loop_start = loopStart;
   loop_end = loopEnd;
   refA = A;
   refB = B;
}
public void run()
{
  for(i= loop_start; i < loop_end; ++i)
  {
    //тело цикла
    A[i] = B[i] + …;
      …
  }
}
}
```

Рис.3 Исходный код задания

Задания распределяются между свободными потоками из пула, после чего основной поток блокируется и ожидает окончания выполнения M заданий. На рис.4 приведен фрагмент исходного кода основного потока многопоточной программы.

```
//создание пула потоков (tPool) с M потоками.
…
LoopTask[] loopTask;
//инициализация массива A
…
//инициализация массива B
…
int remaining = N%M;
int chunk = N/M;
int start = 1;
int end = start+(chunk-1) + (remaining >0?1:0);
for(int j = 0; j < M; ++j)
{
  loopTask[j].setParams(start,end,A,B);
  tPool.setTask(loopTask[j]);
  start = end + 1;
  end = start + (chunk-1) + (--remaining >0?1:0);
```

```
}
//Основной поток блокируется в ожидании выполнения заданий
tPool.waitForAll();
//печать массива A
…
```

Рис 4. Исходный код распараллеленной программы.

3. Результаты численных расчетов

Рассмотрим параллельную программу быстрого преобразования Фурье FT из набора NAS Parallel Benchmarks [10,11]. Первоначальная версия NPB на языке Java был разработан и реализован в университете Coruna [12].

В рамках работы проведенной в данной статье, был реализован многопоточный вариант программы FT: коммуникации между процессами программы обеспечиваются функциями MPI, а для взаимодействия между потоками в каждом процессе используются функции из библиотеки времени выполнения mpiJava.threads. Проведена серия экспериментов по сравнению параллельной программы основанной только на библиотеке MPI и параллельной программы на основе MPI и потоков Java.

Приведем описание параллельной программы FT. В данной статье рассматривается 3D версия программы FT (расчетная область представляет собой трех мерную матрицу комплексных чисел). Пусть имеется последовательность $u=\{u_0, u_1,\dots u_{N-1}\}$. Дискретное преобразование Фурье (задача 1D FFT) преобразует последовательность u в другую последовательность U, где

$$U_k = \sum_{n=0}^{N-1} u_n e^{\frac{-2\pi ikn}{N}}$$

В реализованном алгоритме FT дискретное преобразование Фурье применяется на 3D сетке размерностью LxMxN. В этом случае формула преобразование Фурье принимает следующий вид:

$$F_{q,r,s}(u) = \sum_{l=0}^{L-1}\sum_{k=0}^{M-1}\sum_{j=0}^{N-1} u_{j,k,l} e^{\frac{-2\pi ijq}{L}} e^{\frac{-2\pi ikr}{M}} e^{\frac{-2\pi ils}{N}}$$

Данная задача вычисляется на высокопроизводительной вычислительной платформе с распределенной памятью. Параллельную программу FT можно запустить как с использованием 1D декомпозиции, так и 2D декомпозиции данных.

В задаче 3D FFT при применении 1D декомпозиции пространство данных разбивается на слои по направлению оси x– каждому процессу отдается один

слой (см. рис. 5). В рамках каждого процесса имеются все локальные данные для применения 1D FFT по направлениям y и z.

В каждом процессе параллельные вычисления производится в 4 этапа

1. Вдоль направления y применяется 1D FFT.
2. Вдоль направления z применяется 1D FFT.
3. Производится глобальное транспонирование по направлению x, что представляет собой коллективную коммуникацию AllToAll между всеми процессами. В результате у всех процессов появляются данные для счета вдоль оси x.
4. Вдоль направления x применяется 1D FFT.

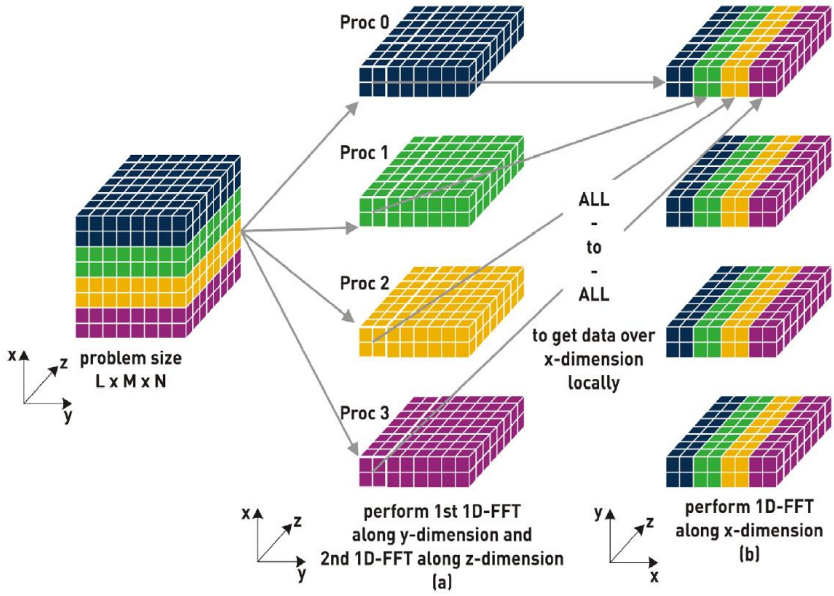

Рис. 5. 1D декомпозиция в решении задачи 3D FFT

При использовании 1D декомпозиции используется только одно глобальное транспонирование для получения в локальную память процесса данных необходимых для счета. Недостатком 1D декомпозиции является ограничение масштабируемости (максимального параллелизма) размером наибольшей длины 3D сетки данных. Однако степень параллелизма можно увеличить количеством ядер на каждом узле кластера при использовании МПМП версии алгоритма.

В задаче 3D FFT при применении 2D декомпозиции пространство данных разбивается как по оси x так и по оси z (см. рис. 6). В рамках каждого процесса имеются все локальные данные для применения 1D FFT только по

направлению оси y. Для применения 1D FFT по осям z и x необходимо провести транспонирование матрицы.

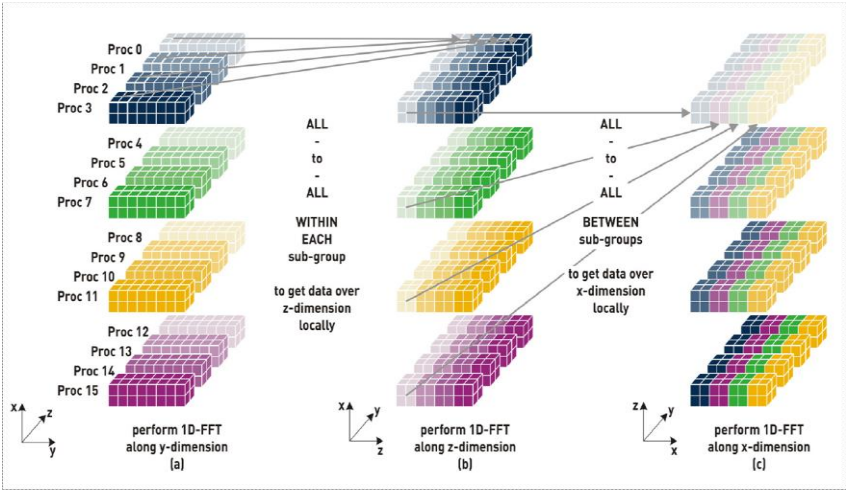

Рис. 6. 2D декомпозиция в решении задачи 3D FFT

В случае 2D декомпозиции вычисления производятся в 5 этапов

1. Вдоль направления y применяется 1D FFT
2. Глобальное транспонирование по оси z
3. Вдоль направления z применяется 1D FFT
4. Глобальное транспонирование по оси x
5. Вдоль направления x применяется 1D FFT

Пусть исходные данные представляют собой куб N^3. В этом случае при применении 1D декомпозиция параллельная программа масштабируется O(N), а при 2D декомпозиция $O(N^2)$.

Программа быстрого преобразования FT тестировалась на кластере МВС-100К (результаты см. рисунок XXX1), на 1410-ти 4-ядерных процессора Intel(R) Xeon(R) CPU X5365 с частотой 3.00GHz (два процессора на каждом узле), с интерфейсной платой HP Mezzanine Infiniband 4x DDR и 8Gb памяти на каждом узле.

FT_T представляет собой параллельную программу быстрого преобразования Фурье с использованием трех мерной расчетной матрицы, где в качестве коммуникаций используется интерфейс MPI, а также в рамках одного процесса используются несколько потоков Java. Рассмотрим некоторые особенности параллельного приложения FT_T. В программе FT_T применяется 1D декомпозиция. На каждом узле кластера запускается по одному процессу, а в рамках каждого процесса в определенных точках программы используются TSize потоков. На каждом узле кластера МВС-100К

имеется 8 ядер и количество используемых потоков должно быть меньше или равно 8-ми во избежание просадки производительности параллельной программы. При проведении исследований использовались два варианта максимального количества потоков TSize={4,8}.

Как уже было отмечено ранее, в целях повышения производительности для управления потоками используется пул потоков. В каждом процессе параллельной программы при инициализации данных создается пул потоков с максимальным количеством потоков TSize. После чего рабочие потоки блокируются в ожидании пользовательских заданий.

Профилирование программы показало, что большая часть времени работы программа проводит в функциях вычисления одномерного 1D FFT, а также в функциях по транспонированию трехмерной матрицы по разным осям. Было выделено шесть функций, которые необходимо распараллелить в рамках процесса. В результате были созданы шесть классов пользовательских заданий:

- два класса для 1D FFT по осям z и y (по оси x используется тот же класс, что и для оси z),
- три класса для локального транспонирования в рамках одного процесса (узла),
- один класс расчетного характера.

При достижении этих функций процесс формирует TSize заданий, инициализирует необходимые начальные параметры и помещает ссылки на сформированные задания в очередь заданий, откуда задания передаются первому свободному потоку, в котором и выполняется данное задание.

В результате распараллеливается расчетная часть – локальное транспонирование в рамках одного процесса (узла) по осям z и y. Транспонирование по оси x производится с применением коллективной коммуникационной функции AlltoAll, которое в данной реализации вызывается в каждом процессе (не в потоке). Поэтому в описываемый работе распараллеливание транспонирования по оси x не проводилось.

В оригинальной версии FT на языке Java использовались локальные массивы, которые аллоцировались каждый раз при вызове функций. Профилирование показало, что некоторые функции, в которых использовались такие локальные массивы, вызываются от десяти до сотен тысяч раз. В результате память набивалась устарелыми данными очень быстро, и проводились многократные сборки мусора, что увеличивало время работы программы. После проведения оптимизаций, большая часть таких массивов были заменены на глобальные. Это позволило снизить количество сборок мусора до одного, что привело к улучшению производительности параллельного приложения.

В приведенных ниже графиках исследуется масштабируемость по Амдалью, то есть фиксируется объем рассчитываемой матрицы и исследуется время выполнения параллельной программы на разных количествах используемых

ядер. Размер рассчитываемой матрицы 512x512x256, объем памяти требуемой в задаче примерно 5Gb, количество внешних итераций 20. В тестах использовалась 64-разрядная версия Java.

В приведенных ниже рисунках FT представляет собой параллельное приложение БПФ на языке Java с применением интерфейса MPI. В данном случае рассматривается параллельная многопроцессная программа без применения многопоточности. FT запускается с применением 2D декомпозиции: для запуска с N(8,16,32,64) ядрами применяется декартовое разбиение пространства процессов [N1,N2], где N=N1*N2 и для запуска используется N1 узлов, на каждом узле N2 процессов. FT_T представляет собой МПМП программу и является собой модификацией FT. В данном случае применяется 1D декомпозиция: для запуска с N(8,16,32,64) ядрами используется линейное пространство процессов [N1], где N1 количество узлов и на каждом узле в определенных точках программы запускаются TSize=N2 потоков (N=N1*N2).

На рис. 7 представлены графики зависимости времени выполнения параллельной программы БПФ от числа используемых ядер.

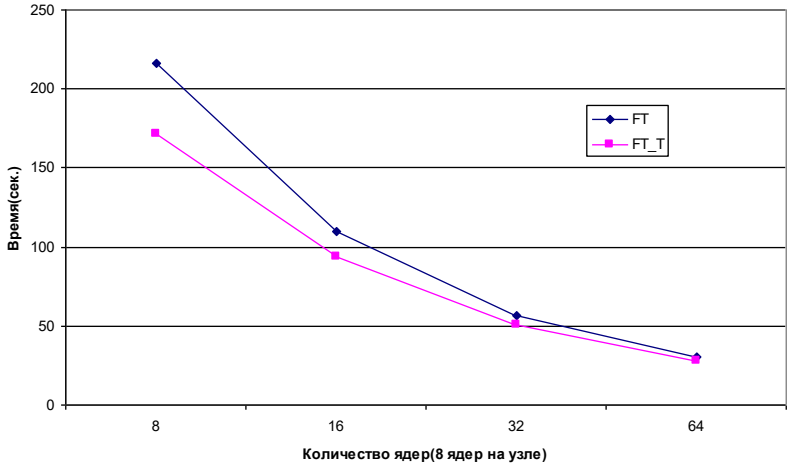

Рис. 7. Сравнение времени выполнения многопроцессной программы БПФ с МПМП версией.

На рис. 7 и рис. 8 на каждом узле используются по восемь ядер: для программы FT N2=8, для программы FT_T TSize=8. МПМП программа FT_T выполняется быстрее, чем многопроцессная программа FT на 9,5%-20%. В FT_T были распараллелены на потоки функции связанные с локальным транспонированием и расчетами.

Инициализация же данных в каждом процессе выполняется в последовательно (используется один поток). То есть степень параллелизма в данном случае меньше чем для программы FT. Если распараллелить на потоки и инициализационные функции, то время выполнения программы FT_T сократиться еще больше.

Нужно заметить, что с увеличением количества используемых ядер преимущество FT_T над FT уменьшается. Профилирование программы показало, что связано это с глобальным транспонированием по оси x, которое производится посредством коллективной коммуникационной функции AlltoAll между процессами программы. На рис. 8 приведены графики зависимости времени выполнения коммуникационных функций в программах FT и FT_T от количества используемых ядер.

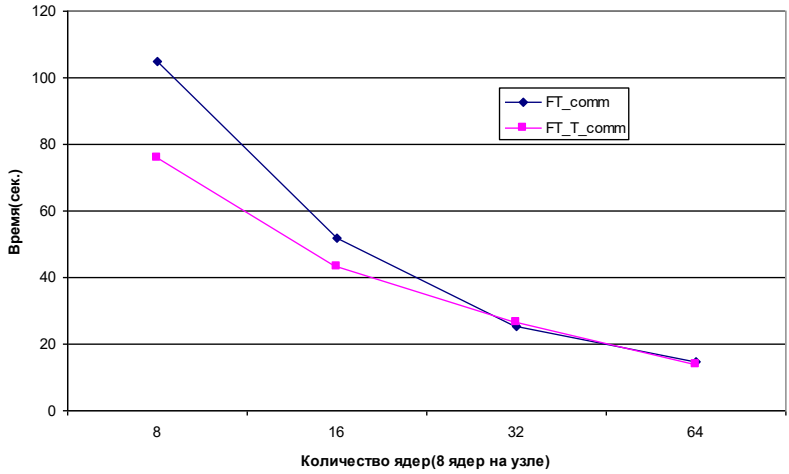

Рис. 8. Сравнение времени выполнения коммуникаций в многопроцессной программе БПФ с МПМП версией.

Как видно из графиков в начальных точках (8 ядер, 16 ядер) время выполнения коммуникаций в программе FT_T заметно меньше, чем для коммуникаций в FT, но с увеличением количества ядер разрыв уменьшается. Это связано с тем, что в FT_T количество обменивающихся процессов TSize раз меньше чем в FT, а размер пересылаемых сообщений гораздо больше. В случае с коллективной коммуникацией AlltoAll время выполнения функции меньше, когда задействованы больше процессов. В этом случае задействованы больше сетевых карт, больше каналов связи и соответственно сокращается простои по ожиданию тех или иных процессов.

На рис. 9 приведены графики времени выполнения функции транспонирования и ее составных частей для программы FT. На данном рисунке FT_1Loc и FT_1Fin относятся к подготовке транспонированию по оси x, FT_1Glo относится к транспонированию по оси x (глобальная коммуникация между узлами кластера), FT_2Loc и FT_2Fin относятся к подготовке транспонированию по оси z, FT_2Glo относится к транспонированию по оси z (глобальная коммуникация между процессами внутри узлам кластера).

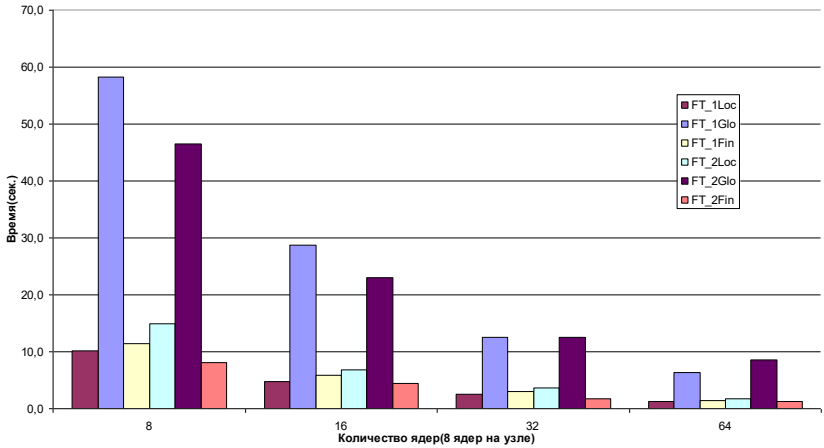

Рис.9. Временной профиль транспонирования в многопроцессной программе БПФ.

На рис. 10 приведены графики времени выполнения функции транспонирования и ее составных частей для программы FT_T. На данном рисунке FT_T_1Loc и FT_T_1Fin относятся к транспонированию по оси z, FT_T_1Glo относится к транспонированию по оси x (глобальная коммуникация между узлами кластера). В программе FT_T отсутствует необходимость коммуникаций при транспонировании по оси z поскольку все потоки в рамках одного процесса имеют доступ к необходимым данным благодаря общей памяти. В этом случае для повышения производительности (кэш попадания) производится лишь копирование в последовательный массив.

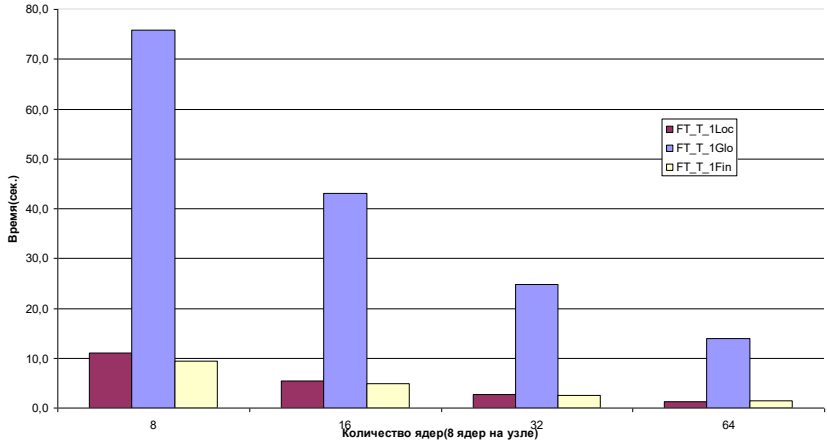

Рис. 10. Временной профиль транспонирования в МПМП программе БПФ.

Из рис. 9, 10 видно, что основная причина деградации производительности программы FT_T это эффект связанный с графиком FT_T_1Glo. В программе FT производятся две коммуникации FT_1Glo и FT_2Glo. При увеличении количества ядер в FT_2Glo не меняется количество обменивающихся процессов, но сокращается размер сообщений. А в FT_1Glo увеличивается количество обменивающихся процессов, одновременно уменьшается размер сообщений. Ситуация с FT_T_1Glo похожа на FT_1Glo, однако в данном случае размер сообщений больше и в функции FT_1Glo параллельно выполняются N2 AlltoAll коммуникаций.

Увеличение количества используемых ядер приводит к существенному сокращению времени выполнения параллельных участков программы, а последовательная часть программы остается почти неизменной, что приводит к увеличению ее доли. Поскольку же степень параллелизма в FT_T меньше чем в FT, FT_T и ускоряется меньше.

Таким образом, в текущей версии программ FT, FT_T при использовании от восьми до тридцати двух ядер МПМП программа FT_T работает в среднем на 10-20% быстрее, чем аналогичная многопроцессная программа FT. Но при дальнейшем увеличении количества ядер преимущество сокращается. В дальнейшем планируется замена схемы коммуникаций в МПМП версии с целью увеличения степени параллелизма, что приведет к ускорению программы.

4. Заключение.

Библиотека mpiJava, разработанная в среде ParJava, позволяет разрабатывать параллельные МПМП приложения на языке Java оставаясь в рамках стандарта MPI. В этом случае параллельная программа запускается в нескольких

процессах, где в начале программы выполняется один поток (основной поток), который последовательно выполняет инструкции программы. В определенных точках программы, где необходимо выполнить параллельно на нескольких потоках некий набор инструкций, пользователь формирует задания, содержащие инструкции, и передает ссылки на задании библиотеке времени выполнения. Полученные пользовательские задания выполняются каждый в отдельном потоке.

В рамках данной работы была реализована МПМП программа быстрого преобразования Фурье на языке Java. Проведенные эксперименты показали, что МПМП программа (FT_T) работает быстрее, чем многопроцессная программа (FT) в среднем на 9,5%-20%. Однако с приростом количества ядер преимущество FT_T над FT уменьшается. Связано это в основном с глобальным транспонированием по оси x, которое производится посредством коллективной коммуникационной функции AlltoAll между процессами программы. В каждом процессе уменьшается доля параллельных вычислений по сравнению с последовательными (инициализация, коммуникации). В дальнейшем планируется замена схемы коммуникаций в МПМП версии с целью увеличения степени параллелизма, что приведет к ускорению программы.

Список литературы

[1]. Иванников В. П., Аветисян А. И., Гайсарян С. С., Акопян М. С. Особенности реализации интерпретатора параллельных программ в среде ParJava. // «Программирование» 2009, №1, с. 10-25

[2]. М.С. Акопян. Расширение модели ParJava для случая кластеров с многоядерными узлами. Труды Института системного программирования РАН, том 23, 2012, с. 13-32

[3]. http://docs.oracle.com/javase/tutorial/essential/concurrency/

[4]. http://www.ispras.ru/ru/parjava/mpijava.php

[5]. Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko and Sang Lim. Object Serialization for Marshalling Data in a Java Interface to MPI. Revised version, August 1999.

[6]. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. MPI – The complete Reference, Volume 1, The MPI Core, Second edition. / The MIT Press. 1998

[7]. https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

[8]. Barbara Chapman, Gabriele Jost, Ruud van van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). ISBN-13: 978-0262533027, MIT, October 2007

[9]. Husbands P, Iancu C, Yelick KA (2003) A performance analysis of the Berkeley UPC compiler. In: International conference on supercomputing, pp 63–73

[10]. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga. THE NAS PARALLEL BENCHMARKS. THE NAS PARALLEL BENCHMARKS. RNR Technical Report RNR-94-007, March 1994

[11]. H. Jagode, “Fourier Transforms for the BlueGene/L Communications Network”, Master’s thesis, University of Edinburgh, 2006.

[12]. Dami´an A. Mall´on, Guillermo L. Taboada, Juan Touri˜no, and Ram´on Doallo. NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java. // Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09). Weimar, Germany, Feb 2009, pp. 181-190.

Using Multithreaded Processes in ParJava Environment

M.S. Akopyan <manuk@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004.

Abstract. Modern processors are based on multicore architectures. Such an approach improves the productivity of processors by decreasing the cost of each processor and increasing its performance. When using multicore processors in nodes of HPC cluster we could use following approaches utilizing node resources: (1) multiprocess program (nx1 - running n processes on a node using one thread in each process); or (2) multiprocess-multithreaded (MPMT) (1xn - running one process on a node and inside of a process n threads may work sharing program data of the process). When using multiple threads in a process inside each process privatization is usually performed to reduce critical sections. In this article we consider the second approach, which will bring better results for parallel application presented in this article because of lack of critical sections. The API and appropriate library has been developed and implemented for MPMT applications. The library allows developing parallel applications using MPI interface and inside of each process it is possible to run a few threads. The parallel MPMT application of FT (Fast Furier Transformation) on Java has been developed. The comparison of multiprocess version of FT to MPMT version of FT has been made. Tests on implemented application show 9,5-20% performance improvement. The profiling of developed application shows the bottleneck of MPMT FT is mostly in communication scheme between nodes. Improving the communication scheme will bring better results.

Keywords: parallel computing; parallel SPMD programs using MPI; multi-core; multithreaded Java programs; multiple process-multithreaded programs.

DOI: 10.15514/ISPRAS-2015-27(2)-1

For citation: Akopyan M.S. Using Multithreaded Processes in ParJava Environment. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 5-22 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-1

References

[1]. Ivannikov V.P., Avetisyan A.I., Gaissaryan S.S., Akopyan M.S. Implementation of Parallel Interpreter in the Development Environment ParJava. Programming and Computer Software. 2009. Volume 35, Issue 1. pp. 6-17. doi: 10.1134/S0361768809010034

[2]. Akopyan M.S. Rashirenije modeli ParJava dlja klasterov s mnogojadernymi uzlami [Extension of ParJava model for HPC clusters with multicore nodes]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 23, pp. 13-32 (in Russian).

[3]. http://docs.oracle.com/javase/tutorial/essential/concurrency/

[4]. http://www.ispras.ru/ru/parjava/mpijava.php

[5]. Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko and Sang Lim. Object Serialization for Marshalling Data in a Java Interface to MPI. Revised version, August 1999.

[6]. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. MPI – The complete Reference, Volume 1, The MPI Core, Second edition. / The MIT Press. 1998

[7]. https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

[8]. Barbara Chapman, Gabriele Jost, Ruud van van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). ISBN-13: 978-0262533027, MIT, October 2007

[9]. Husbands P, Iancu C, Yelick KA (2003) A performance analysis of the Berkeley UPC compiler. In: International conference on supercomputing, pp 63–73

[10]. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga. THE NAS PARALLEL BENCHMARKS. THE NAS PARALLEL BENCHMARKS. RNR Technical Report RNR-94-007, March 1994

[11]. H. Jagode, “Fourier Transforms for the BlueGene/L Communications Network”, Master’s thesis, University of Edinburgh, 2006.

[12]. Dami´an A. Mall´on, Guillermo L. Taboada, Juan Touri˜no, and Ram´on Doallo. NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java. // Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09). Weimar, Germany, Feb 2009, pp. 181-190.

Подход к проведению динамического анализа Java-программ методом модификации виртуальной машины Java[1]

М. К. Ермаков <mermakov@ispras.ru>
С. П. Вартанов <svartanov@ispras.ru>
Факультет вычислительной математики и кибернетики,
Московский государственный университет им. М.В. Ломоносова,
119991, Россия, г. Москва, Ленинские горы, д. 1, с.52

Аннотация. На настоящий момент при разработке программного обеспечения активно используются методы автоматического статического и динамического анализа программ. Так, динамический анализ предоставляет возможности обнаруживать дефекты и ошибки проектирования, проявляющиеся во время выполнения программ, требуя минимального участия эксперта. При проведении динамического анализа распространены два подхода к исследованию выполнения программы — внешний мониторинг, осуществляющийся системными средствами и отладчиками, и инструментация программного кода — модификация и внедрение дополнительной функциональности. При анализе исполняемого кода во внутреннем представлении, обрабатываемого интерпретатором или виртуальной машиной, становиться возможно применение третьего подхода — мониторинга средствами самого интерпретатора или виртуальной машины. В данной статье рассмотрены возможности подобного подхода на примере проведения анализа использования динамической памяти в виртуальной машине Dalvik операционной системы Android.

Работа над представленным методом обусловлена ограничениями виртуальной машины Dalvik, приводящими к невозможности использования большого количества существующих инструментов профилирования памяти. В данной статье рассмотрены непосредственные модификации виртуальной машины Dalvik, включающие расширение поддержки стандартного протокола Java Debug Wire Protocol, для отслеживания событий выделения, освобождения и доступа к памяти. Дополнительно приведён обзор возможностей разработанного отладчика, обеспечивающего регистрацию данных событий в режиме реального времени для последующей обработки. Представленный отладчик, основанный на существующем средстве Dalvik Debug Monitor, позволяет проводить работу одновременно с несколькими активными процессами и обработку результатов множественных запусков. В рамках практических экспериментов были рассмотрен набор стандартных пользовательских приложений

[1] Исследование проводится в рамках научно исследовательских работ Института системного программирования РАН в 2014—2017 годах.

Android, выполняющихся на модифицированной версии виртуальной машины Dalvik под контролем разработанного отладчика. Эксперименты позволили выявить некоторые базовые особенности работы с памятью, включающие активное использование массивов примитивных типов языка Java и неэффективность использования памяти, выделяемой для объектов классов, отвечающих за отображение элементов графического интерфейса приложений.

Ключевые слова: динамический анализ программ; профилирование памяти; Java; Android

DOI: 10.15514/ISPRAS-2015-27(2)-2

Для цитирования: Ермаков М. К., Вартанов С. П. Подход к проведению динамического анализа Java-программ методом модификации виртуальной машины Java. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 23-38. DOI: 10.15514/ISPRAS-2015-27(2)-2.

1. Введение

1.1 Подходы к проведению анализа программ

В настоящее время в процессе разработки и сопровождения программного обеспечения значительную роль играют инструментальные средства для отладки и проведения анализа ключевых аспектов времени выполнения целевого приложения (корректное использование ресурсов, предоставляемых устройствами, и эффективность реализации алгоритмов обработки данных и выполнения заявленной функциональности). Среди данных вспомогательных средств, рассчитанных на нативные приложения (исполняемые файлы которых содержат непосредственно машинные инструкции процессора устройства и управляющую информацию, считываемую загрузчиком операционной системы) можно выделить две большие группы:

- Системные средства отладки и трассировки, предоставляющие стандартные возможности управления выполнением программы и извлечения необходимой информации;
- Средства инструментации кода, производящие модификацию отдельных частей программы для добавления дополнительной функциональности по взаимодействию с некоторым управляющим инструментом во время выполнения или непосредственной генерации данных во время выполнения для последующей обработки.

Средства второй группы являются в высокой степени гибкими и позволяют автоматически производить сложные модификации кода целевого приложения для достижения нужного результата. Непосредственная настройка модификации кода для определённых типов структурных и функциональных единиц (функции, блоки, отдельные инструкции) позволяет эффективно выполнять поставленную задачу. Тем не менее, проведение инструментации неотделимо от проблемы побочных эффектов. Гарантированным побочным

эффектом является непосредственно выполнение действий в рамках анализа программы и увеличение объёма ресурсов, потребляемых программой. При некорректной реализации инструментационного кода побочные эффекты могут вызывать нарушение базовой функциональности программы и появление критических дефектов.

В то же время системные инструменты обычно обеспечивают высокую степень изолированности от выполняемого кода и извлечение данных без влияния на ресурсы, отведённые целевому приложению во время исполнения (за исключением ресурса процессорного времени).

1.2 Анализ интерпретируемых языков

Для приложений, файлы которых содержат интерпретируемый код, наличие дополнительного функционального уровня (непосредственно интерпретатора) вносит ряд особенностей. Описанное выше разбиение на две группы инструментов анализа расширяется до трёх групп инструментов по степени абстракции от внутренней организации исследуемых приложений [1]:

- Системные средства могут быть применены только к интерпретатору, что значительно уменьшает возможность подробного анализа целевого кода — полученные результаты в основном описывают поведение интерпретатора, внешнюю статистику по использованию ресурсов и эффективности реализации алгоритмов самого интерпретатора. Тем не менее, при использовании разметки исполняемых файлов и библиотек интерпретатора возможно учитывать внутреннюю структуру и логику кода анализируемых программ [1,2].
- Средства инструментации, направленные на работу с интерпретируемым кодом, позволяют создавать инструменты анализа, применимые (при наличии интерпретатора) на разных машинных архитектурах и операционных системах.
- Третья группа средств анализа основана на прямом взаимодействии с интерпретатором — использовании подключаемых агентов и отладчиков. Также возможно непосредственное изменение механизмов интерпретации или создание нового интерпретатора с встроенным функционалом по извлечению специфической информации при выполнении кода анализируемой программы.

Как и для нативных приложений, системные средства оказывают минимальное влияние на выполнение анализируемой программы, однако не имеют возможности оперировать высокоуровневыми конструкциями целевой программы. Инструментация кода оказывает наибольшее влияние на выполнение программы, однако позволяет проводить анализ на точечном уровне как по выбору исследуемых частей программы, так и по доступу к параметрам инструкций, блоков и функций. В то же время непосредственная

интеграция анализа с интерпретатором позволяет манипулировать данными приложения, предоставляет доступ ко всему контексту выполнения программы, и вносит меньший уровень влияния на внутренние ресурсы, используемые целевым приложением (так, например, для языка Java инструментационный код должен работать с кучей и стеком виртуальной машины, в то время как модификация самой виртуальной машины Java позволит проводить извлечение информации без всякого влияния на данные области памяти). Недостатком подхода анализа на уровне интерпретатора является затратность реализации и настройки инструмента анализа, а также зависимость от особенностей непосредственно интерпретатора.

1.3 Анализ использования памяти в Java-программах

Рассмотрим подробно программы, написанные на языке Java и выполняющиеся на виртуальной машине Java, с точки зрения вопроса эффективности использования памяти. Использование автоматических и полуавтоматических средств анализа, позволяющих обнаруживать ошибки работы с памятью, снижает вероятность наличия критических дефектов в конечной выпускаемой версиях разрабатываемых программ. Целый набор средств (YourKit Java Profiler [3], JProfiler [4], Eclipse MAT [5], hprof [6] и др.) активно используется при разработке приложений Java. Большинство данных инструментов нацелено на официальную реализацию виртуальный машины HotSpot и на стандарты дополнительных функций виртуальных машин Java. Соответственно, для альтернативных версий виртуальных машин Java данные средства либо не могут быть применены напрямую, либо имеют целый ряд ограничений, в то время как проведение анализа использования памяти для данных виртуальных машин остаётся по-прежнему актуальной задачей.

В рамках данной статьи будет рассматриваться задача анализа использования памяти приложениями операционной системы Android, выполняющимися на виртуальной машине Dalvik. В качестве решения данной задачи предлагается проведение модификации виртуальной машины Dalvik с целью добавления возможности сохранения информации о выделении, освобождении и использовании памяти и передачи данной информации с устройства Android на хост-устройство для последующей обработки. В части 2 статьи будут рассмотрены основные положения предметной области (виртуальная машина Dalvik, JDWP — протокол взаимодействия с отладчиком для виртуальных машин Java [7], подходы к проведению анализа использования памяти). В части 3 будет дано описание проведённых в рамках исследования модификаций виртуальной машины Dalvik, обеспечивающих сохранение и передачу необходимой для оценки использования памяти информации. Часть 4 содержит краткое описание клиента, обеспечивающего приём и организацию данных от виртуальной машины Dalvik на хост-устройстве, и оценку тенденций использования памяти для ряда стандартных приложений системы Android. Наконец, в части 5 будут рассмотрены особенности

представленного подхода в контексте результатов тестовых запусков и возможные направления дальнейших исследований.

2. Анализ использования памяти для приложений Android

2.1 Виртуальная машина Dalvik

Виртуальная машина Dalvik является составной частью операционной системы Android, предназначенной для запуска пользовательских и стандартных системных приложений. Dalvik обладает рядом отличий от стандартных виртуальных машин, использующихся для выполнения Java-приложений:

- Формат байт-кода приложений (dex) для виртуальной машины Dalvik отличается от стандартного формата байт-кода Java. Схема разработки приложений для операционной системы Android предполагает написание кода приложения, генерацию стандартного байт-кода и его трансформацию в формат dex средствами, предоставляемыми в рамках набора инструментов разработки программного обеспечения Android (Android SDK).
- Реализация Dalvik включает поддержку набора стандартных функций виртуальных машин Java (протокол JDWP, генерация слепков кучи, подключение нативных библиотек через интерфейс JNI).
- В виртуальной машине Dalvik отсутствует поддержка подключаемых динамических агентов (протокол JVMTI).
- Код виртуальный машины Dalvik находится в открытом доступе (как и все основные составляющие операционной системы Android).

Данные свойства определяют ряд ограничений на возможность создания инструментов анализа:

- Использование внутреннего формата байт-кода не позволяет напрямую применять имеющиеся средства статической инструментации Java байт-кода (полноценных средств для проведения подобных манипуляций для формата dex на данный момент не существует). Инструментация байт-кода возможна косвенно как промежуточный этап создания приложения Android — написание кода на Java, компиляция в байт-код Java, инструментация байт-кода Java, трансформация байт-кода Java в байт-код dex. В этом случае, однако, возрастает степень накладных расходов на выполнение инструментационного кода (этап трансформации в dex ограничивает оптимизацию инструментационного кода).
- Отсутствие поддержки протокола JVMTI не позволяет использовать имеющиеся средства анализа и динамической инструментации кода, реализованные для стандартных виртуальных машин Java.

В то же время, открытость кода виртуальной машины Dalvik предоставляет возможности внесения любых модификаций для проведения необходимого анализа. Указанная выше поддержка стандартного протокола коммуникации с отладчиками JDWP и работы с динамической кучей означает наличие базы, на основе которой возможна эффективная реализация инструмента анализа.

2.2 Протокол JDWP

Протокол Java Debug Wire Protocol (JDWP) представляет собой стандарт низкоуровневого взаимодействия виртуальной машины Java с некоторым средством, производящих отладку целевого приложения. В данный протокол входит описание формата поддерживаемых пакетов, типов данных, отвечающих функциональным элементам, используемым виртуальной машиной (классы, объекты, потоки управления и т.д.). Виртуальная машина Java, предоставляющая поддержку протокола отладки, может быть запущена с соответствующими опциями для инициации прослушивания канала связи, по которому может произойти попытка соединения со стороны средства отладки. В протоколе зафиксирована последовательность пакетов аутентификации и порядок реакции на предусмотренные запросы и команды. Список запросов и команд включает следующие основные группы:

- Базовые команды управления виртуальной машиной — остановка и возобновление потоков управления, инициация сборки мусора и т.д.
- Запросы информации о сущностях, используемых программой, — классах, объектах и т.д.
- Создание точек наблюдения и триггеров событий, автоматически отправляющих данные отладчику при фиксировании некоторой ситуации.
- Команды создания и модификации параметров объектов и классов.

Значительным преимуществом протокола JDWP является возможность расширения пакетов за счёт использования пользовательских кодировок. При реализации соответствующей поддержки на уровне виртуальной машины можно определять фактически произвольные команды и запросы для извлечения нужной информации или влияния на целевое приложение.

Поддержка протокола JDWP в виртуальной машине Dalvik является неполной. В частности, отсутствует возможность создавать триггеры на доступ к полям объектов и ряд других опций протокола.

Протокол JDWP в Dalvik был расширен группой пакетов для работы специального средства Dalvik Debug Monitor Service (DDMS) [8], предоставляющего возможности получения информации по аллокациям памяти, работе потоков управления и содержимого стандартного журнала, используемого большинством приложений операционной системы Android. Данные по аллокациям памяти, получаемые сервисом отладки Android, позволяют проводить анализ использования памяти целевым приложением.

Генерируемые наборы данных, однако, не позволяют проводить полный анализ выделения, освобождения и использования памяти.

2.3 Анализ памяти по слепкам кучи

Виртуальная машина Dalvik поддерживает возможность извлечения полного слепка кучи по запросу от отладчика или анализатора. Данный слепок содержит информацию о всех активных в данный момент времени объектах, а также информацию о связи между ними. Связи между объектами рассчитываются по возможности осуществить доступ к одному объекту при наличии доступа к другому (например, объекты A и B связаны, если ссылка на B записана в одно из полей объекта A). В куче выделяются корневые объекты [9], создаваемые близко к точке входа в приложение и активные в течение всего времени выполнения программы. Если для объекта существует цепочка связей до одного из корневых объектов, то он считается активным. Если такой цепочки нет, то невозможно осуществить доступ к данному объекту, а значит он не может использоваться в программе и память, выделенная для него, может быть освобождена на ближайшем этапе сборки мусора.

Анализаторы, направленные на обработку слепков кучи (Eclipse MAT, hprof), позволяют проводить сравнительную оценку состояния кучи до и после некоторого события в программе, что, в свою очередь, позволяет эксперту выявлять потенциальные проблемы утечек памяти и поддержки излишнего количества долгоживущих объектов, реально не используемых в программе. Подобный анализ, тем не менее, не позволяет получать информацию обо всех созданных объектах (генерация слепка кучи — достаточно дорогая операция, которая требует остановки выполнения виртуальной машины) и не предоставляет информацию о непосредственном использовании выделенной памяти.

2.4 Анализ памяти в режиме реального времени

Большая группа анализаторов (например, указанное выше средство JProfiler) использует протокол JVMTI для подключения и работы агента, обеспечивающего прослушивание ряда событий, включающих создание объектов и освобождение памяти во время проведения сборки мусора. Данные событий, описывающие типы создаваемых и освобождаемых объектов, точки создания и другие параметры, поступают на клиентское приложение для обновления базы данных в реальном времени. Собираемая статистика может быть пополнена и слепками кучи, создаваемыми в ключевые моменты времени.

Отсутствие поддержки протокола JVMTI в виртуальной машине Dalvik делает невозможным применение данных средств. Тем не менее, описанный подход (получение всей информации в режиме реального времени) представляется наиболее мощным подходом анализа использования памяти.

2.5 Анализ памяти в Dalvik в режиме реального времени

Описанные выше особенности реализации виртуальной машины Dalvik и возможные подходы к анализу использованию памяти обуславливают основные составляющие выбранного подхода:

- Проведение анализа в реальном времени — сбор всех событий выделения, освобождения и использования памяти, и пересылка собранных данных на хост-устройство;
- Реализация сбора событий путём проведения модификации непосредственно кода виртуальной машины Dalvik;
- Упаковка собираемых данных в специализированные пакеты и передача пакетов отладчику на основе протокола JDWP;
- Реализация программного средства, осуществляющего получение и обработку данных, генерируемых модифицированной виртуальной машиной Dalvik, на хост-устройстве.

3 Модификации виртуальной машины Dalvik

3.1 Общая структура модификаций

Модификации виртуальной машины Dalvik нацелены на выполнение дополнительных действий при возникновении следующих событий:

- Создание объекта определённого типа;
- Освобождение памяти, выделенной под объект, при проведении сборки мусора;
- Загрузка класса и инициализация его параметров;
- Использование объекта (вызов метода объекта, доступ к полю объекта, синхронизованный доступ к полю объекта, доступ к элементу массива, базовые методы чтения массивов JNI, нативные реализации методов базовых классов, таких как java.lang.String).

Обработка данных событий заключается в извлечении параметров событий и упаковки данных параметров в специализированные пакеты для последующей передачи на хост-устройство. Накапливаемая в пакетах информация буферизуется; размеры внутренних буферов выбраны таким образом, чтобы обеспечивать частые обновления состояния памяти приложения, избегая перегрузки канала связи между устройством Android и хост-устройством.

Полное описание содержимого пакетов данных представлено на следующей схеме, использующей обозначения:

- REquest ALlocations (REAL) — пакет с данными о созданных объектах,
- REquest FReed (REFR) — пакет с данными об освобождённых объектах,

- OBject USage (OBUS) — пакет с данными об использовании объектов,
- CLaSS information (CLSS) — пакет с данными о загруженных классах.

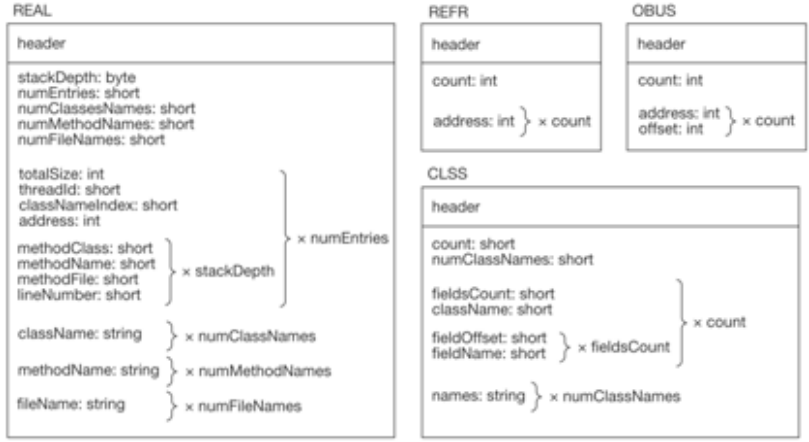

Рис. 1: Форматы пакетов данных, генерируемых модифицированной виртуальной машиной Dalvik

3.2 События создания объекта

Создание объекта и выделение памяти в виртуальной машине Dalvik происходит в рамках единственной процедуры, которая в свою очередь вызывается несколькими модулями (модуль создания стандартных объектов по обрабатываемому байт-коду, модуль загрузки классов и создания объектов-контейнеров для данных классов, управляющий модуль обработки запросов JNI и т.д.), что позволяет производить точечную обработку данных событий. Для работы средства DDMS протокол JDWP уже имеет расширение, описывающее пакеты с информацией о выделенных объектах. Данные пакеты имеют сложную структуру, описывающую основные параметры создаваемых объектов:

- количество объектов пакета;
- классы объектов (индексы множества строк);
- размеры выделенной памяти;
- идентификаторы потоков;
- реальные адреса выделенной памяти (для идентификации объектов);
- счётчики количества элементов во множестве строк;
- стеки вызовов (определяющие точки кода, в которых произошло создание объектов);
- множество строк, разделяемое между всеми остальными полями для сокращения размера пакета.

3.3 События освобождения памяти

Сборка мусора в виртуальной машине Dalvik осуществляется на основе алгоритма MarkSweep [9]. Работа алгоритма состоит из нескольких этапов, наиболее важными из которых является разметка объектов по степени активности (корневые объекты кучи помечаются как активные, все объекты, связанные по ссылке с активными, также считаются активными) и обход меток с целью освобождения памяти, занимаемой неактивными объектами. Во время этапа обхода работа с объектами осуществляется по реальным адресам памяти. Именно данные адреса сохраняются во внутренние списки с целью пересылки на хост-устройство. Пакеты данных формируются из двух групп адресов — использованных объектов и неиспользованных объектов (по биту структуры объекта при проведении частичного анализа использования памяти). При проведении полного анализа использования памяти разделение на две группы не используется.

3.4 События использования памяти (неполный анализ)

Виртуальная машина Dalvik содержит значительное количество модулей, осуществляющих прямой доступ к содержимому объектов приложения. Реализованные в рамках исследования модификации виртуальной машины Dalvik направлены на перехваты событий доступа к объектам в следующих модулях:

- Внутренние модули интерпретации операций байт-кода dex: доступ к полю объекта заданного типа, доступ к элементу массива по заданному индексу, вызов метода объекта.
- Частные реализации методов базовых классов (java.lang.String, java.lang.System и т. д.).
- Реализации основных операций интерфейса JNI: работа с массивами, трансформация внутренних типов JNI в объекты Java, выделяемые на куче.

В рамках проведения неполного анализа использования памяти обработка этих событий приводит к взведению флага использования во внутренней структуре виртуальной машины Dalvik, определяющей объект Java. Данные подход не приводит к необходимости записи информации во внутренний список и передачи на хост-устройство, однако позволяет определять объекты приложения только как использованные/неиспользованные без указания о том, какая часть выделенной памяти была прочитана за время существования объекта.

3.5 События использования памяти (полный анализ)

При проведении полного анализа использования памяти в качестве событий рассматриваются все вышеуказанные группы. Однако вместо взведения флага,

информация о том, какое поле объекта или какой элемент массива был использован, записывается во внутренний список для последующей передачи на хост-устройство. Это значительно повышает накладные расхода на проведение анализа по сравнению с неполным подходом, однако позволяет получать более точные данные об эффективности использования созданных объектов.

Пакеты данных о использовании объектов содержат непосредственно адреса использованных объектов и смещения в памяти, по которым произошёл доступ на чтение.

3.6 События загрузки классов Java

Для корректной работы полного анализа использования памяти в рамках модификаций, внесённых в виртуальную машину Dalvik, была добавлена обработка событий загрузки классов (как системных, так и пользовательских). Данные о полях класса и внутренних смещениях, используемых виртуальной машиной Dalvik, формируются в список и в пакеты, передаваемые на хост-устройство. Впоследствии, они используются для отображения смещений, передаваемых в пакетах использования объектов, на имена полей класса.

4 Практические результаты

4.1 Средство сборки и обработки данных анализа

На основе библиотеки ddmlib [10] был разработан специализированный инструмент, предназначенный для работы на рабочей станции, к которой подключено устройство Android, и обработки данных, получаемых от анализируемого приложения. Работа с инструментом возможна как в автоматическом режиме для трассировки длительной сессии выполнения анализируемого приложения, так и в интерактивном режиме с целью более строгого контроля за выполнением анализируемого приложения.

В число функций разработанного инструмента входят следующие:

- Установление соединения с процессом, в котором выполняется виртуальная машина, и инициализация параметров передачи пакетов от устройства и команд на устройство.
- Активное прослушивание канала связи, приём пакетов данных и обновление общего набора информации, полученного во время сессии работы с анализируемым приложением.
- Командный пользовательский интерфейс для отображения общего набора информации по работе анализируемого приложения с предоставленной виртуальной памятью. Данный интерфейс предоставляет набор настраиваемых фильтров для отображения данных, возможность сохранения и загрузки данных сессии для

анализа в автономном режиме (без активного взаимодействия с устройством Android).

- Работа с несколькими наборами данных, полученными в разные моменты времени в рамках одного запуска приложения или в рамках нескольких запусков приложения.
- Отсылка контрольных запросов и команд виртуальной машине Dalvik с целью выполнения задач отладки (временная остановка и возобновление выполнения, запроса запуска сборки мусора, запрос создания слепка кучи для использования возможностей существующих инструментов профилировщиков Java).

4.2 Анализ стандартных приложений Android

Модифицированная версия виртуальной машины Dalvik была использована для исследования работы с памятью ряда стандартных приложений операционной системы Android:

- Browser – браузер ресурсов сети Интернет;
- Calendar – приложение, предоставляющее возможности органайзера;
- Contacts – менеджер списка контактов и групп;
- Mms — приложение для обмена текстовыми и мультимедийными сообщениями;
- Deskclock – экранные часы и менеджер будильников.

Для указанных приложений был проведён ряд запусков на выполнение некоторой последовательности действий после подключения средства приёма пакетов. Запуски производились на устройстве PandaBoard [11] с версией операционной системы Android 4.2.2, подсоединённой к хост-устройству через подключение USB. Замедление работы приложений по сравнению с работой без проведения обработки событий, рассмотренных ранее, не превышало двухкратного. Потеря производительности была обусловлена в большей степени значительным количеством передаваемых данных при ограниченной (2-3 Мб/с) пропускной скорости соединения (внутренняя организация и передача данных через утилиту adb, предоставляемую в рамках набора инструментов разработки программного обеспечения Android (Android SDK).

Проведённые запуски выявили ряд закономерностей использования памяти в приложениях.

Для всех рассмотренных приложений основная доля выделенной памяти (от 50% для приложения Calendar до 80% для приложения Browser) приходится на объекты типов char[], byte[] и int[].

- Объекты типа byte[] являются долгоживущими и используются для хранения изображений и контейнеров, отображаемых на экране устройства. Классами, осуществляющими создание наибольшей части

объектов данного типа являются java.nio.ByteBuffer, android.graphics.BitmapFactory и android.graphics.Bitmap.

- Объекты типа char[] создаются в значительном количестве при работе со строками классами java.lang.String и java.lang.StringBuilder.
- Объекты типа int[] являются короткоживущими и создаются в основном при работе системы исключений классом java.lang.Throwable. Обработка ситуаций возникновения исключений, проводимая виртуальной машиной Dalvik, включает в себя набор действий, выполнение которых является более затратным (по времени и ресурсам), чем реализация в программном коде развёрнутой системы проверок ограничений. Большое количество исключений, генерируемых при выполнении приложений, позволяет говорить о целесообразности проведения оптимизации методов, приводящих к появлению данных исключений.

Доля использованной памяти для объектов классов с большим количество полей, например, стандартных классов контейнеров и графических элементов операционной системы Android, в значительной степени варьируется.

В качества примера можно рассмотреть использование класса графических объектов android.widget.TextView, активно применяемого для отображения данных в приложении. Следующий график показывает распределение доли использованных байтов на момент освобождения памяти объектов TextView в рамках сессии работы с приложением.

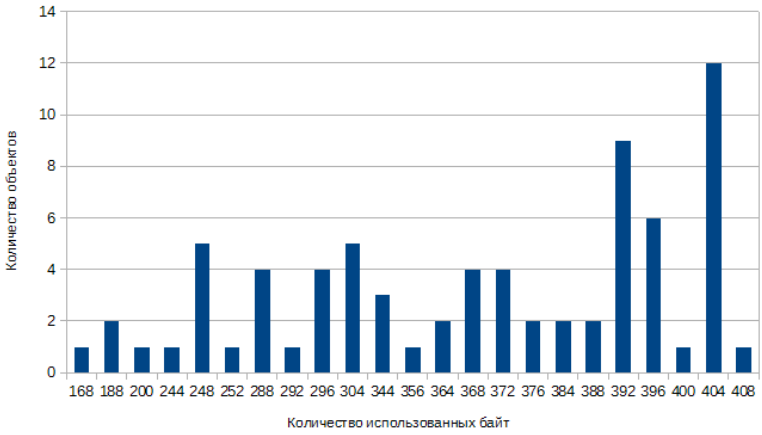

Рис. 2: Распределение доли использованной памяти объектов android.widget.TextView

В то время как полный размер аллокации каждого объекта android.widget.TextView составляет 688 байт, менее, чем 412 байт было

использовано для каждого созданного объекта данного типа за время его существования. В данные расчёты входят как поля непосредственно самого класса android.widget.TextView, так и классов, от которых данных класс наследуется. Подобные результаты свидетельствуют о возможности оптимизации классов, используемых для отображения пользовательского интерфейса.

5 Заключение

Предложенный подход, основанный на получении и обработке данных от модифицированной виртуальной машины Dalvik, предоставляет возможности определения критических и других проблем работы с динамической памятью целевого приложения, выполняющегося в рамках виртуальной машины. Получаемая статистика позволяет осуществлять профилирование как кода пользовательских приложений, так и компонентов системных классов Android. Возможность совмещения и сравнительной оценки наборов данных, соответствующих независимым сессиям выполнения приложения, позволяет проводить анализ стабильности целевого приложения при различных сценариях его использования. Возможность одновременного анализа нескольких процессов, в контексте которых выполняются виртуальные машины, позволяет исследовать взаимодействие приложений.

Тем не менее, существует ряд вопросов и возможностей получения некорректных результатов при активном использовании нативного кода JNI. На данный момент покрыт ряд операций интерфейса JNI по доступу к данным объектов (прежде всего, массивов), однако непосредственная обработка содержимого данных объектов проводится внутри модулей, реализованных в машинных кодах целевой архитектуры, и недоступна виртуальной машине Dalvik. При активном использовании подобных модулей появляется вероятность наличия некорректных данных в итоговой статистике из-за невозможности перехвата событий доступа к содержимому объектов.

В качестве потенциального решения данной проблемы рассматривается возможность проведения совмещённого анализа на основе модификаций виртуальной машины Dalvik и дополнительных методов, осуществляющих контроль за выполнением внешнего для виртуальной машины кода. В качестве подобных методов может быть использована статическая бинарная инструментация библиотек, реализованных в машинных кодах целевой архитектуры, или перехват механизмов интерфейса JNI для запуска инструмента динамического анализа.

Список литературы

[1]. Stephen Kell, Danio Ansaloni, Walter Binder, Lukáš Marek. The JVM is not observable enough (and what to do about it). Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages, 2012. pp. 33-38. doi: 10.1145/2414740.2414747

[2]. Страница документации механизма Dtrace для виртуальной машины HotSpot (http://docs.oracle.com/javase/6/docs/technotes/guides/vm/dtrace.html) [HTML]. Дата обращения: 24.06.2015.
[3]. Страница продукта YourKit Java Profiler (https://www.yourkit.com/features/) [HTML]. Дата обращения: 30.06.2015.
[4]. Страница продукта Jprofiler (https://www.ej technologies.com/products/jprofiler/overview.html) [HTML]. Дата обращения: 30.06.2015.
[5]. Страница проекта Eclipse MAT (https://eclipse.org/mat/) [HTML]. Дата обращения: 30.06.2015.
[6]. Страница документации инструмента hprof (http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html) [HTML]. Дата обращения: 30.06.2015.
[7]. Страница документации протокола JDWP (https://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html) [HTML]. Дата обращения: 23.06.2015.
[8]. Страница описания средства DDMS (http://developer.android.com/tools/debugging/ddms.html) [HTML]. Дата обращения: 30.06.2015.
[9]. Paul R. Wilson. Uniprocessor garbage collection techniques. Proceedings of the International Workshop on Memory Management, 1992. pp. 1-42.
[10]. Репозиторий библиотеки ddmlib (https://android.googlesource.com/platform/tools/base/+/master/ddmlib/) [HTML]. Дата обращения: 30.06.2015.
[11]. Сайт платформы PandaBoard (http://pandaboard.org/) [HTML]. Дата обращения 30.06.2015.

Dynamic Java Program Analysis Using Virtual Machine Modification

M. Ermakov <mermakov@ispras.ru>
S. Vartanov <svartanov@ispras.ru>
Department of Computational Mathematics and Cybernethics, Moscow State University, 1/52, Leninskie Gory, Moscow, Russia, 119991

Abstract. At the present time automatic static and dynamic program analysis methods are extensively used during software development. Dynamic methods in particular allow cost-efficient detection of various runtime issues with minimal expert interaction, thus saving time and resources. Dynamic analysis methods for native applications employ external monitoring instruments and code modification in order to evaluate execution, while programs running under control of an interpreter or a virtual machine it offers a balanced observation layer – the interpreter or virtual machine itself. In this paper we focus on automatic memory profiling methods for Java applications running on Dalvik virtual machine – a part of rapidly growing Android operating system.

Current Dalvik VM limitations make it impossible to use complex Java profilers designed for desktop applications; Dalvik-compatible profiling tools provide insufficient data to perform deep memory usage analysis. In this paper we describe a set of extensions for Dalvik implementation of standard Java Debug Wire Protocol which allow tracking of in-depth program memory events: object allocation, garbage collection, memory read and write access. We present a debugger-level tool based on existing Dalvik Debug Monitor utility, that is compatible with the extended Dalvik VM and allows online tracking of aforementioned events along with base debugging features and gprof memory dump support. As the original DDM utility, our design supports concurrent tracking of multiple applications to identify memory usage issues related to interprocess communication.
We have used the developed system to analyze a set of standard Android applications identifying several memory usage trends, including prevalence of primitive array allocations and inefficient use of memory among standard Android GUI Java classes.

Keywords: dynamic program analysis; memory profiling; Java; Android

DOI: 10.15514/ISPRAS-2015-27(2)-2

For citation: Ermakov M.K., Vartanov S.P. Dynamic Java Program Analysis Using Virtual Machine Modification. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 23-38 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-2.

References

[1]. Stephen Kell, Danio Ansaloni, Walter Binder, Lukáš Marek. The JVM is not observable enough (and what to do about it). Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages, 2012. pp. 33-38. doi: 10.1145/2414740.2414747
[2]. Dtrace for HotSpot VM (http://docs.oracle.com/javase/6/docs/technotes/guides/vm/dtrace.html) [HTML]. Retrieved: 24.06.2015.
[3]. YourKit Java Profiler tool overview (https://www.yourkit.com/features/) [HTML]. Retrieved: 30.06.2015.
[4]. JProfiler tool overview (https://www.ej technologies.com/products/jprofiler/overview.html) [HTML]. Retrieved: 30.06.2015.
[5]. Eclipse MAT project (https://eclipse.org/mat/) [HTML]. Retrieved: 30.06.2015.
[6]. hprof tool documentation (http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html) [HTML]. Retrieved: 30.06.2015.
[7]. JDWP protocol overview (https://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html) [HTML]. Retrieved: 23.06.2015.
[8]. DDMS tool overview (http://developer.android.com/tools/debugging/ddms.html) [HTML]. Retrieved: 30.06.2015.
[9]. Paul R. Wilson. Uniprocessor garbage collection techniques. Proceedings of the International Workshop on Memory Management, 1992. pp. 1-42.
[10]. ddmlib project online repository (https://android.googlesource.com/platform/tools/base/+/master/ddmlib/) [HTML]. Retrieved: 30.06.2015.
[11]. PandaBoard project overview (http://pandaboard.org/) [HTML]. Retrieved: 30.06.2015.

Поиск состояний гонки в программах на языке Java при помощи динамического анализа

С. П. Вартанов <svartanov@ispras.ru>
М. К. Ермаков <mermakov@ispras.ru>
Факультет вычислительной математики и кибернетики,
Московский государственный университет им. М.В. Ломоносова,
119991, Россия, г. Москва, Ленинские горы, д. 1, с.52[1]

Аннотация. Язык программирования Java обладает встроенными средствами поддержки многопоточного выполнения программ. Из-за ошибок при создании подобных программ в процессе выполнения могут возникать ошибки синхронизации, одной из которых является возникновение состояния гонки между потоками, которые осуществляют доступ к разделяемому ресурсу (как минимум один поток модифицирует ресурс) и порядок работы потоков с ресурсом не фиксирован. В статье рассматривается подход к поиску состояний гонки при помощи динамического анализа. Преимущества динамического анализа по сравнению со статическим заключаются в отсутствии ложных срабатываний при определённых ограничениях, накладываемых на анализируемую программу. Для проведения динамического анализа предлагается использовать статическую инструментацию байт-кода программы, которая позволяет в ходе выполнения программы извлекать информацию о выполнении инструкций и методов, осуществляющих работу по синхронизации потоков. Построенная трасса представляет собой модель конкретного выполнения программы. На её основе с помощью отношений предшествования и механизма отслеживания блокировок определяется, возможна ли ситуация, при которой возникает состояние гонки. Для инструментации с целью сбора трассы используется инструмент динамического анализа Coffee Machine. Статическая инструментация, используемая в инструменте, позволяет проводить анализ программ на виртуальных машинах, не предоставляющих интерфейс для динамической инструментации. Анализ построенной модели и поиск потенциальных состояний гонки осуществляется при помощи инструмента ThreadSanitizer Offline. Благодаря использованию инструментации байт-кода наличие связи с исходным кодом для проведения анализа не является необходимым, однако это позволяет более точно определить причины возникновения

[1] Работа проводится при финансовой поддержке Российского фонда фундаментальных исследований, номер проекта 14-07-00609

ошибки. Реализация была проверена на ряде проектов с открытым исходным кодом и продемонстрировала свою эффективность для поиска состояний гонки.

Ключевые слова: динамический анализ, состояния гонки, ошибки синхронизации.

DOI: 10.15514/ISPRAS-2015-27(2)-3

Для цитирования: Вартанов С.П., Ермаков М.К. Поиск состояний гонки в программах на языке Java при помощи динамического анализа. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 39-52. DOI: 10.15514/ISPRAS-2015-27(2)-3.

1. Введение

В последнее время при создании программного обеспечения всё чаще используется многопоточность для увеличения производительности за счёт распараллеливания процессорных вычислений. Работа программы в нескольких потоках может способствовать увеличению её эффективности, но также приводит к росту сложности. Это в свою очередь затрудняет понимание работы программы, её разработку, поддержку и поиск в ней ошибок и уязвимостей.

Наиболее сложные для обнаружения программные ошибки при работе с потоками возникают, когда параллельно исполняемые потоки имеют доступ к одним и тем же данным или устройствам. В этом случае на программиста возлагается ответственность за обеспечение корректного взаимодействия между потоками. В противном случае в программе могут возникать ошибки, которые принято называть ошибками синхронизации. Такого рода ошибки могут быть разделены на четыре основных класса:

- взаимная блокировка (deadlock),
- состояние гонки (data race),
- нарушение атомарности операций,
- нарушение свойств детерминированности.

Для предотвращения ошибок синхронизации традиционно используется ряд механизмов, таких как: взаимоисключения, семафоры, события, критические секции и так далее. Корректное их использование может полностью устранить ошибки синхронизации, однако зачастую этому препятствуют размеры системы и сложность взаимодействия её компонентов.

Большинство современных языков программирования предоставляют широкий спектр механизмов для создания многопоточных программ. В этой статье будут рассмотрены принципы поиска состояний гонки в программах, которые представлены в виде байт-кода языка Java.

1.1 Механизмы создания многопоточных программ на языке Java

Для работы с потоками в Java используются стандартные классы java.lang.Thread и интерфейс java.lang.Runnable. Создание нового потока происходит с помощью создания класса, реализующего интерфейс Runnable, либо с помощью переопределения метода run() у потомка класса Thread. Управление потоками происходит с помощью методов run(), join(), sleep(). Методы stop(), suspend() и resume() были объявлены небезопасными, их использование не рекомендуется. Прерывание работы потока с помощью метода stop() приводит к принудительному снятию блокировок, которыми могут быть защищены объекты, находящиеся в несогласованном состоянии. [1] Использование методов suspend() и resume() зачастую приводит к возникновениям взаимных блокировок, поскольку ни один поток не может получить доступ к ресурсу, заблокированному прерванным потоком.

1.1.1 Синхронизация в Java

Синхронизация потоков

На уровне исходного кода для синхронизации потоков в библиотеке времени выполнения (runtime library) Java присутствует ряд стандартных механизмов. Методы wait() и notify() имеются у любого объекта и используются для ожидания одним потоком действий другого. Метод join() интерфейса Runnable позволяет потоку ожидать завершения другого потока.

Также методы классов из пакета java.util.concurrent представляют широкий спектр механизмов для межпоточного взаимодействия. Пакет содержит в себе реализацию таких синхронизационных примитивов, как семафора, циклического барьера и др.

Доступ к разделяемым ресурсам

Работа с мониторами отражена также в двух инструкциях Java байт-кода: monitorenter и monitorexit, которые считывают из вершины стека ссылку на объект, который собирается захватить монитор для выполнения следующих инструкций, либо, соответственно, освободить его.

Ключевое слово synchronized может быть применено к блоку инструкций программы или метода целиком. Работа его обеспечивается при помощи создания монитора для соответствующего блока. Выполнение секции возможно только для потока, объект которого захватил монитор. Ключевое слово volatile обязывает потоки использовать единственный экземпляр переменной.

1.2 Состояние гонки

В этой статье рассматриваются методы поиска ошибок типа состояния гонки. Под состоянием гонки мы будет понимать ситуацию, при которой два или более потоков обращаются к одним и тем же данным, причём как минимум один из них производит изменение этих данных и эти обращения не синхронизованы, т. е. последовательность их выполнения зависит исключительно от скорости работы отдельных компонентов программы и от принятия системным планировщиком решений о переключении потоков. В этом случае невозможно утверждать, произойдёт ли изменение данных одним потоком к моменту, когда они будут считаны или изменены другим потоком. Данный дефект может приводить как к незначительным нарушениям функциональности программы, так и к серьёзным программным сбоям вплоть до аварийного завершения.

Особенность этого типа дефектов заключается в том, что программная ошибка, к которой приводит дефект, проявляется нерегулярно и её обнаружение и воспроизведение может быть затруднительно как в процессе тестирования из-за особенностей работы планировщика потоков, так и в процессе отладки из-за различий в поведении программы при наличии отладчика.

1.3 Автоматический поиск ошибок

Методы автоматического поиска ошибок в программном обеспечении традиционно разделяют на методы динамического и статического анализа. Также, когда используются оба подхода, говорят о гибридном анализе. Под статическим анализом обычно понимают исследование программы без запуска её на выполнение, которое основывается исключительно на её исходном коде или другом представлении. В этом случае зачастую анализируется некоторая модель, построенная на основе исходного кода программы. Основными достоинствами статического анализа являются скорость его работы (вплоть до возможности анализа «на лету» в процессе редактирования исходного кода программы) и масштабируемость (возможность анализировать либо программу целиком, либо отдельные её части — модули, файлы, функции). К недостаткам статического анализа обычно относят возможность возникновения как ошибок первого рода (false positive), так и ошибок второго рода (false negative) связанных с недостаточной полнотой модели программы в статическом анализаторе.

1.3.1 Динамический анализ

Динамический анализ подразумевает запуск программы на исполнение. В ходе выполнения программы происходит анализ её состояния, проверка каких-либо условий, либо запись трассы событий. Динамический анализ отличается низкой скоростью работы из-за необходимости запуска программы

на выполнение, однако позволяет избежать ложных срабатываний в силу обнаружения ошибок непосредственно в процессе или по результатам работы программы.

Основным механизмом проведения динамического анализа является инструментация — процесс внедрения дополнительного программного кода с целью сбора информации о состоянии программы или трассе выполнения. Различают подходы статической и динамической инструментация. Статическая инструментация полностью завершается до начала выполнения программы, что позволяет избежать дополнительных накладных расходов в ходе выполнения программы. Динамическая инструментация производится непосредственно в ходе выполнения программы перед запуском целевого кода. Таким образом для её осуществления имеется вся необходимая информация.

Основной проблемой при проведении статической инструментации является необходимость на этапе до выполнения программы иметь необходимую информацию о целевых фрагментах кода и соответствующих инструментационных действиях.

2. Инструменты поиска синхронизационных ошибок

2.1 RoadRunner

Инструмент RoadRunner написан целиком на языке Java и представляет собой фреймворк для создания инструментов динамического анализа, предназначенных для поиска синхронизационных событий. Инструментация происходит на этапе загрузки классов при помощи библиотеки ASM [2]. Инструмент различает набор классов событий времени выполнения: доступ в память, блокировка, создание потока и т. д. Для проведения анализа используются так называемые теневые значения для объектов класса Thread, элементов памяти: полей классов, массивов и др. Создание инструментов на основе RoadRunner требует только описания обработчиков соответствующих событий.

Использование динамической инструментации требует дополнительных накладных расходов на этапе загрузки классов в виртуальной машине.

2.2 Sherlock

Инструмент Sherlock [3] представляет собой инструмент динамического анализа для поиска взаимных блокировок. Одним из основных понятий, используемых в инструменте является понятие расписания, которое определяется как последовательность событий. Для предварительного поиска потенциальных мест, в которых возможно возникновение взаимных блокировок, используется инструмент GoodLock.

Sherlock реализует механизм итеративного динамического анализа. На первой итерации программа выполняется на начальных входных данных. На каждой следующей итерации происходит выполнение инструментированной версии программы и записывается трасса, в которую входят синхронизационные события, а также информация об ограничениях, накладываемых на путь выполнения. Полученное в итоге расписание преобразуется с целью достижения ситуации взаимной блокировки потоков. Для воспроизведения полученного расписания строятся наборы входных данных для программы при помощи SMT-решателя на основе построенных ограничений пути. Это становится возможным за счёт контролирования инструментом системного планировщика потоков.

2.3 ThreadSanitizer

Проект ThreadSanitizer посвящён поиску состояний гонки в программах. Он представляет собой набор инструментов динамического анализа, которые производят построение модели конкретного пути выполнения программы и на основе этой модели осуществляют поиск состояний гонки.

В проекте используются два основных подхода к поиску состояний гонки:

- установление отношений предшествования между синхронизационными событиями,
- отслеживание множества блокировок для каждого события.

Основной инструмент проекта осуществляет поиск состояний гонки в программах, представленных в бинарном коде. В первых версиях инструмента этот анализ происходил на основе фреймворков динамической инструментации Valgrind [7] и PIN [8]. Более поздняя версия основывается на статической инструментации, что позволяет снизить накладные расходы.

В проект также входят два экспериментальных инструмента. Один из них — ThreadSanitizer Offline — представляет из тебя утилиту, которая на основе трассы событий делает вывод о наличии или отсутствии в ней состояния гонки. Второй — Java ThreadSanitizer — экспериментальный динамический анализатор, который использует интерфейс утилиты ThreadSanitizer Offline для поиска состояний гонки в программах на языке Java.

В инструменте Java ThreadSanitizer применяется динамическая инструментация при помощи библиотеки ASM [2]. Это не позволяет применять инструмент для поиска ошибок на программах, которые запускаются на виртуальных машинах с отсутствующей опцией javaagent. Динамическая инструментация также приводит к дополнительным накладным расходам в ходе выполнения программы, которых можно избежать с применением инструментации на начальном этапе анализа, поэтому от использования инструмента Java ThreadSanitizer было решено отказаться.

3. Используемые подходы

3.1 Отношение предшествования

Отношение предшествования определяется как отношение строгого частичного порядка на множестве событий на одном из путей выполнения программы. Т. е. такое отношение, что

$$\forall x : x \prec x,$$
$$\forall x, y : x \prec y \wedge y \prec x \Rightarrow x = y,$$
$$\forall x, y, z : x \prec y \wedge y \prec z \Rightarrow x \prec z.$$

Согласно определению, событие A предшествует событию B, если оно предшествует ему в рамках одного потока, либо событие A — событие отправки сообщения, а событие B — приёма того же сообщения, либо существует событие C, такое что A предшествует C, а C предшествует B (свойство транзитивности). Другими словами, событие A предшествует событию B, если событие A может повлиять на событие B. [4]

3.2 Отслеживание блокировок

Модель отношений предшествования учитывает синхронизацию событий при помощи отправки и получения сообщений. Для синхронизации с использованием блокировок используется механизм, который строит для каждого объекта в ходе выполнения программы множество блокировок, которые захвачены объектом в данный момент.

Совместное использование двух описанных механизмов позволяет инструменту ThreadSanitizer производить эффективный поиск состояний гонки. В терминах ThreadSanitizer состоянием гонки считается ситуация, при которой существуют два или более событий доступа к одним данным, которые происходят в разных потоках, по крайней мере одно из которых — операция записи и которые не связаны отношением предшествования.

4 Реализация поиска состояний гонки

Поиск состояний гонки осуществляется при помощи совмещения работы двух инструментов: Coffee Machine и ThreadSanitizer Offline.

4.1 Инструментация

Модульная структура инструмента Coffee Machine позволяет реализовать на его основе компоненты для печати трассы с синхронизационными событиями в ходе выполнения программы.

В качестве параметров инструменту Coffee Machine передаётся путь к директории, в которой располагаются класс-файлы либо JAR-файлы, в которых содержится целевой байт-код, а также список методов, которые

необходимо проинструментировать.

В зависимости от того, известен ли пользователю исчерпывающий список методов, которые будут вызваны в ходе выполнения программы, передаваемый список методов воспринимается иснтрументом либо как окончательный, либо как начальный набор единиц инструментации. В первом случае процесс инструментации заканчивается после прохода по полученному списку. Во втором случае запускается итеративный процесс поиска необходимых методов, на каждой итерации которого выбранный из списка метод инструментируется, удаляется из списка, и в список добавляются все методы, которые не были проинструментированы и вызовы которых присутствуют в текущем методе. Во втором случае, в зависимости от особенностей анализируемой программы, может быть проинструментировано значительное количество методов, которые никогда не будут вызваны в силу их недостижимости при заданных входных данных программы, однако следует заметить, что это влияет лишь на размер результирующих класс-файлов и на время работы инструментатора на начальном этапе. В силу того, что методы в языке Java могут быть вызваны динамически, статическое построение полного списка может быть невозможно. В связи с этим, если в ходе выполнения программы, вызывается метод, который не был ранее проинструментирован, выполнение может быть прервано для дополнительной инструментации.

4.1.1 Инструментация отдельных инструкций

Инструментация представляет собой внедрение дополнительной функциональности в байт-код таким образом, чтобы это в как можно меньшей степени влияло на изначальную функциональность анализируемой программы. О полном отсутствии влияния говорить не приходится, поскольку любые дополнительные действия в любом случае занимают процессорное время, что приводит к замедлению работы программы, часто неравномерному. Этот эффект является одной из основных проблем динамического анализа в целом и может решаться различными способами, в зависимости от задачи, которая встаёт перед анализом. Так, для избавления от неравномерности замедления, в программу могут быть принудительно добавлены задержки, либо же для нивелирования воздействия инструментации на системные механизмы они могут быть откалиброваны с учётом особенностей инструментации на уровне интерпретатора языка или операционной системы. К таким механизмам можно отнести системные часы, планировщик выполнения потоков, сборщик мусора и т. д.

Для выбранного подхода поиска ситуаций состояния гонки эта проблема актуальна в меньше степени из-за особенностей механизма проверки, построенной в ходе выполнения модели работы программы. В силу этих особенностей, результирующая модель не зависит от работы системного планировщика выполнения потоков и поэтому влиянием временных задержек можно пренебречь.

Инструмент Coffee Machine посредством использования библиотеки BCEL [5] предоставляет интерфейс для внедрения в байт-код для каждого типа инструкции при определённых условиях двух дополнительных наборов инструкций: предварительного и заключительного. Каждый из двух наборов при этом может быть пустым. Если оба набора пусты, это означает, что для данного типа инструкций инструментационного кода не создаётся и этот тип не представляет интереса с точки зрения анализа программы.

В рамках архитектуры инструмента Coffee Machine для определённого типа инструментации создаётся собственный класс. Класс для печати синхронизационных событий имеет имя Concurrency и содержит в себе всю необходимую функциональность. Непосредственно в предварительный и заключительный наборы инструкций, которые будут добавлены до и после целевой инструкции соответственно, входят инструкции дублирования элементов стека, загрузки константных значений и инструкция вызова инструментационной функции класса Concurrency.

Например, для инструкции байт-кода monitorenter, существуют оба набора инструкций. Предварительный набор состоит из одной инструкции дублирования вершины стека (DUP) для передачи в инструментационный код ссылки на объект, а в заключительный набор входят инструкции добавления в стек текущего значения симулятора счётчика команд (PUSH) и инструкция вызова метода monitorEnter(Object, int) класса Concurrency (инструкция INVOKESTATIC). Таким образом, вместо одной инструкции monitorenter, в байт-коде инструментированной программы будут четыре инструкции:

```
DUP
MONITORENTER
PUSH <program counter>
INVOKESTATIC
   ru.ispras.coffeemachine.target.Concurrency.monitorEnter
   (Ljava/lang/Object;I)V
```

Листинг 1. Список инструкций байт-кода, на которые заменяется инструкция MONITORENTER

Хотя бы один из инструментационных наборов не пуст для следующих инструкций:

загрузки значения в стек,сохранения значения в локальную переменную,вызова функции и выход из неё,обращения к полям объектов,выброса исключенияmonitorenter, monitorexit.

Оба блока инструментационного кода будут сгенерированы для инструкций вызова функции и инструкции monitorenter.

Методы класса Concurrency, вызовы которых присутствуют в инструментационном коде, производят действия по обработке переданных им параметров, обновлению соответствующих объектов и печати строки

синхронизационного события. Информация о синхронизационных событиях записывается в стандартный поток вывода с определённым префиксом, что позволяет отличить её от вывода программы. В этой строке содержится тип события, номер текущего потока, значение симулятора счётчика команд и дополнительные параметры события, если такие есть. Формат данных соответствует формату входных файлов инструмента ThreadSanitizer Offline, который производит построение модели выполнения программы.

```
...
UNLOCK 0 8be 2a675b7986 0
SBLOCK_ENTER 0 8c 0 0
WRITE 0 8c 2f9ee1ac00000003 1
SBLOCK_ENTER 0 8c 0 0
RTN_CALL 0 0 0 0
SBLOCK_ENTER 0 88 0 0
RTN_EXIT 0 88 0 0
SBLOCK_ENTER 0 8c 0 0
READ 0 8c 2f9ee1ac00000000 1
THR_START 1 0 0 0
THR_FIRST_INSN 1 8c 0 0
SBLOCK_ENTER 0 8c 0 0
...
```

Листинг 2. Трасса синхронизационных событий

В приведённом примере присутствуют события обращения к объекту на чтение (READ), запись (WRITE), вызов функции (RTN_CALL) и выход из неё (RTN_EXIT), снятие блокировки (UNLOCK), начало нового потока (THR_START и THR_FIRST_INSN).

Симулятор счётчика команд — простая целочисленная статическая переменная класса-инструментатора, её значение известно на этапе инструментирования кода и в инструментационном коде она представляет собой константное значение. Это значение будет добавлено в строковое представление инструментационного события и будет использовано для связывания этого события с инструкцией байт-кода, которая привела к возникновению этого события. Инструкция байт-кода в свою очередь при наличии соответствующей информации в класс-файле может быть связана со строкой исходного кода программы, что позволит определить место возникновения событий, которые приводят к состоянию гонки. Номер потока вычисляется динамически, в ходе исполнения инструментационного кода.

4.1.2 Инструментация инструкций вызова метода

Особая работа производится с инструкциями вызова метода. Инструментация этой инструкции не может быть универсальной и производится в основном для покрытия функциональности методов из пакета java.util.concurrent. Также

инструментация производится для вызовов базовых механизмов взаимодействия потоков из пакета java.lang и для вызовов набора методов пакета org.jtsan, которые могут быть встроены пользователем в программу для управления ходом работы анализа.

До операции вызова в байт-код внедряются операции дублирования вершины стека — параметров вызываемого метода. Если инструментационные действия необходимо производить и до, и после вызова, вершина стека дублируется дважды. Затем происходит вызов предварительного инструментационного кода, если он не пуст. После выполнения возвращение из целевого метода происходит вызов заключительного инструментационного кода, опять же, если он не пуст. Метод, реализующий заключительный набор инструментационных инструкций, обязан возвращать значение метода, которое он получает на вход вместе с продублированными параметрами, если метод не объявлен с ключевым словом void.

Из пакета java.util.concurrent инструментируются методы классов

- locks.AbstractQueuedSynchronizer,
- locks.Condition,
- locks.ReentrantLock,
- locks.ReentrantReadWriteLock,
- locks.ReentrantReadWriteLock_ReadLock,
- locks.ReentrantReadWriteLock_WriteLock,
- locks.LockSupport,
- CountDownLatch,
- CyclicBarrier и
- Semaphore.

Также частично осуществляется поддержка системных коллекций.

4.2 Поиск ошибок

Сгенерированная описанным путём трасса подаётся на вход инструменту ThreadSanitizer Offline. В случае обнаружения состояния гонки, инструмент выдаёт сообщение, в котором указывается тип ошибки: сколько потоков участвует в гонке, какие из потоков обращаются к данным посредством чтения, а какие посредством записи. Также добавленные в трассу ссылки на исходный код программы, если он доступен, указывают программисту на место в коде, где произошла ошибка.

5. Результаты

Работа инструмента была проверена на ряде проектов с открытым исходным кодом. Для проверки базовой функциональности инструмента был расширен набор стандартных тестов инструмента Java ThreadSanitizer. Инструмент

продемонстрировал эффективность поиска дефектов, как на проектах с искусственно внесёнными ошибками, так и на проектах без модификаций. Основные исследования проводились на стандартных приложениях платформы Android, которые в силу использования графического пользовательского интерфейса активно используют механизмы многопоточности.

В результате были обнаружены ситуации состояния гонки, одна из которых приводит к аварийному завершению приложения. Для воспроизведения обнаруженных дефектов на сегодняшний момент автоматические средства не реализованы, и эта задача решается вручную путём внедрения в программу временных задержек. Наличие в класс-файлах информации о связи байт-кода с исходным кодом программы не является обязательным для работы механизма поиска дефектов, однако это существенно упрощает для разработчика поиск причин возникновения дефекта и его воспроизведения.

6. Заключение

В статье была рассмотрена реализация механизма поиска состояний гонки в программах на языке Java, либо программах, которые могут быть конвертированы в Java байт-код. Механизм представляет собой динамический анализ трассы выполнения программы с её предварительной статической инструментацией, — набора событий, которые представляют интерес с точки зрения взаимодействия между потоками. На основе полученной трассы сторонним инструментом ThreadSanitizer Offline производится построение модели выполнения программы и поиск состояний гонки. Проведённые исследования показали эффективность работы инструмента для поиска состояний гонки в многопоточных программах.

Список литературы

[1]. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7), 1978. pp. 558–565. doi: 10.1145/359545.359563

[2]. Java Thread Primitive Deprecation [HTML] (http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html)

[3]. K. Serebryany and T. Iskhodzhanov. ThreadSanitizer – data race detection in practice. Proceedings of the Workshop on Binary Instrumentation and Applications, 2009. pp. 62-71. doi: 10.1145/1791194.1791203

[4]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)

[5]. Bruneton E. ASM 4.0. A Java bytecode engineering library, 2011 [PDF] (http://download.forge.objectweb.org/asm/asm4-guide.pdf)

[6]. M. Eslamimehr, J. Palsberg. Sherlock: scalable deadlock detection for concurrent programs // Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014. P. 353–365

[7]. Nethercote N., Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation // PLDI. 2007.

[8]. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation // PLDI. 2005.

Detecting Race Conditions in Java Programs Using Dynamic Analysis

S. Vartanov <svartanov@ispras.ru>
M. Ermakov <mermakov@ispras.ru>
Department of Computational Mathematics and Cybernethics, Moscow State University, 1/52, Leninskie Gory, Moscow, Russia, 119991

Abstract. Multithreading support is one of Java programming language built-in features. Synchronization defects are a serious issue despite the extensive thread processing API provided by standard classes. One of the most common synchronization defects is a race condition. It arises when two different threads read and modify shared data and data access order is not fixed. This paper focuses on an approach to automatically detect race conditions using dynamic analysis. The main advantage of dynamic analysis is the lack of false positives if program under analysis fits a number of requirements. Our approach for dynamic analysis uses static bytecode instrumentation to extract execution trace at run-time which allows to track usage for thread synchronization methods and functions. Generated trace is a concrete execution model which is processed to identify happen-before relations and lock sets for possible data race detection. We use Coffee Machine tool for static instrumentation. Static instrumentation approach is applicable even to virtual machines which do not provide support for dynamic instrumentation. We use ThreadSanitizer Offline to process generated trace and detect race conditions. While debug information in bytecode is unnecessary it provides more precise error messages. Described tool chain has been tested on a set of open source projects.

Keywords: dynamic analysis, race condition, synchronization defects.

DOI: 10.15514/ISPRAS-2015-27(2)-3

For citation: Vartanov S.P., Ermakov M.K. Detecting Race Conditions in Java Programs Using Dynamic Analysis. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 39-52 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-3.

References

[1]. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7), 1978. pp. 558–565. doi: 10.1145/359545.359563

[2]. Java Thread Primitive Deprecation [HTML] (http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html)

[3]. K. Serebryany and T. Iskhodzhanov. ThreadSanitizer – data race detection in practice. Proceedings of the Workshop on Binary Instrumentation and Applications, 2009. pp. 62-71. doi: 10.1145/1791194.1791203

[4]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)

[5]. Bruneton E. ASM 4.0. A Java bytecode engineering library, 2011 [PDF] (http://download.forge.objectweb.org/asm/asm4-guide.pdf)

[6]. M. Eslamimehr, J. Palsberg. Sherlock: scalable deadlock detection for concurrent programs // Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014. P. 353–365

[7]. Nethercote N., Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation // PLDI. 2007.

[8]. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation // PLDI. 2005.

Анализ сущностей программ на языках Си/Си++ и связей между ними для понимания программ *

А.А. Белеванцев <abel@ispras.ru>
Е.А. Велесевич <evel@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. В статье рассматривается инструмент статического анализа программ, определяющий сущности программы на языке Си или Си++, их метрики и связи между ними. Сущностями программы являются файлы, функции, классы, методы и т.п., а связями – вызовы, наследование, чтение/запись глобальных переменных, включение, агрегация. Необходимость построения такого инструмента возникает в связи с тем, что для широкого круга задач понимания программ основой для построения решения является автоматическое извлечение необходимой информации о программе из ее исходных кодов. Инструмент должен поддерживать все конструкции языков Си и Си++ и масштабироваться для анализа реальных программных систем в миллионы строк исходного кода.

Статья посвящена методам построения такого инструмента на основе открытой компиляторной инфраструктуры LLVM[1]. Для разбора текстов программ используется промышленный компилятор Clang[2], а для последующего анализа – собственный инструмент на основе LLVM, обеспечивающий консолидацию информации обо всей программе. Так как окончательный анализ выполняется на уровне внутреннего представления (биткода) LLVM, то необходимо обеспечить сохранение в файлах с этим представлением дополнительной информации уровня исходного кода, теряющейся при изначальной трансляции. Для этого, а также для поддержки разнообразных диалектов Си и Си++, было выполнено более 400 доработок компилятора Clang[2]. В анализаторе уровня биткода центральной частью является компоновщик, задачей которого является объединение информации об одних и тех сущностях, участвующих в сборке различных компонент программной системы (например, библиотеки и приложения, ее использующего). Построенные с использованием компоновщика связи между сущностями позволяют точнее отследить отношения между компонентами всей системы. В статье также представляются результаты тестирования инструмента на коде ОС Android.

Ключевые слова: понимание программ; LLVM; статический анализ; метрики исходного кода.

* Работа поддержана грантом РФФИ 14-01-31363 мол_а.

DOI: 10.15514/ISPRAS-2015-27(2)-4

Для цитирования: Белеванцев А.А., Велесевич Е.А. Анализ сущностей программ на языках Си/Си++ и связей между ними для понимания программ. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 53-64. DOI: 10.15514/ISPRAS-2015-27(2)-4.

1. Введение

В жизненном цикле больших программных систем, особенно при их длительном использовании, перед разработчиками возникают задачи, которые можно объединить под термином *понимания* программ – получения некоторой информации о программе, тем или иным способом облегчающей работу с ней. Например, требуется доработка системы под новые требования пользователей, однако имеющаяся документация неполна или вообще отсутствует; или же необходимо выполнить рефакторинг кода системы или изменение ее архитектуры для обеспечения возможности дальнейших доработок. Наконец, важной задачей, особенно актуальной из-за развития открытого ПО, является поддержание в актуальном состоянии собственных доработок некоторой крупной открытой программы и перенос этих доработок на новые версии программы (а иногда и на более старые).

Во всех описанных случаях (естественно, в предположении, что доступен исходный код программной системы) решение задачи можно разделить на две части. Во-первых, необходимо собрать информацию о структуре программы – частей, из которых она состоит, их свойств и характеристик, а также об их организации (иерархии и связях). Во-вторых, нужно придумать, как использовать собранную информацию для решения исходной задачи.

Например, в случае использования компонента системы при отсутствующей документации можно собрать информацию об экспортируемых этим компонентом функциях и типах данных, установить способы вызовов этих функций из остальных частей системы, и после ручного анализа сформулировать контракты на использование компонента. Если этого недостаточно, то можно получить информацию об использовании типов данных и функций в исходном коде самого компонента. Для рефакторинга кода, помимо аналогичной информации о функциях и их вызовах другими функциями, использовании функциями глобальных структур данных, может пригодиться статистика сложности отдельных частей исходного кода и их связей: например, если функция вызывает слишком много других, или ее управляющие конструкции имеют большую вложенность, то, возможно, ее нужно разбить на несколько вспомогательных функций. Наконец, для переноса доработок между версиями программы можно взять используемые в доработках функции основной системы, собрать данные об этих функциях в обеих версиях системы и сравнить их, тем самым найти изменившиеся компоненты системы и по их связям с остальным кодом понять, как нужно обновить собственные доработки.

Как видно, все описанные действия ручного анализа структуры программы предполагают возможность автоматической сборки информации о ней. Необходимую информацию можно обобщить следующим образом:

- *сущности* программы: физические – файлы, каталоги – и логические, в зависимости от заданного языка программирования, – функции, классы, поля, переменные, методы;
- *свойства* этих сущностей: имя, прототип (для функций), место определения в программе, родительская сущность, другие различные атрибуты;
- *количественные характеристики* сущностей: размер, количество вложенных подсущностей, количество определенных конструкций в исходном коде;
- *связи* между сущностями: вызовы функций, чтение и запись глобальных переменных и статических полей классов, наследование классов, агрегация (класс содержит поля, файл включает другой файл).

В работе описывается инструмент, предназначенный для сбора этой базовой информации о программе. При наличии такого инструмента, решающего первую часть описанных задач понимания программ, над ним можно строить другие инструменты, автоматизирующие применение этой информации для решения конкретной задачи. В любом случае это применение требует ручного труда программиста. Однако сейчас даже сбор описанной основной информации не производится автоматически. Некоторую ее часть строят современные среды разработки, но не полностью; при этом для программных систем в миллионы строк кода среды разработки не масштабируются – можно исследовать исходный код системы только покомпонентно, а связи между компонентами будут утеряны. Специализированных открытых инструментов анализа подобного рода нет, но на рынке есть ряд коммерческих продуктов – системы Klocwork Architect, Understand[3], Imagix4D[4]. Насколько можно установить по доступной информации об этих системах, их возможности и поддерживаемые языки пересекаются не полностью, т.е. нет одной системы, которая полностью покрывает потребности разработчиков.

Предлагаемый нами инструмент поддерживает языки Си и Си++, а в будущем и язык Java, и анализирует программы для ОС Linux объемом в миллионы строк кода (например, полный исходный код ОС Android). При разработке инструмента были максимально использованы существующие открытые программные продукты – компилятор Clang, инфраструктура LLVM, компоновщик и архиватор из пакета GNU Binutils [7]. Также были использованы компоненты инструмента статического анализа Svace[5-6], разработка которого ведется в ИСП РАН. Это позволило создать прототип инструмента в сжатые сроки.

Далее в разделе 2 статьи описывается архитектура инструмента и его компоненты, в разделе 3 предлагается список вычисляемых метрик и связей и особенности необходимых для этого вычисления алгоритмов анализа, в разделе 4 содержатся некоторые экспериментальные результаты. Раздел 5 завершает статью.

2. Архитектура инструмента анализа

Для разработки описанного инструмента анализа наилучшим образом подходит инфраструктура компилятора. Поиск сущностей программы и связей между ними требует разбора программы, который хорошо выполняет компилятор, и применения анализа потока управления и потока данных, которые, как правило, уже используются в оптимизационных проходах компилятора. В настоящее время единственными промышленными компиляторами с открытым исходным кодом для языков Си и Си++ являются компиляторы GCC и Clang/LLVM. Оба компилятора поддерживают последние версии стандартов языков, содержат удобное внутреннее представление для анализа и необходимые средства. Тем не менее, для нашего анализатора удобнее система LLVM, т.к. требуется сохранять вместе с промежуточным представлением программы собственную информацию о сущностях и метриках, а в LLVM для этого есть стандартный расширяемый формат – т.н. *метаданные*, которые могут создаваться компилятором, сохраняться на диск и впоследствии считываться анализатором.

Инструмент состоит из трех частей, каждая из которых отвечает за один из трех этапов его работы. На первом этапе требуется организовать автоматическое построение необходимого внутреннего представления программы – многократный запуск компилятора вручную для каждого исходного файла неудобен пользователю. Для автоматического создания представления программы нужно знать, как организована её сборка (т.е. как именно и для каких исходных файлов запускаются компиляторы, компоновщики и другие инструменты, чтобы получить искомое приложение). Тогда для каждого исходного запуска компилятора можно параллельно запустить собственный компилятор, который сгенерирует необходимые данные. Чтобы не зависеть от конкретной системы сборки, такая задача решается универсальным *мониторингом сборки*: организуется перехват запуска всех процессов операционной системы; при запуске анализируется командная строка процесса и определяется, имеет ли запущенный процесс отношение к сборке программы (т.е. является ли он компилятором, компоновщиком или подобной утилитой); если процесс успешно распознан, то выполняются действия по запуску собственного компилятора или компоновщика. Мониторинг процесса сборки является единственным существенно зависящим от операционной системы компонентом: реализация перехвата запуска процессов различна в ОС семейства Linux и Windows. В первом случае используется динамическая библиотека, загружаемая до старта

программы (т.н. механизм LD_PRELOAD), во втором случае – запуск процессов под отладкой.

Действия по запуску своего компилятора зависят от типа перехваченного компилятора: требуется фильтрация исходной командной строки для удаления несовместимых опций компилятора, преобразования одинаковых опций с разным написанием из исходного формата в собственный, преобразования пути выходного файла в собственный, передача стандартных путей поиска включаемых файлов исходного компилятора собственному и т.п. Общая цель этих изменений – добиться того, чтобы свой компилятор максимально точно соответствовал настройкам исходного, чтобы собрать именно то приложение, которое подается на вход исходной системе сборки. Из-за гибкости и широких возможностей современных компиляторов полностью повторить одним собственным компилятором несколько других во всех случаях невозможно, и для корректной сборки программ большого объема требуются существенные усилия. Наш анализатор использует мониторинг сборки инструмента анализа Svace, разрабатываемого в ИСП РАН, что минимизирует необходимые доработки.

На втором этапе запускается собственный компилятор на базе Clang, который строит искомое внутреннее представление программы. Наш компилятор существенно модифицирован по сравнению с исходным – выполнено более 300 изменений. Во-первых, это изменения, поддерживающие совместимость с другими популярными компиляторами – либо требуется реализовать расширения других компиляторов в собственном, либо снизить требования к соответствию компилируемой программы стандарту языка. Во-вторых, изменения, требующиеся для вычисления метрик исходного кода, которые можно выполнить только в компиляторе. Такие изменения включают в себя, например, модификацию лексического и синтаксического разбора программы для учета строк с комментариями и без комментариев, пробельных строк и т.п. Наконец, последней частью изменений является код для создания собственных данных (в формате метаданных инфраструктуры LLVM), которые хранят информацию, необходимую для анализатора скомпонованных приложений во внутреннем представлении LLVM (т.н. *биткоде*). Частью этой информации является стандартная отладочная информация, расширенная для нужд анализатора, уже вычисленные метрики и отношения, дополнительная информация о вызовах (например, неявные вызовы) и т.п. Основным принципом являлось максимальное использование имеющейся отладочной информации, и добавления новых метаданных были минимальными. Однако из-за объема собранных данных итоговый размер файлов с биткодами мог превышать первоначальный в 3-4 раза.

Окончанием второго этапа работы инструмента является компоновка сгенерированных файлов с биткодом вместе так же, как объектные файлы компонуются в приложения и библиотеки. Это делается, чтобы анализатору на третьем этапе предъявлялись уже скомпонованные файлы со связями между

символами программы, полностью соответствующими исходным. За компоновку отвечает отдельный модуль анализатора, управляющий ее ходом согласно собранной на первом этапе трассе сборки, где отмечены все запуски компиляторов, ассемблеров, компоновщиков и архиваторов. При этом поддерживаются правильные отображения файлов с исходным кодом, ассемблером, объектным кодом, библиотеками и приложения друг в друга. Компоновка биткода может вестись параллельно, зависимости между необходимыми запусками компоновщика автоматически вычисляются управляющим модулем по трассе сборки.

Например, пусть в исходной программы файлы `a.o` и `b.o` (полученные из исходных файлов `a.c` и `b.c`) компонуются в библиотеку `lib.so`, а потом эта библиотека компонуется с файлом `main.o` (полученного из файла `main.cpp`) в исполняемый файл приложения `main`. Тогда свой компилятор должен обеспечить компиляцию файлов `a.c`, `b.c` и `main.cpp` в файлы `a.o.bc`, `b.o.bc` и `main.o.bc` соответственно, а компоновщик должен скомпоновать файлы `a.o.bc` и `b.o.bc` в файл `lib.so.bc`, а потом этот файл и `main.o.bc` – в файл `main.bc` (файлы с биткодом LLVM традиционно имеют расширение `.bc`). Полученные файлы библиотеки и приложения в формате LLVM будут переданы анализатору.

Наконец, третья компонента инструмента представляет из себя собственно анализатор файлов с биткодом LLVM. На вход анализатору подается список всех скомпонованных в ходе сборки проекта файлов (библиотек или исполняемых файлов). Анализатор по очереди считывает файлы и метаданные в них стандартными средствами LLVM. После этого для всех упомянутых в файле сущностей создается их представление в памяти, сохраняются метрики и проводятся связи необходимых типов.

Для корректного представления информации о сущностях (в особенности о месте определения сущности в файле) анализатор должен уметь объединять информацию об одной и той же сущности, представленную в разных файлах. Например, при анализе приложения известно, что функция приложения `foo` вызывает функцию `bar`, объявленную в файле `lib.h`, определение функции `bar` недоступно. Если код библиотеки `lib` также недоступен, то более точной информации о функции `bar` получить не удастся, иначе при анализе библиотеки `lib` нужно уметь определять, что функция `bar` из файла `bar.c`, объявленная в файле `lib.h`, – это та же самая функция, что уже была встречена при анализе функции `foo` из основного приложения, то есть одна сущность. Ее свойства, полученные при анализе приложения и библиотеки, нужно объединить вместе. Эта функциональность анализатора аналогична компоновщику, но не в рамках одного приложения, а в рамках программной системы в целом, и является центральной частью анализатора, благодаря которой можно получить корректную информацию обо всей системе со связями между ее компонентами.

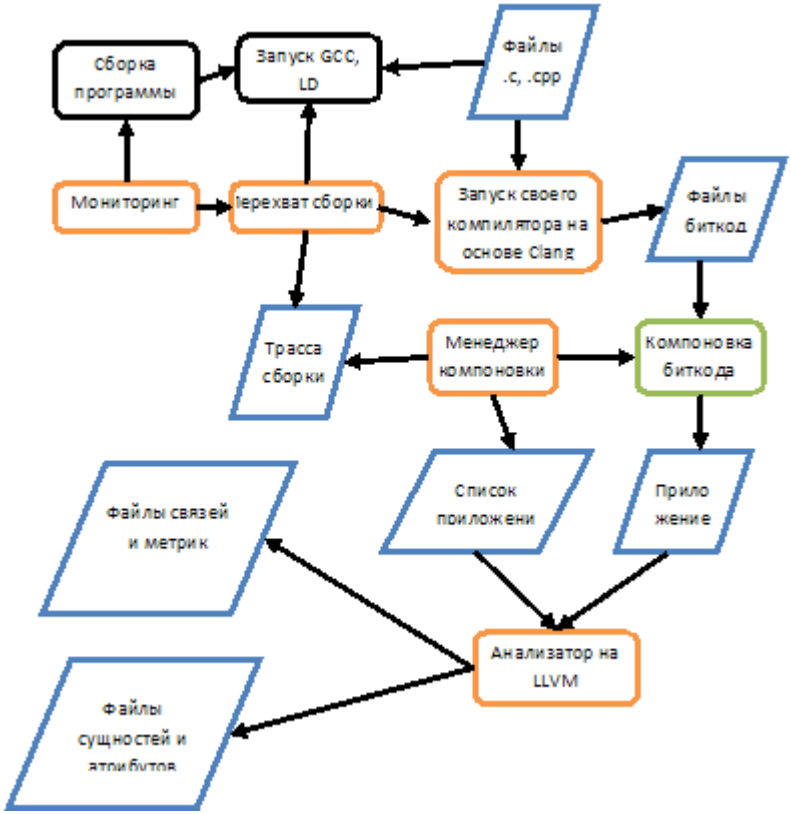

Рис. 1. Архитектура и схема работы предложенного анализатора

Схема работы описываемого инструмента и его архитектура представлены на рисунке 1. Черными прямоугольниками выделены процессы исходной сборки, оранжевым – компоненты анализатора, синим – обрабатываемые данные. Зеленым прямоугольником отмечен используемый компоновщик Gold из пакета GNU Binutils, никак не модифицированный в ходе работы над инструментом (настраивались лишь параметры компоновщика).

3. Сущности программы, их метрики и связи между ними

Поддерживаемыми инструментом сущностями являются: физическими – файл, каталог, строка в файле (при указании места возникновения связи или определения сущности); логическими – глобальные переменные, поля и методы, классы, структуры, функции, макросы, перечислимые и другие сложные типы. Атрибутами сущностей является их место определения, родительская сущность и специфическая информация – прототип для функции, модификаторы доступа для полей, методов и классов и т.п.

Метрики, вычисляемые инструментом, можно классифицировать следующим образом. Во-первых, это метрики физического и логического размеров сущностей (количество строк в функции или файле с учетом/без учета пробелов и комментариев, количество полей/методов в классе, средний/максимальный размер функции в файле). Во-вторых, это метрики сложности сущности (цикломатическая сложность, уровень вложенности управляющих конструкций). Наконец, есть метрики, характеризующие связи между сущностями (количество вызовов данной функции и вызовов других функций из данной). Всего реализовано более 30 различных метрик. Некоторые из них просто агрегируют соответствующие метрики по всем сыновним подсущностям – так, есть метрика среднего и максимального размеров как метода, так и класса; как функции, так и файла и каталога.

Можно заметить, что все метрики так или иначе отражают «сложность» сущности, и для каждой из них можно эмпирически установить некоторый порог, при превышении которого требуется анализ сущности и по возможности ее упрощение или декомпозиция на несколько других сущностей. Наше исследование не ставит себе целью определение таких порогов для каждой метрики, так как эти значения могут отличаться для разных разработчиков и проектов; анализ порогов сложности может выполняться на втором этапе решения задачи понимания программы – при постобработке собранных нашим инструментом данных.

Основными характеристиками сущностей программы являются не метрики, а связи между сущностями. Для функций и методов такой связью является вызов других функций (возможно – неявный вызов, например, конструктора объекта при входе в локальный блок, где объявлена переменная объекта), для глобальных переменных – их чтение или запись. Для построения иерархии сущностей определяется связь «сущность содержит сущность» (класс содержит метод, файл содержит глобальную переменную). Кроме того, для Си++ возникают особые типы связей, характеризующие иерархию классов – связи по наследованию классов и переопределению методов.

Наиболее сложный анализ (потоков управления и данных) требуется как раз при определении связей между сущностями. Для определения вызовов между функциями строится граф вызовов, в том числе с учетом вызовов по указателю. Для Си++ и в будущем для Java важную роль играет девиртуализация вызовов, позволяющая получить более точные данные о вызываемой функции. В будущем планируется использовать в инструменте независимо разработанный в ИСП РАН алгоритм девиртуализации для инфраструктуры LLVM.

Для определения вызовов функций из динамических библиотек (например, с помощью интерфейса `dlopen` ОС Linux) требуется минимальный анализ потока данных и анализ указателей для установления связи между указателем на функцию и переменной, содержащей имя вызываемой функции. Это

позволяет, например, поддержать случаи, когда имеется константный массив имен функций, которому в соответствие ставится массив указателей на функции динамической библиотеки. При этом при вызове функции по некоторому указателю из массива известно ее имя и название библиотеки, полученной через анализ вызова функции `dlopen`.

Наконец, для анализа иерархии сущностей строится граф включения исходных файлов друг в друга и граф наследования между классами программы. Граф наследования используется также для поиска связей переопределения методов и для работы алгоритма девиртуализации.

Анализ остальных связей и метрик, как правило, не доставляет большого труда. Основной задачей анализатора здесь является, как уже было указано, правильное определение эквивалентности сущностей из разных приложений и соответственное объединение их атрибутов, метрик и связей. Кроме этого, анализатор агрегирует составные метрики сущностей из метрик подсущностей (например, среднюю цикломатическую сложность класса из цикломатических сложностей его методов).

4. Экспериментальные результаты

Разработанный инструмент был опробован нами на исходном коде ряда открытых проектов, включая код ОС Android, ядра ОС Linux. Наиболее затратным по времени этапом является компоновка файлов с внутренним представлением, т.к. размер созданных метаданных большой, а инфраструктура LLVM плохо сливает одинаковую информацию из разных файлов (в том числе отладочную). Мы планируем выполнить исследования по разработке алгоритмов лучшего слияния такой информации. Фаза компиляции внутреннего представления и собственно анализа занимает время, сравнимое со временем сборки исходного кода проекта. Точность и полнота получаемых данных составляет свыше 90% по сравнению с доступными коммерческими аналогами. В таблице 1 приведены результаты точности и полноты для некоторых связей для подмножества кода из ОС Android версии 4.4.2.

Табл. 1. Точность построения связей для кода из ОС Android

Тип связи	Точность	Полнота
IMPLICITLY_CALLS	94.70%	85.15%
CALLS	99.48%	88.67%
READS	96.62%	91.71%
WRITES	99.63%	88.91%

INCLUDES	100.00%	99.89%
INHERITS	99.52%	87.38%
OVERRIDES	98.75%	92.96%
Среднее значение	**98.29%**	**90.57%**

Необходимо отметить, что быстрое прототипирование инструмента стало возможно благодаря широкому использованию открытых компонентов проекта LLVM и GNU Binutils[7]. Наши модификации компилятора Clang были сосредоточены только на необходимых нам свойствах, а усилия для полной поддержки языков Си и Си++ тратить не требовалось. Метаданные сохранялась нами также в стандартном формате LLVM, что позволило считывать их имеющимися компонентами инфраструктуры. Наконец, анализ вызовов и анализ указателей могут также использовать имеющиеся компоненты.

5. Заключение

В статье описан разработанный инструмент анализа программ для их понимания, демонстрирующий достаточно точное построение связей между сущностями программы и метрик сущностей на промышленном исходном коде на языках Си и Си++. В дальнейшем предполагается добавить поддержку языка Java и поставить задачу поиска связей между частями программы, написанными на разных языках – например, поиск вызовов функций на языке Си из кода на языке Java через механизм JNI.

Список литературы

[1]. The LLVM Compiler Infrastructure. http://LLVM.org/
[2]. Clang compiler. http://clang.llvm.org
[3]. Инструмент Understand. https://scitools.com/
[4]. Инструмент Imagix4D. http://www.imagix.com/products/source-code-analysis.html
[5]. А. Аветисян, А. Белеванцев, А. Бородин, В. Несов. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды Института системного программирования РАН, том 21, 2011 г, стр. 23-38.
[6]. Иванников, В. П., Белеванцев, А. А., Бородин, А. Е., Игнатьев, В. Н., Журихин, Д. М., Аветисян, А.И, Леонов, М. И. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды Института системного программирования РАН, том 26, выпуск 1, 2014 г., стр. 231-250.

[7]. Пакет GNU Binutils. http://www.gnu.org/software/binutils/

Analyzing C/C++ Code Entities and Relations for Program Understanding*

A. Belevantsev <abel@ispras.ru>
E. Velesevich <evel@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004

Abstract. This paper describes the static analysis tool for finding program entities, their metrics, and relations between entities. Program entities are files/directories (physical structure) and classes/functions/methods/global variables (logical structure). Relations are connections between entities, such as calls, inheritance, aggregation, reading/writing, inclusion. The need for constructing such a tool arises from the program understanding problems. The basis of these problems' solutions should be automatic extraction of necessary data from program source code. The tool implementing this extraction should support all C/C++ constructs and should scale to millions lines of code to be applicable to real life applications.

In the paper we concentrate on the methods for developing such a tool for C/C++ languages based on open source components: LLVM/Clang compiler infrastructure, GNU Binutils linker and archiver. The final whole-program analysis works on the LLVM internal representation (bitcode) level, so it is necessary to save the additional source level data in the bitcode files that is usually lost during compilation. We have made more than 400 patches to the Clang compiler in order to support this additional data storing and also to support many varieties of C/C++ dialects. For the main analyzer, the central component is a kind of linker that unifies the collected data for the same entities used in different program components (like an application and a library it uses). The correct entities unification performed by the linker allows to trace component relations more precisely.

Finally, we briefly present the results of testing our tool on Android OS.

Keywords: program understanding; LLVM compiler; static analysis; code metrics.

DOI: 10.15514/ISPRAS-2015-27(2)-4

For citation: Belevantsev A.A., Velesevich E.A. Analyzing C/C++ Code Entities and Relations for Program Understanding. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 53-64 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-4.

References

[1]. The LLVM Compiler Infrastructure. http://LLVM.org/
[2]. Clang compiler. http://clang.llvm.org

[3]. The Understand tool. https://scitools.com/
[4]. The Imagix4D tool. http://www.imagix.com/products/source-code-analysis.html
[5]. A. Avetisyan, A. Belevantsev, A. Borodin, V. Nesov. Ispol'zovanie staticheskogo analiza dlya poiska uyazvimostej i kriticheskikh oshibok v iskhodnom kode program [The usage of static analysis for searching vulnerabilities and critical errors in source code]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 21, 2011. pp. 23-38 (in Russian)
[6]. V. Ivannikov, A. Belevantsev, A. Borodin, V. Ignatiev, D. Zhurikhin, A. Avetisyan, M. Leonov. Staticheskij analizator Svace dlya poiska defektov v iskhodnom kode program [Svace: static analyzer for detecting of defects in program source code]. Trudy ISP RAN [The Proceedings of ISP RAS], vol 26, issue 1, 2014, pp. 231-250 (in Russian)
[7]. The GNU Binutils package. http://www.gnu.org/software/binutils/

Об особенностях детерминированного воспроизведения при минимальном наборе устройств

В.Ю.Ефимов <real@ispras.ru>
К.А. Батузов <batuzovk@ispras.ru>
В.А.Падарян <vartan@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. Технология (детерминированного) воспроизведения вычислительного процесса в виртуальных вычислительных машинах используется для отладки, повышения отказоустойчивости, а также в различных исследованиях программного кода, в том числе, в области информационной безопасности при обратной инженерия вредоносных программ. В данной работе описывается реализация технологии воспроизведения для гостевых машин на базе Intel Architecture 32-bit в программном эмуляторе QEMU, предлагающая минимизацию перечня воспроизводимых устройств. Подробно рассмотрено устройство эмулятора QEMU и обоснованы технические приемы, использованные при реализации. Приводятся экспериментальные оценки ключевых характеристик: объем записываемого журнала недетерминированных событий и замедление.

Ключевые слова: виртуальная машина; детерминированное воспроизведение; эмулятор; QEMU.

DOI: 10.15514/ISPRAS-2015-27(2)-5

Для цитирования: Ефимов В.Ю., Батузов К.А., Падарян В.А. Об особенностях детерминированного воспроизведения при минимальном наборе устройств. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 65-92. DOI: 10.15514/ISPRAS-2015-27(2)-5.

1. Введение

В различных исследованиях машинного кода используется воспроизведение вычислительного процесса в виртуальной вычислительной машине (ВМ) (также известное как «детерминированное воспроизведение», «deterministic replay»). При воспроизведении гарантируется точное повторение последовательности состояний основных элементов ВМ и, как следствие, пути выполнения машинного кода. Имея такую возможность, можно выполнять

анализ уже на повторном выполнении, при этом факт анализа никак не повлияет на выполнение исследуемого кода: оно уже зафиксировано. Единственное оказываемое влияние — замедление от записи необходимой для воспроизведения информации при оригинальном выполнении, но для большинства известных реализаций технологии величина замедления не превышает десятков процентов. Такое замедление является пренебрежимо малым, особенно если сравнивать с замедлением от применения анализа непосредственно к оригинальному выполнению, которое составляет несколько порядков при трассировке или до нескольких секунд на инструкцию при отладке. Последнее делает невозможным успешное применение подобного анализа к коду чувствительному к задержкам (без технологии воспроизведения). Низкое замедление обусловлено малым объёмом данных, которые приходится записать, чтобы успешно воспроизвести первоначальный запуск. Большинство этих данных составляют взаимодействия ВМ с внешним миром.

Перечислим основные применения воспроизведения:

- реверсивная отладка позволяет проходить путь выполнения в обратном направлении: это даёт возможность искать место возникновения ошибки, начиная с места её проявления и двигаясь назад, а не использовать итеративный запуск программ с самого начала, как при классической отладке;
- устранение трудновоспроизводимых ошибок (в основном это ошибки, возникающие в параллельных программах: состояние гонок за данные, взаимоблокировки и т. д.);
- анализ компьютерных инцидентов (т. е. действий злоумышленника);
- анализ кода, чувствительного к замедлению (примерами такого кода могут служить сетевой сервер, сервер-приманка, защищенный от анализа вредоносный код, узел сети зараженных компьютеров: в последних трёх случаях модель нарушителя включает использование методов противодействия анализу, основанных на контроле замедления относительно внешнего мира).

Рассмотрим особенности воспроизведения вычислительного процесса, протекающего в ВМ на примере эмулятора QEMU [1] версии 2.1, работающего в режиме полносистемной эмуляции с использованием динамической двоичной трансляции. QEMU может эмулировать пользовательское окружение Linux или эмулировать систему с использованием аппаратной виртуализации, но в данной работе рассматривается исключительно вышеупомянутый режим. Фиксирование конкретной ВМ важно, так как наиболее интересные особенности воспроизведения скрыты именно в реализации ВМ, в то время как общая модель воспроизводимой системы включает в себя только высокоуровневые абстрактные правила. Воспроизвести вычислительный процесс — значит совершить ещё один, но таким образом, чтобы он был равен оригинальному

по некоторому заданному критерию (с заданными требованиями к точности). Точность воспроизведения определяется прикладными задачами. Например, иногда достаточно детерминированности выхода [2], что существенно ослабляет требования к методу. В данной же работе применяются следующие требования точности:

- совпадение последовательностей инструкций (коды операций, значения операндов, адреса инструкций в памяти);
- совпадение последовательностей состояний виртуального процессора и оперативного запоминающего устройства (ОЗУ) между инструкциями.

Заметим, что в состояние виртуального процессора также входят запросы прерывания (interrupt request — IRQ), а в состояние ОЗУ: таблица векторов прерываний, таблица виртуальной памяти и другие данные. Т.о., приведённые требования точности достаточны для выполнения анализа кода как на предмет наличия ошибок, так и на предмет уязвимостей и закладок [3].

При разработке технологии воспроизведения выделяют следующие понятия: детерминированная область, журнал недетерминированных событий, детерминированный таймер. Детерминированная область — это подсистема ВМ, поведение которой определяется взаимодействием с остальной частью ВМ. Т.е. если воспроизвести для детерминированной области все её взаимодействия с окружением, то последовательность её состояний также будет воспроизведена. Очевидно, что способ задания детерминированной области не единственен. В случае, когда ВМ — программный эмулятор, детерминированная область является частью данных и кода эмулятора, взаимодействующей с операционной системой (ОС), а также другими частями данных и кодом эмулятора. Если эмулятор работает параллельно, то следует учитывать как относительный порядок получения процессорного времени и взаимодействия между нитями внутри детерминированной области, так и обращения (относительный порядок и записываемые данные) внешних нитей к данным внутри области. Все взаимодействия с окружением (определяющие поведение детерминированной области) называются недетерминированными событиями. Они сохраняются в специальный журнал недетерминированных событий. Поскольку часть событий происходят асинхронно из внешней среды, их нужно повторять искусственно (т.к. внешняя среда недетерминированная и не гарантирует их повторение при каждом воспроизведении) и в правильные моменты относительно детерминированной области. Для последнего в детерминированной области выделяют детерминированный таймер, показывающий течение времени внутри области. В данной работе используется счётчик количества выполненных инструкций. В ряде работ [4, 5] предлагается использовать тройку: указатель инструкций (EIP), регистр-счетчик (ECX) и счётчик количества ветвлений (подразумевается IA-32).

Схематично процесс записи и воспроизведения показан на рис. 1. Он состоит из двух этапов: записи оригинального вычислительного процесса и его

воспроизведения. При записи все недетерминированные события сохраняются в журнал с соответствующими показаниями таймера. При воспроизведении все возникающие события скрываются от детерминированной области, но в соответствующие моменты искусственно подаются события из журнала.

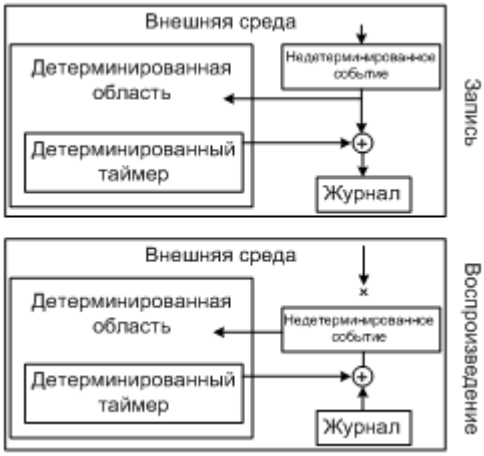

Рис. 1. Запись и воспроизведение.

Как уже отмечалось, детерминированная область не единственна, что создаёт возможность применения разных подходов [6]. В работе [7] был предложен и реализован альтернативный подход, отличающийся принципом выбора детерминированной области. А именно: в детерминированную область было включено большинство устройств ВМ. В предлагаемом подходе, наоборот, в область включены только ЦП и ОЗУ. Будем именовать подходы «Max ВМ» и «Min ВМ», исходя из принципов выбора детерминированной области. Сравнение подходов приведено в табл. 1.

Табл. 1. Сравнение подходов к обеспечению воспроизведения QEMU.

Характеристика	Max ВМ	Min ВМ
Перечень воспроизводимых состояний устройств	**Вся система**: процессор, ОЗУ, шины, контроллеры шин, таймеры, устройства ввода/вывода (жесткий диск, сетевой интерфейс, ...) и т. д. вплоть до взаимодействия с ресурсами основной ОС.	**Процессор**, **ОЗУ**, видео адаптер (как следствие воспроизведения выходных данных из процессора в его сторону).
Замедление	**Выше**. Требуется синхронизация работы всей системы. Задачи, выполняемые параллельно, нужно сериализовать.	**Ниже**. Наибольшая часть событий происходит из одной нити. Количество точек синхронизации нитей минимально.
Размер журнала	**Меньше**. Можно не писать некоторые события, исходя из их происхождения, если известно, что они детерминированы. Напр., образ жесткого диска.	**Больше**. Сложно отличить, какая информация пришла из детерминированных источников. Следовательно, нужно писать всё.
Наглядность событий	События непосредственно связаны с источником, напр., нажатие клавиши, движение мыши, приход сетевого пакета, срабатывание таймера и т. д. Наглядность **выше**.	У событий есть только адреса и разрозненные отрезки байт, которые уже прошли обработку в устройствах. Как следствие, наглядность **ниже**.
Сложность поддержки	Периферийные устройства в настоящее время меняются (добавляются) активно, что приводит к **более высокой** сложности поддержки.	Нужно только поддерживать изменения в интерфейсах взаимодействия с ближней периферией процессора, которые уже устоялись и почти не меняются, что приводит к **более низкой** сложности.

Характеристика	Max ВМ	Min ВМ
Сложность реализации	*QEMU версии до 0.13.0*. Сводится к перехвату необходимого минимума взаимодействий: их много. Сложность **средняя**. *QEMU версии от 1.0.1*. Взаимодействия с внешним миром разнесены в разные нити: дополнительно требуется обеспечить их синхронизацию. Сложность **выше**.	*Независимо от версии QEMU*. Необходимый минимум перехваливаемых событий невелик. Действия, выполняемые в других нитях, находятся за пределами области. Главная проблема — прямой доступ к памяти (DMA), выполняемый из другой нити (в случае версии от 1.0.1): требуется также организовать синхронизацию. Сложность **ниже**.
Расширяемость	Если добавляемое устройство взаимодействует с внешним миром (как чаще всего и есть), оно должно сохранять все свои взаимодействия с оным. Т. е. новое устройство — новый (часто уникальный) тип события. Расширяемость **сложнее**.	Процессор иногда находится через несколько интерфейсов до добавляемых устройств. Вплоть до того, что нет разницы, что именно находится с другого конца. Чаще всего добавление в систему нового устройства вообще не требует модификаций механизма воспроизведения. Расширяемость **легче**.

Объем недетерминированных событий в единицу времени не выходит за рамки пропускной способности жесткого диска компьютера. Высоконагруженные системы, обрабатывающие потоки сетевых данных, не развертываются на эмуляторах с двоичной трансляцией, а основными источниками недетерминизма в ВМ являются: загрузка данных с виртуального диска, пользовательский ввод и входящие сетевые пакеты. Таким образом, сброс на диск журнала перехваченных событий существенно не влияет на замедление ВМ. Помимо того, запись журнала можно буферизировать и выполнять в параллельной нити.

2. Эмулятор QEMU

В качестве ВМ для реализации воспроизведения был выбран QEMU по следующим причинам. Во-первых, необходима открытость исходного кода. Во-вторых, QEMU поддерживает наиболее распространённые архитектуры процессоров (IA-32, AMD64, ARM, MIPS, Power PC, др.). Это позволит позже расширить реализацию на другие архитектуры. В-третьих, QEMU переносим на уровне исходного кода: может быть скомпилирован для Linux и Windows.

Далее рассматриваются особенности работы QEMU, относящиеся к реализации воспроизведения. Эмулятор должен выполнять код, предназначенный для некоторой вычислительной машины (ВМ) (гостевой), опираясь на возможности основной машины (в которой выполняется сам эмулятор). Схема работы эмулятора и его связь с прообразом моделируемой ЭВМ изображены на рис. 2. Практика подразумевает использование упрощённой модели ЭВМ, с подробностью достаточной для выполнения машинного кода. В памяти эмулятора (т.е. памяти основной машины) создаются структуры, описывающие модели процессора, памяти и устройств гостевой машины, а также присутствует служебный код, определяющий взаимодействие между ними. Под выполнением гостевого кода понимается такая его обработка, при которой состояние гостевой машины (модели) изменяется таким образом, как изменилось бы состояние соответствующей реальной ВМ после выполнения этого кода. В QEMU этот процесс осуществляется с помощью динамической двоичной трансляции. Транслятор QEMU называется «Tiny Code Generator» (TCG). Двоичная трансляция происходит следующим образом. Сначала гостевой код транслируется в архитектурно независимое промежуточное представление, а затем оно компилируется в машинный код основной машины. Одной гостевой инструкции обычно соответствует несколько инструкций промежуточного представления. Также на уровне промежуточного представления в код вставляется служебная логика, необходимая для работы эмулятора. Служебная логика прозрачна для модели гостевой ВМ. Если эмуляция гостевой инструкции подразумевает сложную логику, то в эмуляторе реализуется специальная вспомогательная функция, а в промежуточное представление встраивается её вызов. Вспомогательные функции компилируются вместе с эмулятором. С точки зрения воспроизведения, среди вспомогательных функций важно отметить те, которые эмулируют инструкции времени, обращения к периферии и т.п. (рассматриваются ниже). Например, RDTSC из IA-32, возвращающая время. Трансляция кода выполнятся линейными участками (называемыми блоками трансляции) при первом обращении. Эмулятор буферизирует транслированный блок, чтобы не транслировать его каждый раз. Выполнение блока гостевого кода — это выполнение соответствующего транслированного кода.

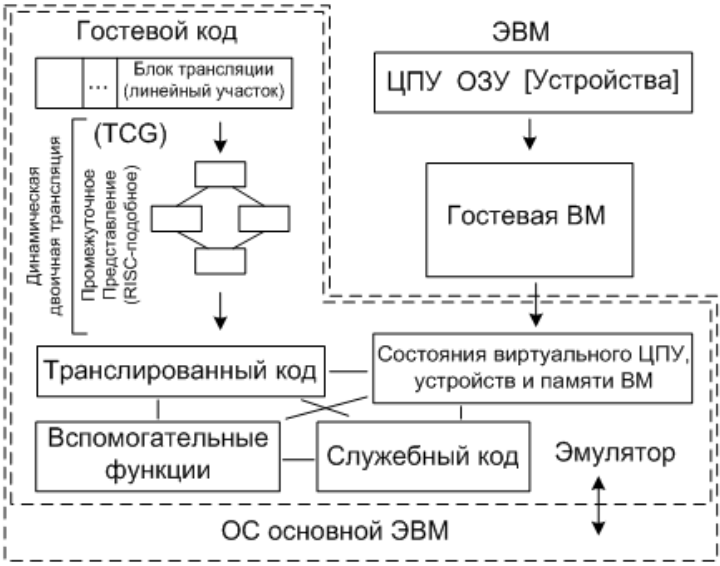

Рис. 2. Эмуляция ВМ в QEMU.

Выполнение в QEMU происходит в отдельной нити TCG в цикле эмуляции процессора. Итерация цикла изображена на рис. 3. Она состоит из выполнения гостевого кода, обработки исключений и запросов прерывания (IRQ), если последние имеются. Выполнение гостевого кода состоит из его получения (либо трансляцией, либо из буфера) и выполнения соответствующего транслированного кода. Если известно, что два блока трансляции всегда следуют один после другого (например, последняя гостевая инструкция в блоке есть переход по константному адресу или смещению), то они сцепляются. Сцепление осуществляется вставкой инструкции прямого перехода в транслированный код предыдущего блока на точку входа в транслированный код следующего блока.

Очевидно, что пока не получен IRQ, выполнение гостевого кода образует вложенный цикл в цикле эмуляции процессора (этот «горячий» путь выделен на рисунке толстой линией). Однако иногда эмулятору требуется по служебным нуждам приостановить эмуляцию. При этом для выхода из вложенного цикла используется специальный флаг — «CPU Exit ReQuest» (CPUERQ). Его опрос встречается дважды в итерации, т.к. получение транслированного кода может быть длительным (если происходит трансляция). Однако если из-за сцепления образовался длинный путь (или цикл) в транслированном коде, то CPUERQ не достаточно. В последнем случае применяется ещё один служебный флаг — «TCG Exit ReQuest» (TCGERQ). Проверка его установки встраивается в начало блока на уровне промежуточного представления при трансляции. Наконец, есть случаи, когда

необходимо прервать выполнение гостевого кода немедленно (не дожидаясь конца текущего блока). Например, для эмуляции исключений. Тогда выход происходит с помощью вызова стандартной функции siglongjmp.

Рассмотрим обработку IRQ, для выполнения которой эмулятор использует унифицированный механизм. В большинстве случаев IRQ соответствует моделируемому IRQ гостевой машины. Но также, например, может соответствовать запросу внешнего отладчика на остановку гостевого процессора. Если IRQ гостевое, то при его обработке, например, может быть изменено состояние контроллера прерывания, или могут быть совершены другие промежуточные действия. Т.е. происходит взаимодействие с устройствами. Признак наличия IRQ реализован в виде битового поля, где за каждым типом IRQ закреплён соответствующий бит-флаг. Кроме того, поведение обработчика прерывания является архитектурно зависимой частью итерации и, как следствие, его поведение может определяться другими переменными модели виртуального ЦПУ (или даже ВМ). В частности, гостевому прерыванию соответствует только один бит, а гостевой номер прерывания на этом этапе извлекается из контроллера прерывания.

Исключения, как и асинхронные прерывания, бывают гостевыми или служебными. Пример служебного исключения — запрос выхода из цикла эмуляции виртуального ЦПУ, например, для передачи управления другому процессору гостевой системы (многоядерные процессоры в QEMU моделируются несколькими виртуальными ЦПУ: по одному на ядро), для освобождения глобальной блокировки, для целей отладки гостя и т.д. Источниками гостевых исключений являются:

- синхронные исключения процессора (промах страницы, деление на ноль и т.д.);
- программные прерывания-ловушки (инструкция int N в архитектуре IA-32 и т.п.);
- в отдельных случаях обработка асинхронных прерываний гостевой ВМ завершается выдачей эмулятором служебного исключения, которое из-за особенностей его контекста следует рассматривать как гостевое.

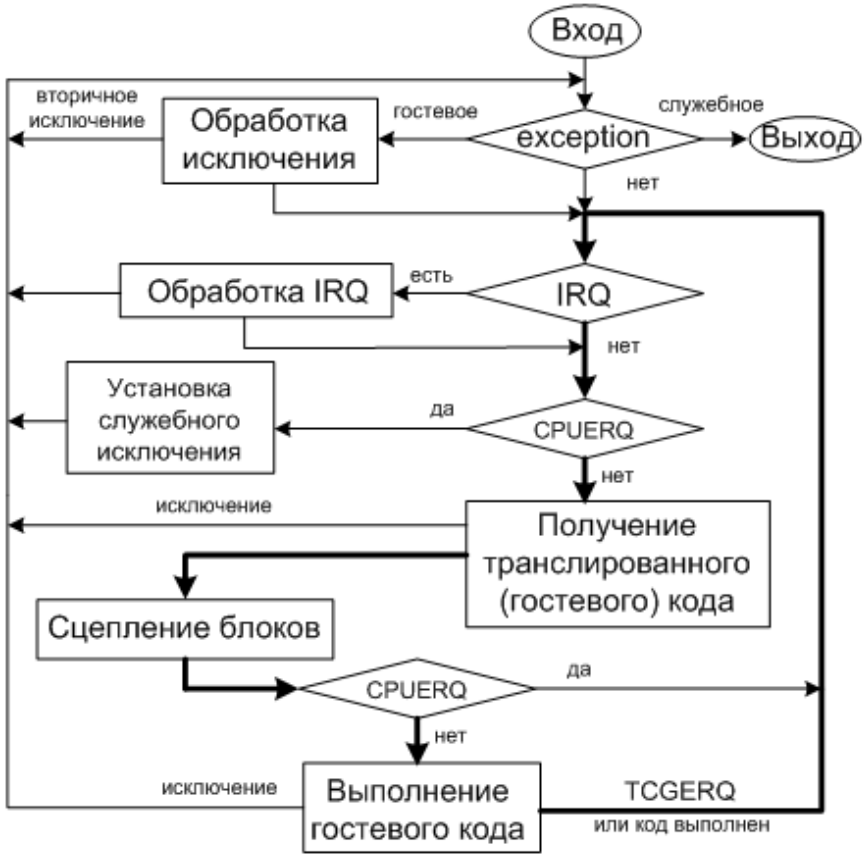

Рис. 3. Итерация цикла эмуляции виртуального ЦПУ.

Рассмотрим механизм обмена данными между виртуальным ЦПУ и устройствами. Он может происходить посредством отображения регистров устройств на физическое адресное пространство (Memory Mapped Input/Output — MMIO), с помощью DMA, а также через программируемый ввод/вывод (Port Mapped Input/Output — PMIO), предусмотренный в архитектуре IA-32. В случае MMIO процессор выполняет инструкцию чтения или запись из памяти по адресу, соответствующему устройству. В случае PMIO процессор выполняет инструкцию, указывая номер порта. В случае DMA процессор сначала задаёт команду устройству одним из вышеупомянутых способов, затем устройство самостоятельно копирует данные в память. Сначала рассмотрим работу MMIO и PMIO.

При моделировании физического адресного пространства используется интерфейс QEMU MMIO API (Application Programming Interface), ключевым

понятием которого является абстракция «регион памяти». Возможны следующие типы регионов памяти:

- ROM (Read Only Memory), RAM (Random Access Memory) — за регионом памяти закреплён буфер в памяти основной машины (например, ОЗУ, BIOS (Basic Input.Output System), видеобуфер, др.), все обращения к региону суть обращения к соответствующему буферу;
- IO (Input/Output) — для региона указаны функции-обработчики чтения и записи различных размеров, функции-обработчики вызываются при обращении к ним виртуального процессора: если выполняется запись в память, то функция принимает записываемое значение, если чтение, то она должна вернуть некоторое значение; кроме того, сам факт обращения может иметь значение для устройства, назначившего функцию-обработчик (в общем случае, логика работы может быть произвольной); примерами являются MMIO, PMIO, отслеживание самомодифицирующегося кода и др.;
- alias (псевдоним) — регион, перенаправляющий все обращения к себе на другой регион со смещением;
- контейнер — регион, содержащий в себе другие регионы (подрегионы) со смещением.

Таким образом, адресное пространство гостевой ВМ моделируется деревом регионов, где рёбро соответствует отношению «контейнер-подрегион». MMIO организуется с помощью регионов типа IO. Для своего регистра (или группы регистров) устройство назначает регион и функции-обработчики, и уже они определяют, как связаны соответствующие участки адресного пространства с состоянием и поведением устройства. PMIO реализовано аналогично MMIO: создано отдельное дерево регионов. Адресами в дереве являются номера портов. Взаимодействие гостевого кода с устройствами через MMIO и PMIO изображено на рис. 4.

Для гостевой инструкции, обращающейся к памяти или порту, генерируется вызов вспомогательной функции. Вспомогательная функция находит регион, соответствующий адресу (номеру порта), и вызывает для него унифицированный обработчик, который в свою очередь передаёт управление обработчику региона.

Важно отметить следующие особенности работы обработчиков:

- специальный обработчик может обратиться к ОЗУ гостя (например, контроллер прерываний APIC (Advanced Programmable Interrupt Controller) хранит в ОЗУ часть своего состояния и синхронизирует его при каждом обращении к своему регистру через MMIO);
- унифицированный обработчик может рекуррентно вызвать сам себя для фрагментации доступа (если произошло обращение длины

большей чем та, для которой устройство назначило обработчик, то обращение будет разбито на серию обращений подходящей длины).

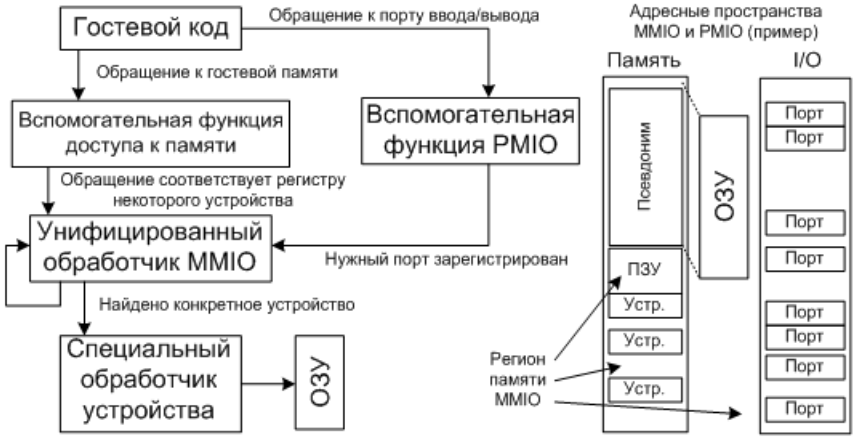

Рис. 4. Взаимодействие виртуального процессора с устройствами через MMIO и PMIO.

Таким образом, был освещен процесс выполнения гостевого кода и его взаимодействие с периферией. Однако для выполнения гостевого кода эмулятору также необходимо эмулировать работу устройств, которые, вообще говоря, работают независимо от ЦПУ. Наконец, эмулятор должен обеспечивать графический интерфейс пользователя и мониторы команд управления. На рис. 5 приведена схема организации QEMU в части распределения работы между нитями, показаны компоненты эмулируемой ВМ (выделены серой заливкой), служебные механизмы и данные. Обращения к ОЗУ и APIC происходят во всех нитях, поэтому на рисунке они вынесены за их пределы.

Нить TCG обеспечивает рассмотренное выше выполнение гостевого кода. Нить ввода-вывода выполняет в основном служебные задачи. А именно, она содержит главный цикл, который обрабатывает команды от пользователя как через графический интерфейс, так и через мониторы управления; обеспечивает интеграцию с ОС и т.д. Кроме того, главный цикл эмулирует работу таймеров: устройство может назначить обратный вызов на определённый момент времени относительно гостя. С точки зрения воспроизведения, важным в работе нити ввода-вывода является следующее:

- нить производит предварительную инициализацию ВМ, включая запись в ОЗУ;
- обратные вызовы таймеров, как правило, изменяют состояние IRQ.

В QEMU для параллельного обращения к файлам-образам ПЗУ ВМ (образа ROM, образа жестких дисков и др.) используются нити-рабочие. Это позволяет не блокировать другие нити на время выполнения обращения. Таким способом эмулируется DMA: нить-рабочий читает (пишет) данные из файла-образа в память основной машины, соответствующую памяти гостевой машины, в которую назначена транзакция DMA.

Описанные потоки данных обозначены на рис. 5 линиями со стрелками, соответствующими возможным направлениям копирования данных.

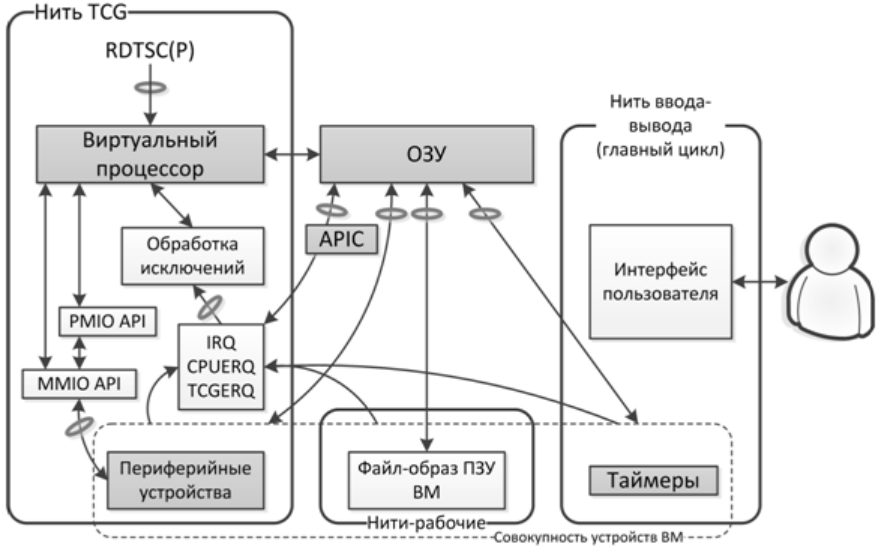

Рис. 5. Схема организации нитей выполнения в QEMU. Места перехвата и последующего вброса недетерминированных событий выделены овалами

3. Запись и воспроизведение

Перейдём к рассмотрению реализации предлагаемого подхода к записи и воспроизведению. Напомним, что одной из частей реализации воспроизведения является детерминированный таймер. В качестве детерминированного таймера используются счётчик выполненных виртуальным ЦПУ инструкций и счётчик недетерминированных событий.

Счётчик инструкций необходим для привязки ко времени IRQ и обращений к ОЗУ из нитей-рабочих и нити ввода-вывода. Подсчёт инструкций реализован на уровне промежуточного представления. В начало каждого блока трансляции вставляется инструкция, увеличивающая счётчик на размер блока в гостевых инструкциях. Кроме того, дополнительно учитываются случаи преждевременного выхода из блока через вызов функции siglongjmp (когда

количество действительно выполненных инструкций меньше, чем всего инструкций в блоке). К таким случаям относятся:

- инструкция, вызвавшая исключение или программное прерывание;
- обработчик MMIO/PMIO, произведший немедленный выход из цикла выполнения гостевого кода по какой-либо причине.

Когда инструкция вызывает исключение, то считается, что она не выполнена. Т.е. предполагается, что гость устранит причину исключения, выполнив другой код, и в следующий раз выполнение этой инструкции не вызовет исключение. Инструкции программного прерывания, напротив, следует считать выполненными. Подобное также справедливо для некоторых особенных инструкций, прерывающих поток управления. Для IA-32 это: `HLT` (останавливающая процессор до следующего прерывания), `INT` и др. Компонента TCG, генерирующая из гостевого кода промежуточное представление, обеспечивает, чтобы подобные инструкции были последними в блоке, как и инструкции перехода. Таким образом, достаточно вычесть из счётчика единицу (или не вычитать: в зависимости от типа инструкции). Кроме того, обращения к MMIO/PMIO могут прерывать выполнение блока в произвольном месте. Это нужно в случае, если после выполнения обращения выполнение последующего кода будет некорректным. Например, через MMIO API осуществляется контроль целостности гостевого кода для поддержки самомодифицирующегося кода. Если код изменил страницу, в которой находится сам, то нельзя считать, что последующие инструкции транслированного кода соответствуют текущему содержимому памяти (в QEMU применяется постраничная гранулярность).В таких случаях используется процедура восстановления регистра счётчика команд, изображенная на рис. 6. Блоки трансляции обозначены на рисунке «БТ», а вспомогательные функции – «helper».

Проблема преждевременного выхода из блока обусловлена оптимизациями, реализованными в QEMU, а именно ленивым вычислением флагов и регистров гостевого процессора. Т.е. они вычисляются только тогда, когда их значения необходимы. В частности, PC (program counter) не устанавливается после выполнения каждой инструкции на адрес следующей. Это не нужно, т.к. инструкции разбиты на блоки и известно, какая будет следующей в пределах блока трансляции. Необходимо только знать, какой блок следует выполнять следующим если они не сцеплены. Т.е. PC имеет корректное значение только между блоками трансляции или при явном чтении его значения. Если произошел преждевременный выход из блока, то PC указывает на его начало, а не на инструкцию, вызвавшую выход. В то же время нужно продолжить выполнение с этой инструкции. Следовательно, нужно узнать её гостевой адрес.

Восстановление правильного гостевого PC происходит следующим образом. Известно, что каждой инструкции гостевого кода соответствует диапазон инструкций кода основной машины. Преждевременный выход происходит из

вспомогательной функции (helper), адрес возврата (в памяти основной машины) из которой известен (TCG передаёт одним из параметров вспомогательной функции адрес возврата из неё). Т. о. можно определить, к какому диапазону он относится. При определении PC блок транслируется повторно, но сопровождается информацией об адресах, порядковых номерах в блоке и отображении гостевых инструкций на диапазоны памяти основной машины. PC определяется по этой информации. Поскольку в ходе восстановления PC известно количество реально выполненных инструкций, нетрудно скорректировать счетчик.

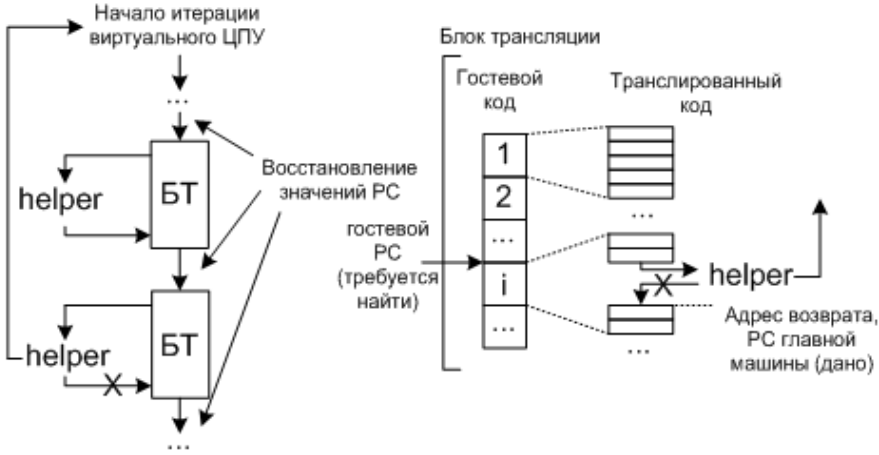

Рис. 6. Восстановление регистра счётчика команд (PC).

Восстановление PC происходит непосредственно перед выходом из гостевого кода. Но после восстановления не обязательно произойдёт выход (например, в случае APIC). Т.е. будет выполнено восстановление, но блок трансляции всё равно будет выполнен до конца. Очевидно, что в этом случае корректировать счётчик инструкций не нужно. Для обработки подобных случаев в состоянии виртуального ЦПУ было добавлено специальное поле, хранящее результат последнего восстановления. Он применяется, если только действительно произошел преждевременный выход. Однако, если преждевременный выход после восстановления всё-таки произошел, но из другой инструкции, которая не предусматривает восстановление PC (например, исключение или программное прерывание, которые явно знают адрес следующей инструкции посредством обращения к TCG), то содержимое этого специального поля будет некорректно применено. Следовательно, нужно обнулять это специальное поле после выполнения специального обработчика (если он вернул управление в универсальный обработчик, то он не совершил преждевременного выхода из блока, а значит, результат восстановления точно не пригодится).

В итоге счётчик инструкций имеет корректное значение между блоками. Этого достаточно, так как:

- IRQ обрабатываются между блоками трансляции;
- параллельные обращения к ОЗУ из других нитей дополнительно блокируются на время выполнения гостевого кода, и, следовательно, выполняются между блоками трансляции.

Однако внутри блока счётчик инструкций имеет некорректное значение. В то же время, доставка некоторых событий (в частности MMIO и PMIO) требует привязки к конкретной инструкции. Для решения этой проблемы используется счётчик событий. Каждое обращение к MMIO вызывает увеличение счётчика, даже если событие детерминировано (например, запись из виртуального процессора в устройство) и не требует записи в журнал. Это порождает в журнале *виртуальные* события, требующие особой обработки при воспроизведении. Подробнее об обработке MMIO будет указано ниже.

Обратим внимание, что в ранних версиях использовался подсчёт инструкций поштучно, т.е. перед каждой инструкцией вставлялся инкремент счётчика. Подобное решение вносило замедление 4 — 11% (в зависимости от характера выполняемого кода). Текущий подход вносит замедление 2 — 6%, соответственно.

Далее рассматриваются места и способы перехвата недетерминированных событий. На рис. 5 эти места обозначены овалами вокруг стрелок. Очевидно, что в соответствии с подходом в детерминированную область входят виртуальный процессор и ОЗУ, но не APIC.

3.1. Инструкции RDTSC и RDTSCP

Инструкция `RDTSC` (и её вариант `RDTSCP`) возвращает гостевой показатель времени виртуального ЦПУ. Возвращаемое значение зависит от времени основной машины, поэтому является недетерминированным. Хотя есть режим работы QEMU «icount», при котором течение виртуального времени выравнивается по числу выполненных инструкций; но даже в этом режиме время корректируется по реальному времени в случае простоя процессора, когда инструкции не выполняются (например, если процессор ожидает прерывание). Инструкция `RDTSC` реализована вспомогательной функцией. При записи возвращаемое значение инкапсулируется в событие и записывается в журнал, а при воспроизведении берётся из него и подменяет текущее.

3.2. MMIO, PMIO и синхронный доступ к ОЗУ

Чтение данных виртуальным процессором из устройств через MMIO и PMIO эмулируется вызовом функции, возвращаемое значение которой — прочитанные данные. Эти данные являются недетерминированными и сохраняются в журнал при записи. При воспроизведении ими подменяются текущие данные. Очевидно, что данные, записываемые процессором в

устройство, во-первых, являются детерминированными, во-вторых, не влияют на поведение детерминированной области: их записывать не нужно. При отладке, тем не менее, они записывались и использовались для проверки корректности воспроизведения.

Основной проблемой является недетерминированность специального обработчика и потенциальная неограниченность его возможных действий. Известно следующее поведение (важное для воспроизведения):

- рекуррентное обращение к MMIO/PMIO,
- обращение к ОЗУ,
- модификация исполняемого в данный момент гостевого кода (особый случай обращения к ОЗУ).

Как параметры, так и сам факт выполнения вышеперечисленных действий являются недетерминированными.

В случае рекуррентного обращения важными являются только данные, возвращенные корневым вызовом. Промежуточные данные возвращаются специальному обработчику, а не виртуальному ЦПУ, поэтому их записывать не нужно. Для их отсева отслеживается откуда обработчик был вызван.

Обращение к ОЗУ из устройства является недетерминированным событием и должно быть записано. Очевидно, что нужно воспроизвести данные, которые записываются в память. Кроме того, факт обращения зависит от недетерминированного состояния устройства. Т.е. при воспроизведении нужно искусственно сделать вызов, если он не был сделан устройством.

Особым случаем обращения к ОЗУ является модификация текущего кода. Единственным известным устройством, выполняющим это, является APIC. Последовательность действий его специального обработчика следующая:

- восстановление PC (чтобы потом продолжить выполнение с того же адреса);
- изменение гостевого кода;
- перетрансляция изменённого кода;
- выход из текущего блока, т.к. он больше не соответствует гостевому коду в этом месте ОЗУ.

Восстановление PC нужно сделать именно до модификации кода, т.к. алгоритм восстановления (среди прочего) требует неизменность кода. Это справедливо и при воспроизведении. Изменение гостевого кода здесь — запись в ОЗУ, которая обрабатывается так же, как и обычное обращение к ОЗУ. В частности, APIC изменяет код инструкции, вызвавшей обращение к нему: вместо неё вставляется вызов функции в гостевой ОЗУ с параметром равным тому, который был передан специальному обработчику APIC. Вышеописанные действия нужно воспроизводить в таком же порядке относительно друг друга. При этом для воспроизведения восстановления PC и перетрансляции с выходом из блока потребовалось ввести специальный тип событий (изменение гостевого кода описывается событиями записи в ОЗУ).

Для обработки событий доступа к ОЗУ, MMIO и PMIO нужно задать:

- порядок среди соответствующих инструкций (внутри блока трансляции),
- порядок в дереве вызовов обработчика каждой конкретной инструкции.

При этом значение счетчика выполненных инструкций внутри блока некорректно, и не может быть использовано. Для решения данной проблемы реализовано следующее:

- подсчёт всех инструкций MMIO/PMIO как событий (вторая составляющая детерминированного таймера, описанная выше), для определения нужной (по порядку) инструкции в блоке трансляции;
- сохранение стека вызовов (путь в дереве вызова обработчика инструкции MMIO/PMIO) для соответствующих событий, чтобы их вставить искусственно, если они были пропущены специальным обработчиком.

Ввиду недетерминированности устройств, при воспроизведении они могут находиться в состоянии, при котором обращение к ним в неподходящий момент или с неудачными данными может привести к отказу эмулятора. Но, между тем, есть и устройства, обращаться к которым можно/нужно. Например, для удобства пользователя желательно воспроизводить состояние графического буфера видеоадаптера. Решение проблемы — при воспроизведении выборочно разрешать обращения к конкретным устройствам.

Помимо того, возможности MMIO API шире, чем эмуляция регистров устройств и портов ввода/вывода. В частности, эмулятор использует его для поддержки самомодифицирующегося кода, простановки точек останова по обращению к данным (watchpoint) и др. Т.е. в адресное пространство добавляются регионы памяти со специальными функциями-обработчиками. Наличие и поведение таких регионов MMIO, во-первых, не влияет на состояние гостевой ВМ, во-вторых, недетерминировано. Т.о. эти события не нужно учитывать, поскольку это может сбить счетчик событий внутри блока, что приведёт к доставке событий MMIO в неправильные обработчики. Для решения последних проблем структуры данных, описывающие регионы памяти, были снабжены полями, регулирующими обработку перехваченных обращений к MMIO API:

- учитывать ли как событие;
- (если учитывать) запрещать ли выполнение специального обработчика чтения/записи при воспроизведении.

Кроме обработчиков MMIO, синхронно к ОЗУ может обращаться APIC в момент получения номера прерывания при обработке IRQ (в теле цикла эмуляции виртуального процессора). Присутствие данного события является

недетерминированным, т.е. не каждое получение номера прерывания сопровождается обращением к ОЗУ. Ввиду аналогии с обращением к ОЗУ из MMIO, было принято решение использовать тот же механизм обработки: место вызова функции, возвращающей номер прерывания, окружено кодом, эмулирующим для последующих вызовов то, что они находятся в дереве вызовов MMIO. Т.о. требуемое обращение к ОЗУ будет записано как обращение к ОЗУ из обработчика MMIO. При воспроизведении такие события выравниваются по счётчику событий как и обычный доступ к ОЗУ из MMIO. Сам же факт получения номера прерывания воспроизводится как запрос прерывания.

3.3. Запросы прерывания

IRQ могут выставляться устройствами в произвольные моменты времени из любой нити. Однако для воспроизведения работы процессора важен момент обработки IRQ относительно инструкций и его недетерминированные данные. Т.о. при записи события сохраняются:

- номер инструкции,
- битовое поле, кодирующее IRQ, и дополнительные архитектурно-зависимые параметры.

При воспроизведении все возникающие IRQ игнорируются, а записанные IRQ встраиваются искусственно. Особенность заключается в том, что разбиение гостевого кода на блоки трансляции является недетерминированным: например, точка остановки, заданная пользователем в отладчике, разобьет блок трансляции на две части. Т.е. нельзя гарантировать, что при записи и каждом воспроизведении место вставки IRQ будет само собой попадать на разрыв между блоками. Поэтому задача вставки IRQ между конкретными инструкциями при воспроизведении требует внесения следующих трех изменений в алгоритм обработки гостевого кода.

- Сцепления блоков не происходит. Сцепление нужно для того, чтобы не возвращать управление эмулятору лишний раз. При записи, когда возникает IRQ, асинхронно (из другой нити) устанавливается флаг TCGERQ, что приводит к возврату управления перед выполнением очередного блока. При воспроизведении этот способ не приемлем, т.к. нельзя гарантировать точную доставку TCGERQ. Отключение сцепления позволяет получать управление после каждого блока и вставлять IRQ. Это замедляет воспроизведение.
- Длина (в гостевых инструкциях) блока трансляции ограничивается таким образом, чтобы его последняя инструкция была не позже, чем следующий записанный IRQ.
- Код повторно транслируется, если длина блока достаточно велика, и его последняя инструкция будет выполняться позже следующего IRQ. Этот случай возможен, т. к. при трансляции блока количество

инструкций до ближайшего IRQ могло быть больше, чем при последующих его выполнениях. Фактически происходит уничтожение блока и генерация нового по приведенному выше (второму) правилу.

Описанные изменения также необходимы и для воспроизведения событий асинхронного обращения к ОЗУ. При реализации этих изменений события асинхронного доступа к ОЗУ становятся частью потока событий IRQ.

3.4. Асинхронный доступ к ОЗУ

Под асинхронным доступом к ОЗУ здесь понимается любой доступ к ОЗУ, который происходит не из потока управления нити TCG. Примерами асинхронного доступа к ОЗУ являются:

- инициализация памяти из нити ввода-вывода при начальной загрузке ВМ;
- изменение состояния APIC (его часть, которая хранится в ОЗУ) при изменении статуса IRQ устройством по таймеру (из нити ввода-вывода) или другой причине;
- выполнение DMA нитью-рабочим и др.

Основная проблема асинхронного доступа к ОЗУ — это состояние гонок между нитью TCG и другими нитями, влияющее на путь выполнения гостевого кода. Решение о выделении процессорного времени нити принимает ОС, а эмулятор может на это повлиять посредством примитивов синхронизации. В текущей реализации записи ОЗУ защищена двоичным семафором. Он принадлежит нити TCG, пока выполняется гостевой код. Между блоками семафор освобождается и может быть захвачен другой нитью. При записи обращение сохраняется с привязкой к счётчику выполненных инструкций (при выполнении нескольких обращений порядок между ними задаётся по счётчику событий). Напомним, что между блоками трансляции счетчик инструкций содержит корректное значение.

При воспроизведении обращение к ОЗУ из других нитей не выполняется. Все записанные обращения встраиваются искусственно и синхронно (из нити TCG). Особенности вставки этих событий между конкретными инструкциями такие же, как и при доставке IRQ, и описаны выше.

Особый случай асинхронного доступа к ОЗУ представляет эмуляции DMA из нитей-рабочих. Основная проблема в том, что копирование происходит между жестким диском и ОЗУ. Обращение к диску сравнительно длительная операция, и захват семафора на время её выполнения вызвал бы неоправданное замедление при записи. Поэтому соответствующий код был модифицирован так, чтобы при записи сначала параллельно с нитью TCG производилось копирование данных между диском и специально выделенным

буфером в ОЗУ, а затем захватывался семафор, и производилось копирование между буфером и первоначальным местом в ОЗУ.

3.5. Запись журнала

Поток событий обладает следующими свойствами:

- количество байт журнала производимых за единицу времени может отличаться на несколько порядков в разные моменты времени и может превысить пропускную способность распространённых жестких дисков;
- усреднённое по времени количество байт журнала производимых за единицу времени не превышает пропускную способность жесткого диска (за исключением случая, когда ВМ активно работает с виртуальным диском, чей образ располагается на диске основной машины);
- количество событий на единицу времени может отличаться на порядки;
- основным источником событий является нить TCG, поэтому низкая скорость записи журнала вызовет замедление виртуального ЦПУ.

С учётом этих особенностей запись журнала была организована по схеме, представленной на рис. 7.

Во-первых, события буферизуются, что позволяет сгладить замедление при большом количестве событий в единицу времени, однако не решает проблему замедления от копирования данных на диск при заполнении буфера. Но количество системных вызовов уменьшается на порядки.

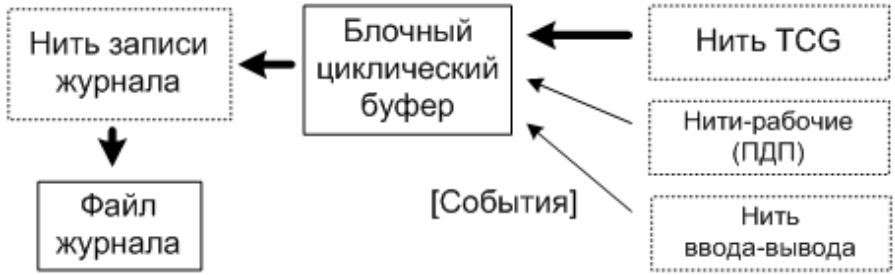

Рис. 7. Схема записи журнала.

Во-вторых, для записи журнала на диск создаётся специальная нить, работающая по схеме поставщик-потребитель. В памяти присутствует несколько буферов. Они упорядочены циклически: когда один заполняется, для записи выбирается следующий. Когда нить записи журнала замечает очередной заполненный буфер, она записывает его на диск. Такая схема позволяет исключить замедление записи журнала от относительно низкой пропускной способности диска. Т.е. при условии, что размер буфера

достаточно велик, замедление гостевой системы от записи будет определяться лишь пропускной способностью ОЗУ. Практически, всегда можно назначить буфер достаточного размера: для большинства гостевых ОС достаточно 100МБ.

4. Результаты

Для оценки ключевых свойств реализаций детерминированного воспроизведения были проведены численные эксперименты. Конфигурация основной машины следующая: процессор Intel Core i7-3770, 8ГБ ОЗУ, ОС Linux Ubuntu 14.04 64-х битная версия. Для сравнения реализаций оценивались:

- загрузка ОС Microsoft Windows XP, Service Pack 3;
- загрузка ОС Microsoft Windows 7, Service Pack 1;
- результаты бенчмарка nbench [8] на базе ОС GNU/Linux, версия ядра 3.12;
- результаты бенчмарка iperf [9] и утилиты ping на базе ОС Microsoft Windows XP, Service Pack 3.

В табл. 2 представлены результаты сравнения свойств реализаций подходов на загрузке гостевых ОС. Напомним, что основными характеристиками реализации, является относительное замедление записи и размер журнала. С точки зрения практического применения, реализация не должна сильно замедлять вычислительный процесс и при воспроизведении: хотя замедление при воспроизведении не скажется на точности, оно может сказаться на времени выполнения последующего анализа. Измерение времени загрузки Windows производилось 5 раз, в таблице приведены средние арифметические значения.

Табл. 2. Экспериментальное сравнение свойств реализаций подходов на загрузке гостевых ОС.

Тест	QEMU 2.1.0	Max BM	Min BM
ОС Windows XP, SP 3			
Время загрузки	23,6 сек	37,0 сек (+57%)	33,6 сек (+42%)
Размер журнала	-	11 МБ	476 МБ
Время воспроизведения	-	~96 сек замедление ~4 раза	~88 сек замедление ~3,76 раза
ОС Windows 7, SP 1			
Время загрузки	105,6 сек	189,3 сек (+79%)	150,0 сек (+42%)
Размер журнала	-	155 МБ	1036 МБ
Время воспроизведения	-	~17 мин замедление ~9,66 раза	~15 мин замедление ~ 8,52 раза

В табл. 3 представлена оценка замедления различных подходов с использованием утилиты nbench. Запуск nbench производился 10 раз и усреднялся. Приведённые значения индексов есть отношение количества выполненных операций в единицу времени к некоторому эталонному значению, соответствующему скорости работы определённого реального процессора. Т.е. чем больше индекс, тем быстрее работает ВМ. Поскольку вырабатываемый бенчмарком результат – индекс производительности, его измерение осмыслено только на первом проходе, когда пишется журнал событий, а измерение течении времени выполнятся по реальным часам.

Табл. 3. Экспериментальное сравнение свойств реализаций подходов с использованием nbench.

Тест	QEMU 2.1.0	Max ВМ	Min ВМ
nbench на базе ОС Linux, ядро 3.12			
Memory Index	3,553	1,003 (-72%)	3,208 (-9%)
Integer Index	3,881	1,040 (-73%)	2,289 (-41%)
Float-point Index	1,735	1,441 (-17%)	1,635 (-6%)
Размер журнала	-	26 МБ	28 МБ

Результаты, приведенные в таблицах 2 и 3, позволяют сделать следующие выводы. Во-первых, наблюдается значительная разница размера журнала для загрузки ОС. Большой размер журнала подхода «Min ВМ» обусловлен записью всех данных, считанных с диска. Напротив, реализация подхода «Max ВМ» считает образ диска детерминированным и не записывает считанные из него данные. С другой стороны, для воспроизведения по предлагаемому подходу достаточно только журнала, в то время как для воспроизведения по подходу «Max ВМ» требуется неизменный образ диска. Это особенно важно в случае распределённой работы с журналом, т.к. обычно размер образа диска значительно больше журнала (даже для подхода «Min ВМ»). Для случая записи приложения (nbench) размеры журналов почти не отличается. Имеющуюся разницу теоретически можно объяснить записью в журнал загрузки приложения с диска (в случае подхода «Min ВМ»).

Во-вторых, по результатам nbench можно сказать, что замедление зависит от характера выполняемых инструкций. Сравнительно низкая разница замедления инструкций математического сопроцессора (float-point index) обусловлена тем, что в QEMU инструкции сопроцессора эмулируются вспомогательными функциями. Последнее приводит к тому, что на одну инструкцию сопроцессора приходится значительно больше инструкций процессора основной машины, чем на одну инструкцию арифметико-логического устройства процессора. Т.е. замедление непосредственно от записи журнала при активном использовании сопроцессора составляет меньшую долю от общего времени.

В-третьих, в целом предлагаемый подход вносит меньшее замедление. Замедление в подходе «Max BM» может быть обусловлено сериализацией всех нитей QEMU для обеспечения корректной записи.

В-четвёртых, время воспроизведения в реализации предлагаемого подхода, как и в другом подходе, значительно больше времени записи. В подходе «Min BM» основными причинами этого являются:

- отсутствие сцепления блоков и их дополнительная перетрансляция (ради своевременной доставки IRQ);
- при чтении журнала не используются те же оптимизации, как при записи (распараллеливание, буферизация).

На рис. 8 приведён график роста журнала для ОС Windows XP SP3. Ниже приводятся действия, которые производились с целью выявления влияния на них записи и выявления их влияния на скорость роста журнала. Номерами на рис. 8 обозначены соответствующие этапы.

1. *Появление рабочего стола*. Этот момент соответствует окончанию измерения времени загрузки из табл. 2. ОС дано некоторое время на завершение загрузки. Во время загрузки и некоторое время после появления интерфейса пользователя наблюдается хаотичность скорости роста журнала. Это связано с загрузкой драйверов и опросом соответствующих устройств, загрузкой графических элементов пользовательского интерфейса, автозапуском приложений и др. В самом начале загрузки скорость достигает ~100 МБайт/сек, что можно объяснить загрузкой основных компонентов ОС.
2. *Активные движения указателем мыши и ввод текста в текстовой редактор*. Очевидно, что действия пользователя вносят вклад в журнал, т.к. являются недетерминированными. Кроме того, в течение данного этапа был запущен текстовой редактор, т.е. происходило чтение данных с диска.
3. *Работа без внешних воздействий.* Скорость роста журнала около 60 Кбайт/сек.
4. *Запуск утилиты iperf* [9] в режиме клиента для тестирования пропускной способности соединения с основной машиной через виртуальное сетевое окружение, реализуемое QEMU (SLIRP). На основной машине iperf была запущена в режиме сервера. Тестирование выполнялось с использованием протокола TCP. Достигнутая пропускная способность соединения: 41,3 Мбит/сек (при аналогичных измерениях, но без записи журнала — 54,2 Мбит/сек: замедление ~24%). Измерение проводилось 5 раз, приведены средние арифметические значения. Обмен информацией по сети также вызвал рост журнала с соответствующей скоростью.
5. *Запущено штатное завершение работы ОС*.

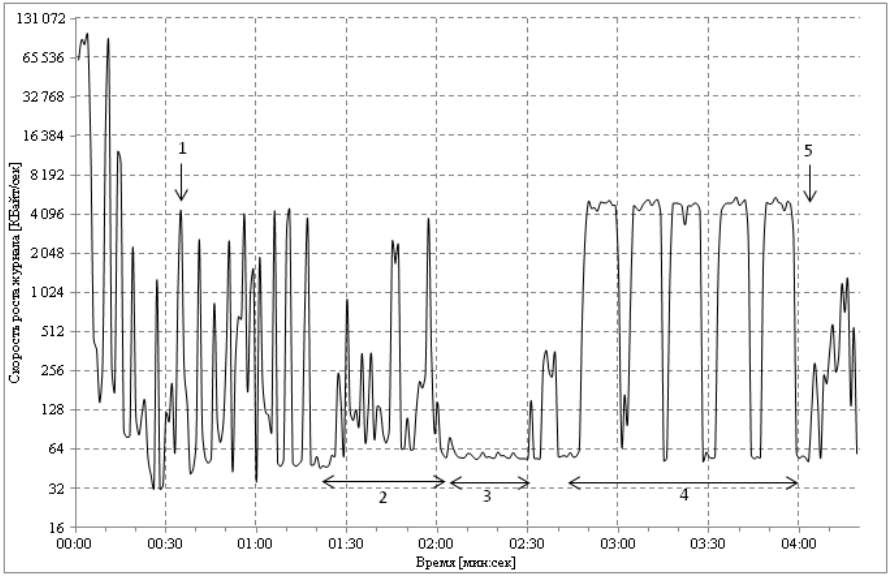

Рис. 8. Скорость роста журнала для ОС Windows XP SP3

Отдельно измерялось влияние записи на время отклика по сети с использованием утилиты ping. Для измерения было запущено две ВМ, сопряженные в одну сеть, и время отклика измерялось между ними (вторая ВМ нужна, т.к. SLIRP не поддерживает протокол ICMP). Запись производилась только на той ВМ, где была запущена ping. Как при записи, так и без неё первые 2-4 ответа приходили с задержкой менее 5мс, а последующие пакеты приходили с задержкой не более 1 мс. Т.о. запись не вызывает заметного увеличения времени отклика. Сравнительно высокая задержка первых пакетов может быть объяснена трансляцией соответствующего гостевого кода.

Все записанные действия были воспроизведены, отклонений выявлено не было. Время воспроизведения вычислительного процесса, соответствующего рис. 8, составило 8 мин 56 сек (+106%).

5. Заключение и дальнейшая работа

Таким образом, существуют два подхода к реализации воспроизведения ВМ QEMU. В работе был исследован подход «Min ВМ» к воспроизведению гостевой среды QEMU, основывающийся на минимизации детерминированной области. Было проведено теоретическое сравнение данного подхода с альтернативным подходом «Max ВМ», основывающимся на максимизации детерминированной области. Реализация предлагаемого подхода поддерживает воспроизведение однопроцессорной ВМ на базе IA-32.

Она была протестирована на популярных ОС: Microsoft Windows XP, Microsoft Windows 7 и GNU/Linux 3.12.

Свойства реализации предлагаемого подхода соответствуют практическим требованиям, не уступают свойствам альтернативного подхода, а в части замедления (как при записи журнала, так и при воспроизведении) показываются лучшие результаты. Вносимое предложенным методом замедление составляет от 6 до 42% в зависимости от вида гостевого кода. Снижение пропускной способности сетевого взаимодействия гостевой системы составляет порядка 24% при несущественном увеличении времени отклика. Во время воспроизведения вычислительный процесс замедляется не более чем на десятичный порядок. Скорость роста журнала непосредственно зависит от интенсивности обмена информацией с внешним миром (включая трафик с виртуальным диском). При работе без взаимодействий с внешним миром скорость роста журнала имеет порядок десятков КБайт в секунду, что близко к росту журнала в подходе «Max BM».

В текущей реализации используется один счётчик инструкций. В итоге корректная работа ВМ с несколькими процессорам (многоядерный ЦПУ) не гарантируется. Очевидное решение: использовать разные счётчики инструкций и разные потоки событий для разных ЦПУ. Данная проблема ещё подлежит исследованию.

Список литературы

[1]. QEMU open source processor emulator — http://wiki.qemu.org/Main_Page
[2]. G.Altekar, I.Stoica. ODR: Output-Deterministic Replay for Multicore Debugging, UC Berkley, October 2009.
[3]. А.Ю.Тихонов, А.И. Аветисян. Развитие taint-анализа для решения задачи поиска программных закладок. Труды Института системного программирования РАН, том 20, 2011 г., стр. 9-24.
[4]. P. Colp, S. Dadizadeh, M. Nanavati. Deterministic Replay for Xen. Department of Computer Science. University of British Columbia. Vancouver, BC, Canada
[5]. Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, Boris Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. Workshop on Modeling, Benchmarking and Simulation (MoBS), June 2007.
[6]. К. Батузов, П. Довгалюк, В. Кошелев, В. Падарян. Два способа организации механизма полносистемного детерминированного воспроизведения в симуляторе QEMU. Труды Института системного программирования РАН, том 22, 2012г., стр. 77-94.
[7]. Довгалюк П. Детерминированное воспроизведение процесса выполнения программ в виртуальной машине. Труды Института системного программирования РАН, том 21, 2011г., с. 123-132.
[8]. NBench benchmark port to Linux/Unix — http://www.tux.org/~mayer/linux/bmark.html
[9]. Iperf — The TCP/UDP Bandwidth Measurement Tool — https://iperf.fr

Deterministic Replay Specifics in Case of Minimal Device Set

V.Y. Efimov <real@ispras.ru>
K.A. Batuzov <batuzovk@ispras.ru>
V.A. Padaryan <vartan@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004

Abstract. The deterministic replay technique can be used for debugging, improving reliability and robustness, software development and incident investigation (including reverse engineering of malware). The paper describes the implementation of deterministic replay of IA-32 based boards in QEMU. Another implementation of this technique in QEMU had been previously published, but it uses a significantly different approach. Deterministic replay implementation details and features substantially depend on deterministic area — the part of virtual machine which execution is being replayed. For replay to be deterministic the implementation must ensure that (1) all information flows across deterministic area borders should be logged and then replayed, and (2) there is no non-determinism inside deterministic area. The proposed approach is called «Min VM» because it's based on the minimal deterministic area while the former one should be called «Max VM» as it attempts to stretch deterministic area to cover whole virtual machine. The proposed approach shows the advantages of lower time overhead for logging phase and easier support (because it is much easier to ensure determinism of small deterministic area). On the other side the shortcoming is larger log size mostly because deterministic area doesn't include hard disks so all data flows from disks are being logged. It makes the self-sufficient replay log: image of the original HDD is not needed to replay the execution. The implementation has been tested on popular operating systems: Windows XP, Windows 7 and GNU/Linux 3.12. The current implementation shows 6 – 42% slowdown depending on application code that exceeds previous approach slowdown (17 – 79%).

Keywords: deterministic replay, emulator, QEMU, virtual machine.

DOI: 10.15514/ISPRAS-2015-27(2)-5

For citation: Efimov V.Y., Batuzov K.A., Padaryan V.A. Deterministic Replay Specifics in Case of Minimal Device Set. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 65-92 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-5.

References

[1]. QEMU open source processor emulator — http://wiki.qemu.org/Main_Page
[2]. G.Altekar, I.Stoica. ODR: Output-Deterministic Replay for Multicore Debugging, UC Berkley, October 2009.

[3]. A.Ju.Tihonov, A.I. Avetisjan. Razvitie taint-analiza dlja reshenija zadachi poiska programmnyx zakladok [Development of taint analysis to search for software backdoors]. Trudy ISP RAN [The Proceedings of ISP RAS], volume 20, 2011. p. 9-24. (In Russian)
[4]. P. Colp, S. Dadizadeh, M. Nanavati. Deterministic Replay for Xen. Department of Computer Science. University of British Columbia. Vancouver, BC, Canada
[5]. Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, Boris Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. Workshop on Modeling, Benchmarking and Simulation (MoBS), June 2007.
[6]. K. Batuzov, P. Dovgaljuk, V. Koshelev, V. Padarjan. Dva sposoba organizacii mexanizma polnosistemnogo determinirovannogo vosproizvedenija v simuljatore QEMU [Two approaches of deterministic replay development for QEMU system emulator]. Trudy ISP RAN [The Proceedings of ISP RAS], volume 22, 2012 г. p. 77-94. (In Russian)
[7]. Dovgaljuk P. Determinirovannoe vosproizvedenie processa vypolnenija programm v virtualqnoj mashine [A deterministic replay of software execution in virtual machine]. Trudy ISP RAN [The Proceedings of ISP RAS]. volume 21, 2011, p. 123-132. (In Russian)
[8]. NBench benchmark port to Linux/Unix — http://www.tux.org/~mayer/linux/bmark.html
[9]. Iperf — The TCP/UDP Bandwidth Measurement Tool — https://iperf.fr

Поиск семантических ошибок, возникающих при некорректной адаптации скопированных участков кода

Севак Саргсян <sevaksargsyan@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация: В статье предлагается новый метод поиска семантических ошибок, возникающих при неправильном копировании исходного кода в процессе разработки ПО. Метод состоит из двух основных этапов. На первом этапе производится поиск клонов кода на основе лексического анализа программы. Найденные идентичные последовательности лексем фильтруются путем частичного разбора. После чего в них остаются целостные конструкции, допускаемые языком программирования. На втором этапе производится анализ найденных клонов с целью обнаружения допущенных ошибок при копировании. Для этого строится и анализируется граф зависимостей программы (Program Dependence Graph - PDG). Предложенный подход реализован в компиляторной инфраструктуре LLVM/Clang, что позволяет эффективным образом производить анализ, во время компиляции проекта. Найденные ошибки выдаются в виде предупреждений для разработчика. В статье приводится результаты анализа ядра Linux 2.6 и Android 4.3. Инструмент обеспечивает точность выше 65%.
Ключевые слова: семантический анализ, семантические ошибки, поиск клонов, PDG, LLVM

DOI: 10.15514/ISPRAS-2015-27(2)-6

Для цитирования: Саргсян Севак. Поиск семантических ошибок, возникающих при некорректной адаптации скопированных участков кода. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 93-104. DOI: 10.15514/ISPRAS-2015-27(2)-6.

1. Введение

В процессе разработки программного обеспечения (ПО) часто прибегают к копированию ранее написанного кода, что может стать причиной возникновения семантических ошибок в программе. Чаще всего ошибки возникают из-за необновленных имен переменных в скопированном участке. Программы, насыщенные клонами кода, с большой вероятностью будут

содержать много ошибок. Инструменты поиска клонов кода и семантических ошибок широко применяются в процессе разработки ПО. Согласно исследованиям [1, 2] до двадцати процентов кода может быть клонами. Существуют пять основных подходов к поиску клонов. Текстовый [3] и лексический [4] подходы не могут найти все 3 типа (раздел 2) клонов. Синтаксический подход [5, 6] и метод, основанный на метриках [7, 8] находят третий тип клонов с низкой точностью. Подход, основанный на семантическом анализе программы [9, 10, 11, 12], находит все три типа клонов кода с большой точностью, но у этого подхода большая вычислительная сложность. Есть две основные причины медленной работы: первая это повторный анализ исходного кода для построения PDG, вторая это поиск клонов кода на основе изоморфизма подграфов PDG. Известно, что поиск максимально изоморфных подграфов – NP-сложная задача, и для ее решения применяются приближенные алгоритмы, у которых сложность может быть кубической от количества вершин в PDG.

Анализ больших проектов с открытыми исходными кодами показал, что большое количество ошибок возникает из-за неверно адаптированного кода, так, например, репозитории FreeBSD и Linux по данным на 2013 содержали более 113 и 182 исправлений подобных ошибок [13]. Для поиска ошибок, возникающих из-за неправильной адаптации скопированного кода, как правило, используют методы, основанные либо на лексическом анализе, либо на синтаксическом анализе.

Методы, основанные на лексическом подходе, трансформируют исходный код в последовательность лексем и ищут идентичные последовательности лексем (клоны кода) [14]. На втором этапе проверяются переменные клонированных фрагментов кода, которые используются больше одного раза в данном фрагменте. Если соответствующие переменные во втором фрагменте имеют разные имена, тогда произошло некорректное переименование переменных и данный фрагмент содержит семантическую ошибку. Недостаток такого подхода заключается в большом количестве ложных срабатываний, так как подход не учитывает контекст участка программы вокруг потенциального клона.

Методы, основанные на синтаксическом подходе [13, 15, 16], трансформируют исходный код в абстрактное синтаксическое дерево (AST – abstract syntax tree), и ищут схожие поддеревья. Затем производится анализ найденных поддеревьев для выявления возможных ошибок. Большинство известных инструментов используют [13, 17] репозиторий программы, что является дополнительным ограничением. Каждое изменение в репозитории анализируется, рассматривается его влияние на AST каждой функции и производится поиск дефектов. Недостаток такого подхода заключается в том, что некорректно переименованные переменные влияют на вид AST, поскольку могут появляться/исчезать узлы дерева для выражений (рис. 1).

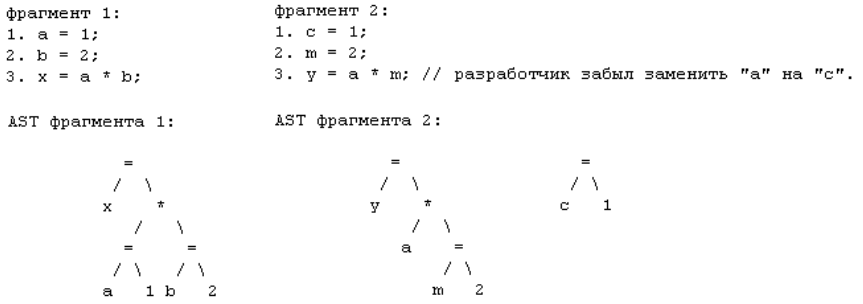

Рис.1. Влияние имен переменных на вид AST, после некорректного переименования

Данная работа описывает новый подход нахождения семантических ошибок, возникающих при неправильной адаптации копированного кода. Он состоит из двух основных этапов. На первом этапе обнаруживаются все клоны второго типа (раздел 2) путем лексического анализа функции. Второй этап строит PDG для этой функции, чтобы найти ошибки, допущенные при копировании.

2. Типы клонов

Есть три основных типа клонов (классификация приведена из [18]). Первый тип (**T1**) – это те фрагменты кода, которые отличаются только пробелами и комментариями. Второй тип (**T2**) – это те фрагменты кода, которые отличаются пробелами, комментариями, именами переменных, типами переменных и значениями переменных. Третий тип (**T3**) – это те фрагменты кода, которые отличаются пробелами, комментариями, именами переменных, типами переменных, значениями переменных. В конкретном фрагменте могут быть также добавлены или удалены некоторые строки.

3. Модель инструмента поиска семантических ошибок

Генерация лексем и построение PDG проекта производится во время компиляции проекта (рис. 2). Новый проход LLVM [19] строит последовательность лексем на основе промежуточного представления. Для каждой функции производится поиск клонов. Клонами считаются совпадающие последовательности лексем. Существуют ряд широко известных инструментов, которые работают на основе лексического подхода [4, 14]. Есть две основные причины того, что предлагаемый метод на первом этапе использует лексический анализ для поиска клонов кода. Основная причина возникновения семантических ошибок - это непереименование переменных после копирования участка кода (рис. 2). Лексический анализ не чувствителен к именам переменных, что помогает найти скопированные участки кода, в которых есть непереименованные переменные (семантические ошибки). При

использовании AST и PDG, имена переменных могут повлиять на вид графа (рис. 1, 5).

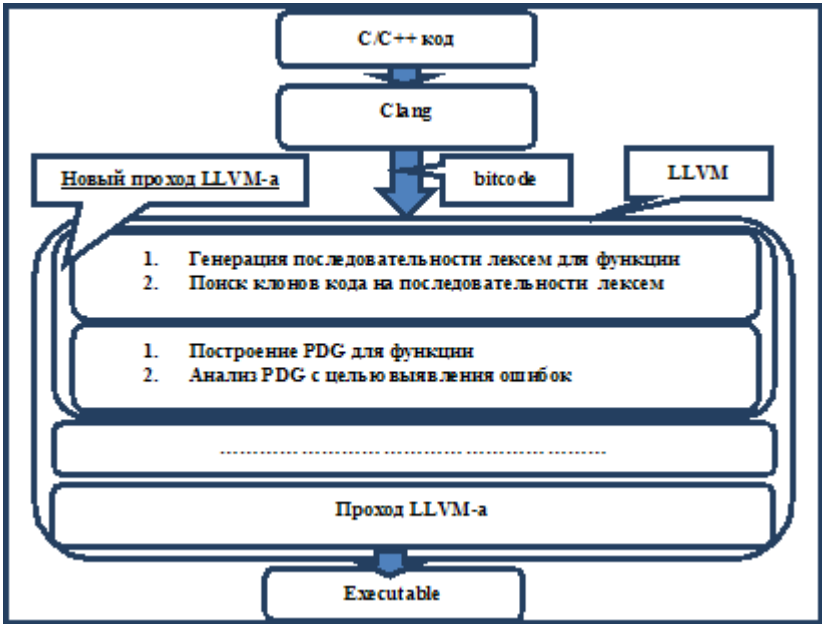

Рис. 2. Поиск клонов кода на базе компиляторной инфраструктуры LLVM.

Для функции, в которой найдена пара клонов, строится PDG для проверки корректности копирования. Вершинами PDG являются инструкции промежуточного представления LLVM. Ребрам соответствуют зависимости по управлению между инструкциями, зависимости, получаемые анализом алиасов и LLVM use-def анализом [19]. Такой подход имеет ряд преимуществ. Он позволяет без повторного анализа исходного кода получить лексемы и графы для больших проектов, содержащих миллионы строк исходного кода. Не требуется проводить дополнительный анализ зависимостей между единицами компиляции.

По сравнению с существующими методами [14, 15] предложенный подход обладает большей точностью, благодаря тому, что использует представление PDG, содержащее всю необходимую информацию о программе, для поиска ошибок.

4. Поиск клонов кода на основе лексического анализа

На первом этапе на основе промежуточного представления LLVM получается последовательность лексем функции (рис. 3). Предлагаемый алгоритм, обрабатывает последовательность лексем и находит все непересекающиеся

пары идентичных подпоследовательностей максимального размера. После этого производится частичный разбор идентичных последовательностей, для корректного определения синтаксических конструкций языка. Идентичные последовательности фильтруются от неполных конструкций (например, если с телом цикла вошли в подпоследовательность и другие лексемы, которые не представляют собой целую инструкцию или блок, то они удаляются из обеих последовательностей). Если отфильтрованные идентичные последовательности имеют достаточно большой размер, они передаются на второй этап для проверки.

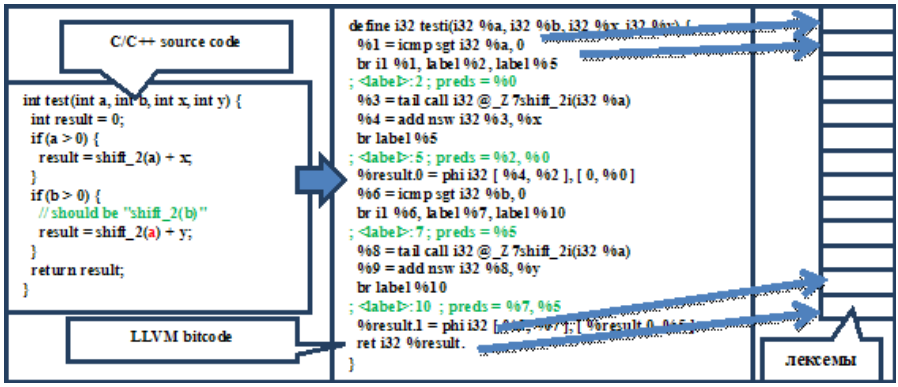

Рис. 3. Построение последовательности лексем.

5. Поиск ошибок

Для поиска ошибок в скопированном фрагменте кода, строится PDG граф соответствующей функции. В полученном графе выделяются два подграфа, соответствующие идентичным последовательностям лексем. Полученные подграфы расширяются путем добавления вершин (назовем эти вершин исходными вершинами - ИВ), соответствующих переменным, которые используются в выделенных подграфах (рис. 4).

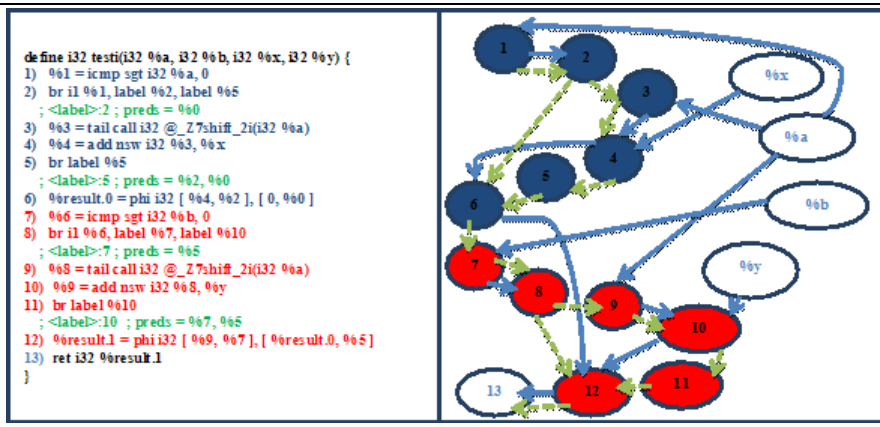

Рис. 4. PDG для функции. {1, 2, 3, 4, 5, 6, %x, %a}, {7, 8, 9, 10, 11, 12, %a, %b, %y} выделенные подграфы.

Если выделенные подграфы не изоморфны, то при копировании, с большой вероятностью, произошла ошибка, поэтому нужен дополнительный анализ. Анализ ИВ дает информацию о возможной ошибке. Если есть ИВ, которые входят в выделенные подграфы, и в каждом подграфе имеют разную степень (рис. 5), тогда переменные, соответствующие этим вершинам, содержат ошибочное использование.

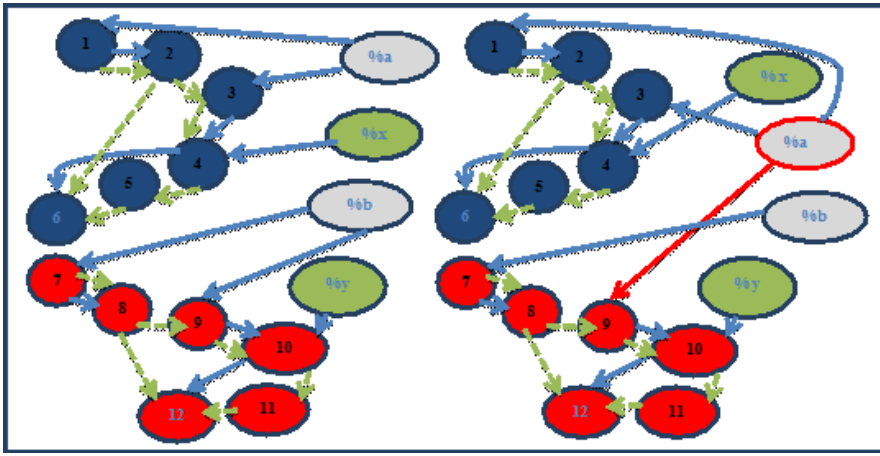

Рис. 5. Изменение структуры PDG при неправильном копировании кода.

6. Проверка изоморфизма на основе метрик

Для проверки изоморфизма пары PDG, сравниваются значения метрик, рассчитанные для ребер соответствующих графов. Каждая вершина PDG имеет индекс, который представляет собой код операции соответствующей инструкции промежуточного представления LLVM. Метрика ребра представляет собой число, которое получается на основе индексов соседних вершин и степени одной вершины.

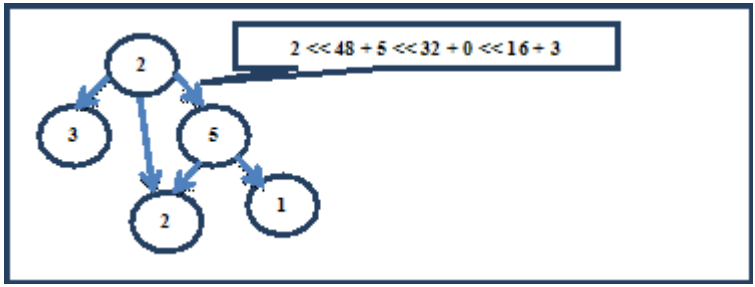

Рис. 6. Метрика ребер PDG

Из рис. 6 видно, что метрика ребра представляется в виде 64 битного целого числа. Первые 16 битов - это индекс первой вершины, следующие 16 битов - это индекс второй вершины, после этого следует количество входящих и выходящих ребер начальной вершины, которым также выделено по 16 битов. Алгоритм проверки изоморфизма вычисляет и сохраняет метрики ребер каждого графа в отдельном множестве, после чего проверяет равенство полученных множеств. Если они неравны, тогда пара PDG графов не может быть изоморфной.

7. Результаты

Ниже представлены результаты анализа нескольких проектов с открытым исходным кодом (рис. 7). Приведен пример семантической ошибки: разработчик не переименовал переменную 'userName' на 'password' (строка 277) после копирования (рис. 8).

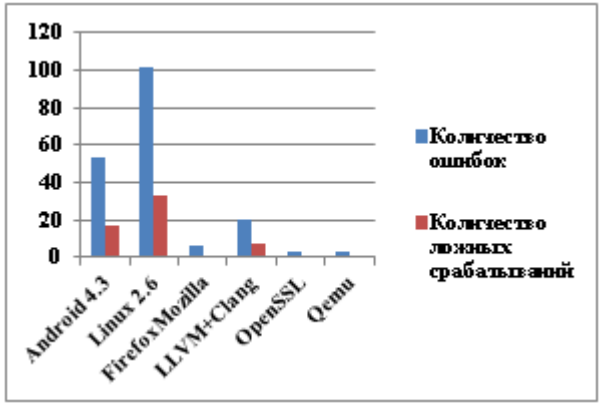

Рис. 7. График найденных семантических ошибок и ложных срабатываний.

Android 4.3 - AccountSetupBasics.java

```
271|    String userName = SetupData.getUsername();
272|    if (userName != null) {
273|        mEmailView.setText(userName);
274|        SetupData.setUsername(null);
275|    }
-----------------------------------------------------
276|    String password = SetupData.getPassword();
277|    if (userName != null) {
278|        mPasswordView.setText(password);
279|        SetupData.setPassword(null);
280|    }
```

Рис. 8. Пример найденной семантической ошибки.

8. Заключение

В статье предлагается новый подход поиска семантических ошибок, возникающих при неправильном копировании участков исходного кода. Применяется гибридный анализ исходного кода для выявления возможных ошибок. Метод использует лексический анализ и частичный разбор для поиска клонов кода, после чего производится семантический анализ. В ходе семантического анализа выявляются некорректно обновленные участки кода, использование которых может содержать ошибку. Предложенный подход реализован в компиляторной инфраструктуре LLVM/Clang, что позволяет выявить семантические ошибки проекта во время компиляции.

Список литературы.

[1]. B. Baker, On finding duplication and near-duplication in large software systems, Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE 1995, pp. 86-95, 1995.

[2]. C. K. Roy and J. R. Cordy, An empirical study of function clones in open source software systems, Proceedings of the 15th Working Conference on Reverse Engineering, WCRE 2008, pp. 81-90, 2008.

[3]. S. Ducasse, M. Rieger and S. Demeyer, A language independent approach for detecting duplicated code, Proceedings of the 15th International Conference on Software Maintenance, (ICSM'99), Oxford, England, UK, pp. 109-119, 1999.

[4]. T.Kamiya, S.Kusumoto and K.Inoue, CCFinder: A multilinguistic token-based code clone detection system for large scale source code", IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654-670, 2002.

[5]. I. Baxter, A. Yahin, L. Moura and M. Anna, Clone detection using abstract syntax trees, Proceedings of the 14th IEEE International Conference on Software Maintenance, IEEE Computer Society, pp. 368-377, 1998.

[6]. L.Jiang, G.Misherghi, Z.Su and S.Glondu, DECKARD : Scalable and accurate tree-based detection of code clones", Proceedings of the 29th International Conference on Software Engineering, (ICSE07), IEEE Computer Society, pp. 96-105, 2007.

[7]. J. Mayrand, C. Leblanc and E. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, Proceedings of the 12th International Conference on Software Maintenance, (ICSM96), Monterey, CA, USA, pp. 244-253, 1996.

[8]. Sargsyan S., Kurmangaleev S., Baloian A., Aslanyan H., Scalable and Accurate Clones Detection Based on Metrics for Dependence Graph, Mathematical Problems of Computer Science, Volume 42, pp. 54-62, 2014.

[9]. R.Komondoor and S.Horwitz, Using slicing to identify duplication in source code, Proceedings of the 8th International Symposium on Static Analysis, pp. 40-56, 2001.

[10]. J. Krinke, Identifying similar code with program dependence graphs, Proceedings of the 8th Working Conference on Reverse Engineering, (WCRE 2001), pp. 301-309, 2001.

[11]. Y. Higo and S. Kusumoto, Code clone detection on specialized PDGs with heuristics, Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR11), Oldenburg, Germany, pp.75-84, 2011.

[12]. С. Саргсян, Ш. Курмангалеев, А. Белеванцев, А. Асланян, А. Балоян, Масштабируемый инструмент поиска клонов кода на основе семантического анализа программ, Труды Института системного программирования РАН Том 27. Выпуск 1. 2015 г.

[13]. K. Miryung, S. Person, N. Rungta, Detecting and characterizing semantic inconsistencies in ported code, Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference, pp. 367-377

[14]. Z. Li, S. Lu, S. Myagmar, Y. Zhou, CP-Miner: Finding copy-paste and related bugs in large-scale software code, IEEE Transactions on Software Engineering 32 (3) (2006) 176-192.

[15]. P. Jablonski, D. Hou, CReN: A Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE, in Proceedings of the 2007 OOPSLA workshop on eclipse technology, 2007, pp.16-20.

[16]. L. Jiang, Z. Su, E. Chiu, Context-Based Detection of Clone-Related Bugs, in Proceedings of the 6th joint meeting of the European software engineering conference, 2007, pp. 55-64.
[17]. Y. Higo, S. Kusumoto, MPAnalyzer: A Tool for Finding Unintended Inconsistencies in Program Source Code, in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp.843-846.
[18]. Bellon S., Koschke R., Antoniol G., Krinke J., Merlo E., Comparison and evaluation of clone detection tools, Transactions on Software Engineering 33 (9) (2007) 577–591
[19]. http://llvm.org

Copy-Paste Semantic Errors Detection

Sevak Sargsyan <sevaksargsyan@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004

Annotation. The paper describes a method for semantic errors detection arising during incorrect code copy-paste made by the developer. The method consists of two basic parts. The first part detects code clones based on lexical analysis of the program. A sequence of tokens is constructed based on the LLVM lexer and then all pairs of maximal, non-intersected matched token sequences are detected. The pairs of identical subsequences are then partially parsed to retain the constructs allowed by the programming language and to remove the incomplete sequences. When the remaining subsequences are big enough, the second stage is applied for them. A Program Dependence Graph (PDG) is constructed for the corresponding function code, and then identical subsequences' subgraphs are considered. If two subgraphs have shared vertices, then outgoing edges of these vertices are analyzed. This allows detecting semantic errors with high accuracy. The described method is implemented for the LLVM/Clang compiler. Due to this semantic mistakes are detected during program compile time, so there is no need for separate lexical and semantic program analysis. A number of widely used open source libraries and software systems were analyzed. The paper contains the list of detected semantic errors for Linux kernel 2.6 and Android 4.3. For these systems, the true positive rate achieved by our approach is above 65%.

Keywords: lexical analysis; semantic analysis; code clones; PDG; LLVM

DOI: 10.15514/ISPRAS-2015-27(2)-6

For citation: Sargsyan Sevak. Copy-Paste Semantic Errors Detection. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 93-104 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-6.

References.

[1]. B. Baker, On finding duplication and near-duplication in large software systems, Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE 1995, pp. 86-95, 1995.

[2]. C. K. Roy and J. R. Cordy, An empirical study of function clones in open source software systems, Proceedings of the 15th Working Conference on Reverse Engineering, WCRE 2008, pp. 81-90, 2008.

[3]. S. Ducasse, M. Rieger and S. Demeyer, A language independent approach for detecting duplicated code, Proceedings of the 15th International Conference on Software Maintenance, (ICSM'99), Oxford, England, UK, pp. 109-119, 1999.

[4]. T.Kamiya, S.Kusumoto and K.Inoue, CCFinder: A multilinguistic token-based code clone detection system for large scale source code", IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654-670, 2002.

[5]. I. Baxter, A. Yahin, L. Moura and M. Anna, Clone detection using abstract syntax trees, Proceedings of the 14th IEEE International Conference on Software Maintenance, IEEE Computer Society, pp. 368-377, 1998.

[6]. L.Jiang, G.Misherghi, Z.Su and S.Glondu, DECKARD : Scalable and accurate tree-based detection of code clones", Proceedings of the 29th International Conference on Software Engineering, (ICSE07), IEEE Computer Society, pp. 96-105, 2007.

[7]. J. Mayrand, C. Leblanc and E. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, Proceedings of the 12th International Conference on Software Maintenance, (ICSM96), Monterey, CA, USA, pp. 244-253, 1996.

[8]. Sargsyan S., Kurmangaleev S., Baloian A., Aslanyan H., Scalable and Accurate Clones Detection Based on Metrics for Dependence Graph, Mathematical Problems of Computer Science, Volume 42, pp. 54-62, 2014.

[9]. R.Komondoor and S.Horwitz, Using slicing to identify duplication in source code, Proceedings of the 8th International Symposium on Static Analysis, pp. 40-56, 2001.

[10]. J. Krinke, Identifying similar code with program dependence graphs, Proceedings of the 8th Working Conference on Reverse Engineering, (WCRE 2001), pp. 301-309, 2001.

[11]. Y. Higo and S. Kusumoto, Code clone detection on specialized PDGs with heuristics, Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR11), Oldenburg, Germany, pp.75-84, 2011.

[12]. S. Sargsyan, S. Kurmnagaleev, A. Belevantsev, H. Aslanyan, A. Baloian, Scalable code clone detection tool based on semantic analysis, The Proceedings of ISP RAS, vol. 27, issue 1, 2015.

[13]. K. Miryung, S. Person, N. Rungta, Detecting and characterizing semantic inconsistencies in ported code, Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference, pp. 367-377

[14]. Z. Li, S. Lu, S. Myagmar, Y. Zhou, CP-Miner: Finding copy-paste and related bugs in large-scale software code, IEEE Transactions on Software Engineering 32 (3) (2006) 176-192.

[15]. P. Jablonski, D. Hou, CReN: A Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE, in Proceedings of the 2007 OOPSLA workshop on eclipse technology, 2007, pp.16-20.

[16]. L. Jiang, Z. Su, E. Chiu, Context-Based Detection of Clone-Related Bugs, in Proceedings of the 6th joint meeting of the European software engineering conference, 2007, pp. 55-64.

[17]. Y. Higo, S. Kusumoto, MPAnalyzer: A Tool for Finding Unintended Inconsistencies in Program Source Code, in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp.843-846.
[18]. Bellon S., Koschke R., Antoniol G., Krinke J., Merlo E., Comparison and evaluation of clone detection tools, Transactions on Software Engineering 33 (9) (2007) 577–591
[19]. http://llvm.org

Метод выявления некоторых типов ошибок работы с памятью в бинарном коде программ

В.В. Каушан <korpse@ispras.ru>
А.Ю. Мамонтов <mamontov@ispras.ru>
В.А. Падарян <vartan@ispras.ru>
А.Н. Федотов <fedotoff@ispras.ru>
ИСП РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В статье рассматривается метод выявления ошибок работы с памятью в бинарном коде программ, таких как выход за границы буфера при чтении и записи. Предлагаемый метод основывается на использовании динамического анализа и символьного выполнения. Метод применяется к бинарным файлам программ без дополнительной отладочной информации. Описанный метод был реализован в виде программного инструмента. Возможности инструмента продемонстрированы на примере поиска ошибок в 11 программах, которые работают под управлением ОС Windows и Linux, в 7 из них ошибки не были исправлены на момент написания статьи.

Ключевые слова: выявление уязвимостей; бинарный код; динамический анализ; символьное выполнение.

DOI: 10.15514/ISPRAS-2015-27(2)-7

Для цитирования: Каушан В.В., Мамонтов А.Ю., Падарян В.А., Федотов А.Н. Метод выявления некоторых типов ошибок работы с памятью в бинарном коде программ. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 105-126. DOI: 10.15514/ISPRAS-2015-27(2)-7.

1. Введение

На сегодняшний день, важными практическими задачами в компьютерной безопасности является поиск ошибок в программном обеспечении (ПО). Люди всё чаще используют различное ПО для собственных нужд: передача и получение разной (в том числе и конфиденциальной) информации, использование программ для работы и т.д. Таким образом, обеспечение надёжности, конфиденциальности и доступности работающего программного обеспечения является актуальной задачей.

Нарушать работу программного обеспечения могут различного рода ошибки. Среди всевозможных типов ошибок присутствует большой класс – ошибки в реализации (дефекты). В свою очередь данные ошибки разделяются на множество различных подклассов [1]. Мы будем рассматривать дефекты, приводящие к нарушению доступа к памяти. Под нарушением доступа к памяти, будем понимать выход за границы буфера в памяти при операции чтения или записи в этот буфер. В статье предлагается метод поиска таких дефектов. Зачастую нарушение доступа к памяти происходит из-за неправильного копирования данных, выделения и освобождения памяти. Поиск дефектов такого рода осуществляется в бинарном коде, где отсутствует какая-либо информация о границах буферов памяти. Следовательно, эту информацию необходимо восстановить. Предложенный метод основан на анализе трасс выполнения программ, полученных при помощи полносистемного симулятора [2-7]. Для заданного набора входных данных фиксированного размера (далее эти входные данные обозначаются как *префикс*), подбираются значения префикса и значение длины этого префикса, при которых происходит нарушение работы с памятью. Для подбора этих данных используется технология символьной интерпретации [8]. Применение символьной интерпретации для анализа трасс выполнения описано в работе [9]. Одним из дополнений в предлагаемом подходе по сравнению с работой [9] является наличие абстрактного значения длины у начального набора данных. Помимо того, предлагается особая обработка некоторых функций: ввод пользовательских данных, строковые функции, функции выделения и освобождения памяти.

Статья организована следующим образом. Во втором разделе описываются теоретические аспекты предлагаемого метода. В третьем разделе описана общая схема работы программного инструмента, реализующего данный метод. В четвертом разделе рассказывается о деталях реализации этого инструмента. В пятом разделе представлены результаты применения данного метода. В шестом разделе представлен обзор близких работ. В последнем, седьмом, разделе анализируются результаты и обсуждаются дальнейшие направления исследований.

2. Моделирование работы с буферами памяти в бинарном коде

Для решения поставленной задачи необходимо формально описать правила доступа к памяти, при нарушении которых возникает ошибочная ситуация. Чаще всего, нарушение работы с памятью происходит во время выполнения машинных команд, которые обращаются к памяти с использованием косвенной адресации. При косвенном обращении к памяти, адрес ячейки памяти, в который происходит загрузка или выгрузка данных, может зависеть от входных данных, что потенциально позволяет обращаться за границы допустимой области памяти. Эти машинные команды могут входить в состав

библиотечных функций работы с памятью. Современные версии библиотечных функций используют расширения процессора, такие как SSE2, для ускорения работы. Анализ машинных команд SSE2 значительно усложняет задачу поиска ошибок. В то же время, семантика многих библиотечных функций работы с памятью известна, что позволяет обрабатывать их как единое целое вместо обработки отдельных инструкций, входящих в эти функции. Можно значительно упростить задачу поиска ошибок доступа к памяти используя следующий подход. Для машинных команд, входящих в состав функций с известной семантикой, выполняется обработка функций целиком с помощью правил, соответствующих семантике этих функций. Для всех остальных машинных команд используется свой набор правил. Таким образом, вытекает необходимость явного выделения в коде программы функций, отвечающих за работу с памятью и формального описания их свойств, что позволит вести учет доступных буферов памяти и обнаруживать выход за их пределы. Помимо того, анализ помеченных данных требует задания функций, осуществляющих ввод пользовательских данных.

Разобьём правила доступа к памяти на две группы:

- правила, описывающие нарушения при использовании библиотечных функций;
- правила, описывающие нарушения при косвенной адресации в машинных инструкциях.

Для задания правил первой группы важным классом функций являются строковые функции, такие как: копирование, конкатенация и определение длины нуль-терминированных строк. В предположении, что семантика таких функций известна, их можно моделировать целиком, не погружаясь в код реализации.

При описании правил доступа к памяти на уровне машинных инструкций достаточно рассматривать адрес памяти, по которому происходит доступ.

Приведенные рассуждения приводят к следующим определениям и правилам интерпретации трассы машинных команд.

Под трассой будем понимать упорядоченную последовательность пар

$$(\mathrm{instr}_0, \mathrm{M}_0)$$
$$(\mathrm{instr}_1, \mathrm{M}_1)$$
$$\dots$$
$$(\mathrm{instr}_{\mathrm{N}-1}, \mathrm{M}_{\mathrm{N}-1})$$

где $\mathrm{instr}_\mathrm{i}, 0 \le \mathrm{i} < N$ – выполнявшаяся на шаге i машинная команда, а M_i – состояние памяти компьютера перед выполнением этой команды. Память представляет набор адресуемых машинных слов $\mathrm{M} = (\mathrm{m}_0 \dots \mathrm{m}_{\mathrm{TOM}})$ в котором единообразно объединены все ячейки, обладающие состоянием: как, собственно, оперативная память компьютера, так и регистры.

Машинная команда – тройка, описывающая операцию над данными и ее фактические операнды: $\mathrm{instr} = \langle \text{КОП}, \{\mathrm{use}_\mathrm{i}\}, \{\mathrm{def}_\mathrm{j}\}\rangle$, где КОП – код операции,

$\mathrm{use}, \mathrm{def} \in \mathrm{M}$ – множества ячеек, считываемых и записываемых данной машинной командой. В наборах операндов явно указываются ячейки памяти, которые считываются и записываются при выполнении команды. Непосредственно адресуемые операнды не указываются. В случае косвенной адресации явно указываются обе ячейки: фактический операнд и ячейка, задающая адрес.

В работах, описывающих динамический анализ помеченных данных, перечень кодов операций реализует минимальный набор RISC команд [10], расширив его псевдокомандой ввода пользовательских данных. В данном случае расширение предполагает следующие 7 псевдокоманд:

- выделение памяти $\langle \mathrm{malloc}, \{\mathrm{size}\}, \{\mathrm{addr}\}\rangle$,
- освобождение памяти $\langle \mathrm{free}, \{\mathrm{addr}\}, \{\}\rangle$,
- определение длины строки $\langle \mathrm{strlen}, \{\mathrm{addr}\}, \{\mathrm{len}\}\rangle$,
- копирование строк $\langle \mathrm{strcpy}, \{\mathrm{dest.\,addr}, \mathrm{src.\,addr}\ , \}, \{\}\rangle$,
- конкатенация строк $\langle \mathrm{strcat}, \{\mathrm{dest.\,addr}, \mathrm{src.\,addr}\ , \}, \{\}\rangle$,
- копирование фиксированной длины $\langle \mathrm{memcpy}, \{\mathrm{dest.\,addr}, \mathrm{src.\,addr}\ , \mathrm{n}\}, \{\}\rangle$,
- ввод пользовательских данных $\langle \mathrm{read}, \{\mathrm{addr}, \mathrm{size}\}, \{\mathrm{readed}\}\rangle$

Остальные команды:

- унарная операция $\langle \Diamond_{\mathrm{u}}, \{\mathrm{use\_cell}\}, \{\mathrm{def\_cell}\}\rangle$,
- бинарная операция $\langle \Diamond_{\mathrm{b}}, \{\mathrm{use\_cell}_1, \mathrm{use\_cell}_2\}, \{\mathrm{def\_cell}\}\rangle$,
- загрузка данных из памяти $\langle \mathrm{load}, \{\mathrm{src\_cell}, \mathrm{src\_addr\_cell}\}, \{\mathrm{dest\_cell}\}\rangle$,
- выгрузка в память $\langle \mathrm{store}, \{\mathrm{src\_cell}, \mathrm{dest\_addr\_cell}\}, \{\mathrm{dest\_cell}\}\rangle$,
- безусловная передача управления $\langle \mathrm{jmp}, \{[\mathrm{addr\_cell}]\}, \{\}\rangle$
- для удобства интерпретации трассы явно разделены сработавший и не сработавший условные переходы: $\langle \mathrm{jct}, \{\mathrm{cond\_cell}[, \mathrm{addr\_cell}]\}, \{\}\rangle$ $\langle \mathrm{jcf}, \{\mathrm{cond\_cell}, \mathrm{addr\_cell}\}, \{\}\rangle$, соответственно.

Стоит отметить, что приведённый список псевдокоманд, обеспечивающих обработку функций с известной семантикой, не является конечным и при необходимости может быть расширен.

В процессе интерпретации поддерживается множество символьных переменных $\mathfrak{S}$, над которыми допустимы унарные и бинарные операции, а также операции загрузки и выгрузки данных.

Для описания работы с памятью используем понятие буфера, имеющего символьную длину [11]. Далее в тексте будем обозначать его как *L*-буфер. Задается *L*-буфер l в виде тройки $l = [\![\mathrm{base}, \mathrm{clen}, \mathrm{slen}]\!]$, содержащей адрес базы, реальную и символьную длину, отвечающую за абстрактный размер буфера. На рис. 1. схематично представлен *L*- буфер.

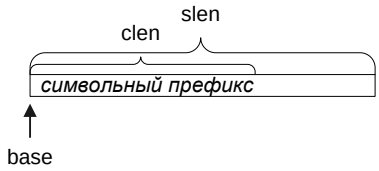

Рис. 1. L-буфер.

Интерпретация шагов трассы происходит в контексте, состоящем из четырёх компонент: текущего символьного состояния, предиката пути Pp, множества символьных переменных $\mathfrak{S}$ и двух множеств *L*-буферов памяти *A* и *I*.

Отображение Δ связывает ячейки (машинные слова) памяти и выражения над символьными переменными. Получение символьного и конкретного численного значения ячейки памяти будем описывать $\Delta[m]$ и $M[m]$ соответственно.

В ходе интерпретации трассы на основе предиката пути и дополнительных ограничений проверяются условия выхода за границы буферов. Для обозначения проверок вводится функция *Assert(условие)*, которая проверяет истинность заданного условия.

Также существует отдельный вид *L*-буферов, которые соответствуют областям выделенной памяти. У таких буферов абстрактная длина может быть константным выражением. В отличие от работы [11], в предлагаемом подходе каждый *L*-буфер помещен в одно из множеств: *A* или *I*, в зависимости от того, соответствует буфер выделению памяти или вводу данных. Входные данные могут поступать из различных источников: сеть, файлы, аргументы командной строки. Поддержка двух множеств *A* и *I* позволит в дальнейшем выполнять дополнительные проверки.

Введем отображение, позволяющие получить для заданного адреса буфер в заданном множестве $T(X, addr) = x, x \in X$, адрес *addr* указывает на одну из ячеек буфера *x, X* одно из множеств *A* и *I*. Для краткости будем в дальнейшем обозначать отображения, связанные с конкретным множеством буферов как *T(A)* и *T(I)*.

Для отображений *T* и Δ введём операцию обновления $\leftarrow$. Например, задание связи между ячейкой памяти m и символьным выражением s будем обозначать как $\Delta[m \leftarrow s]$. Правила интерпретации команд представлены ниже в виде продукций. Для каждой команды приводится ее запись, под которой размещена продукция, описывающая преобразование контекста. В верхней части продукции приведены выполняющиеся действия, в нижней – состояния контекста до и после интерпретации команды. Исходя из рассмотренного выше, контекст представляется кортежем $\Delta, Pp, A, I, \mathfrak{S}$. Для краткости, в нижней части продукции показываются только изменившиеся элементы контекста.

$$\langle \text{malloc}, \{\text{size}\}, \{\text{addr}\} \rangle$$

$$\frac{\text{l} = [\![\text{M}(\text{addr}), \text{M}(\text{ size}), \Delta[\text{size}]]\!]}{\text{A} \vdash \text{A} \cup \text{l}}$$

Malloc. Во множестве *A* происходит создание *L*-буфера. В зависимости от операнда *size*, абстрактная длина *L*-буфера может иметь как символьное, так и константное (конкретное) значение.

$$\langle \text{free}, \{\text{addr}\}, \{\} \rangle$$

$$\frac{\text{l} = \text{T}\big(\text{A}, \text{M}(\text{addr})\big), \text{Assert}(\text{M}(\text{addr}) == \text{l}.\text{base})}{\text{A} \vdash \text{A} \backslash \text{l}}$$

Free. Происходит удаление *L*-буфера из множества *A*, конкретное значение операнда *addr* должно соответствовать полю base у *L*-буфера l.

$$\langle \text{strlen}, \{\text{addr}\}, \{\text{len}\} \rangle$$

$$\frac{\text{l} = \text{T}\big(\text{I}, \text{M}(\text{addr})\big)}{\Delta \vdash \Delta[\text{len} \leftarrow \text{l}.\text{slen} - (\text{M}(\text{addr}) - \text{l}.\text{base})]}$$

Strlen. Происходит присваивание операнду *len* значения абстрактной длины *L*-буфера, который найден, исходя из значения операнда *addr*.

$$\langle \text{strcpy}, \{\text{dest}.\text{addr}, \text{src}.\text{addr} , \dots\}, \{\dots\} \rangle$$

$$\frac{\text{i} = \text{T}\big(\text{I}, \text{M}(\text{src}.\text{addr})\big), \text{i}' = \left[\!\!\left[\begin{matrix}\text{M}(\text{dst}.\text{addr}), \text{i}.\text{clen} - (\text{M}(\text{src}.\text{addr}) - \text{i}.\text{base}), \\ \text{i}.\text{slen} - (\text{M}(\text{src}.\text{addr}) - \text{i}.\text{base})\end{matrix}\right]\!\!\right], \quad \text{a} = \text{T}\big(\text{A}, \text{M}(\text{dst}.\text{addr})\big), \text{Assert}(\text{i}' \sqsubset \text{a})}{\Delta, \text{I} \vdash \Delta[\text{m}_{\text{i}'.\text{base}}, \dots, \text{m}_{\text{i}'.\text{base}+\text{i}'.\text{clen}-1} \leftarrow \text{m}_{\text{M}(\text{src}.\text{addr})}, \dots, \text{m}_{\text{M}(\text{src}.\text{addr})+\text{i}'.\text{clen}-1}], \text{I} \cup \text{i}'}$$

Strcpy (Strcat). Происходит проверка на переполнение буфера при копировании (конкатенации), также создаётся новый буфер из множества *I*, затем обновляется отображения ячеек памяти на символьные переменные.

$$\langle \text{strcat}, \{\text{dest.addr}, \text{src.addr}\ , \}, \{\}\rangle$$

$$\frac{\begin{array}{c} i_{src} = T\big(I, M(\text{src.addr})\big), i_{dst} = T\big(I, M(\text{dst.addr})\big), \\ \text{offset}_{dst} = \ (M(\text{dst.addr}) - i_{dst}.\text{base}), \\ \text{offset}_{src} = (M(\text{src.addr}) - i_{src}.\text{base}), \\ i' = \left[\!\!\left[\begin{array}{c} M(\text{dst.addr})\ , i_{src}.\text{clen} \ - \text{offset}_{src} + \ i_{dst}.\text{clen}, \\ i_{src}.\text{slen} - \text{offset}_{src} + i_{dst}.\text{slen} \end{array} \right]\!\!\right], \\ a = T\big(A, M(\text{dst.addr})\big), \text{Assert}(i' \sqsubset a) \end{array}}{\Delta, I \vdash \Delta[m_{i_{dst}.\text{base}+i_{dst}.\text{clen}}, .., m_{i_{dst}.\text{base}+i'.\text{clen}-1} \leftarrow m_{M(\text{src.addr})}, .., m_{M(\text{src.addr})+i_{src}.\text{clen}-\text{offset}_{src}-1}], I \cup i' \backslash i_{ds}}$$

$$\langle \text{memcpy}, \{\text{dest.addr}, \text{src.addr}\ , \text{n}\}, \{\}\rangle$$

$$\frac{\begin{array}{c} i_{src} = T\big(I, M(\text{src.addr})\big), i_{dst} = T\big(I, M(\text{dst.addr})\big), \\ i' = [\![M(\text{dst.addr})\ , M(\text{n}), i_{src}\Delta(\text{n})]\!], \\ \text{Assert}\big(i_{src}.\text{slen} \ - (\text{src.addr} - \ i_{src}.\text{base}) < \Delta(\text{n})\big) \\ a_{src} = \ T\big(A, M(\text{src.addr})\big), \ \text{Assert}(\ i_{src} \sqsubset \ a_{src}) \\ a = T\big(A, M(\text{dst.addr})\big), \text{Assert}(i' \sqsubset a) \end{array}}{\Delta, I \vdash \Delta\big[m_{i'.\text{base}}, .., m_{i'.\text{base}+i'.\text{clen}-1} \leftarrow m_{M(\text{src.addr})}, .., m_{M(\text{src.addr})+i'.\text{clen}-1}\big], I \cup i'}$$

Memcpy. Происходит проверка на выходы за границы при чтении буфера источника и на переполнение буфера назначения при копировании, далее создаётся новый буфер из множества *I*, затем обновляется отображения ячеек памяти на символьные переменные.

$$\langle \text{read}, \{\text{addr}, \text{size}\}, \{\text{readed}\}\rangle$$

$$\frac{\begin{array}{c} a_{src} = \ T\big(A, M(\text{src.addr})\big), \ s_0, .. s_{M(\text{readed})} \in \mathfrak{S} \\ i_{src} = \ [\![M(\text{addr})\ , M(\text{readed}), s_{\text{len}}]\!], s_{\text{len}} \in \mathfrak{S} \\ \text{Assert}\big(\ i_{src} \ \sqsubset \ a_{src} \ \& \wedge i_{src}.\text{slen} \leq M(\text{size})\big) \end{array}}{\Delta, I \vdash \Delta\big[m_{i_{src}.\text{base}}, .., m_{i_{src}.\text{clen}-1}, \leftarrow s_0, .. s_{M(\text{readed})}\ \big], I \ \cup i_{src}}$$

Read. Происходит проверка на выход за границы буфера при чтении данных из внешнего источника, учитывая максимальный размер считанных данных, также создаётся новый буфер из множества *I*, затем обновляется отображения ячеек памяти на символьные переменные.

$$\langle \Diamond_u, \{\text{use\_cell}\}, \{\text{def\_cell}\} \rangle$$

$$\frac{}{\Delta \vdash \Delta[\text{def\_cell} \leftarrow \Diamond_u \Delta[\text{use\_cell}]]}$$

Унарная операция (Бинарная операция). Происходит обновление отображения ячеек памяти на символьные переменные с учётом выполнения операции.

$$\langle \Diamond_b, \{\text{use\_cell}_1, \text{use\_cell}_2\}, \{\text{def\_cell}\} \rangle$$

$$\frac{}{\Delta \vdash \Delta[\text{def\_cell} \leftarrow \Delta[\text{use\_cell}_1] \Diamond_b \Delta[\text{use\_cell}_2]]}$$

$$\langle \text{load}, \{\text{src\_cell}, \text{src\_addr\_cell}\}, \{\text{dest\_cell}\} \rangle$$

$$\frac{\begin{gathered} a_{src} = T\left(A, M\left(src_{addr_{cell}}\right)\right), \\ Assert\left(\Delta\left(src_{addr_{cell}}\right) < a_{src}.base + a_{src}.slen\right) \end{gathered}}{\Delta \vdash \Delta[dest_{cell} \leftarrow src_{cell}]}$$

Load. Происходит проверка на выход за границы *L*-буфера при чтении данных из него. Описывает операции загрузки данных из памяти.

$$\langle \text{store}, \{\text{src\_cell}, \text{dest\_addr\_cell}\}, \{\text{dest\_cell}\} \rangle$$

$$\frac{\begin{gathered} a_{dst} = T\left(A, M\left(dest_{addr_{cell}}\right)\right) \\ Assert\left(\Delta\left(dest_{addr_{cell}}\right) < a_{dst}.base + a_{dst}.slen\right)' \end{gathered}}{\Delta \vdash \Delta[dest_{cell} \leftarrow src_{cell}]}$$

Store. Происходит проверка на выход за границы *L*-буфера при записи данных в него. Описывает операцию выгрузки данных в память.

$$\langle \text{jct}, \{\text{cond\_cell}, \text{addr\_cell}\}, \{\} \rangle$$

$$\frac{}{Pp \vdash Pp \cup (\Delta[\text{cond\_cell}] = \text{true})}$$

Jct (Jcf). Описывает операцию выполнения (не выполнения) условного перехода, добавляя уравнение в предикат пути.

$\langle \text{jcf}, \{\text{cond\_cell}, \text{addr\_cell}\}, \{\} \rangle$
$\dfrac{}{\text{Pp} \vdash \text{Pp} \cup (\Delta[\text{cond\_cell}] = \text{false})}$

3. Схема работы

В данном разделе описана схема работы алгоритма, реализующего представленный метод.

3.1 Используемые методы анализа

Алгоритм реализован в рамках среды динамического анализа бинарного кода [12]. Эта среда позволяет работать с трассами выполнения программ, полученными в результате работы полносистемного эмулятора. Трасса содержит последовательность выполненных инструкций процессора, а также значения регистров перед выполнением каждой инструкции. Анализ трасс выполнения позволяет анализировать поведение программы после того, как она была выполнена. Благодаря этому, алгоритмы анализа не замедляют работу анализируемых программ, что позволяет анализировать программы, работающие с сетью, и программы, противодействующие отладке.

Среда анализа бинарного кода предоставляет различные инструменты и структуры данных для высокоуровневого анализа: разметка трассы на процессы и потоки ОС, выделение вызовов функций, построение графа потока управлений, графа зависимостей по данным и многие другие. Для сокращения числа анализируемых инструкций используется алгоритм слайсинга трассы, основанный на анализе графа зависимостей по данным. Алгоритм слайсинга трассы отбирает те инструкции, которые имеют отношение к обработке или формированию заданного буфера данных в памяти. В современных процессорных архитектурах содержится множество инструкций со сложной семантикой и нетривиальными побочными эффектами. Для унификации обработки инструкций традиционно применяется метод, основанный на трансляции инструкций в промежуточное представление. В используемой среде анализа бинарного кода используется промежуточное представление Pivot [13], позволяющее единообразно описывать операционную семантику инструкций различных процессорных архитектур.

В процессе работы алгоритма выполняются проверки нарушений доступа к памяти с помощью SMT-решателя. В качестве SMT-решателя в системе анализа бинарного кода используется решатель Z3 [14].

3.2 Параметры алгоритма

Работа алгоритма начинается с некоторого шага трассы, заданного аналитиком. Также, для этого шага трассы задаётся буфер с входными данными. Для этого буфера создаётся абстрактный буфер с символьной длиной, при этом данные из входного буфера выступают в качестве префикса. Помимо этого, иногда конкретная длина входного буфера хранится отдельно от самих данных (например, в случае чтения данных с помощью функции recv). В этом случае, конкретная длина входного буфера связывается с символьной длиной соответствующего ему абстрактного буфера. Аналитик может указать расположение ячейки памяти, в которой хранится конкретная длина входного буфера. Если же длина буфера с входными данными задана неявно (например, в случае нуль-терминированной строки), эту ячейку памяти указывать не обязательно.

3.3 Трансляция инструкций

Начиная с заданного аналитиком шага трассы, начинается обработка инструкций процессора с учётом зависимостей по данным. Каждая инструкция сначала транслируется в промежуточное представление, а затем на основе этого промежуточного представления создаются формулы и уравнения для SMT-решателя. Уравнения, которые создаются во время трансляции инструкций, добавляются в предикат пути – множество уравнений, описывающих прохождение программы по некоторому пути выполнения. Перед началом трансляции множество выделенных буферов памяти пополняется набором буферов, которые уже выделены, но ещё не освобождены. Для отбора инструкций используется алгоритм, аналогичный алгоритму слайсинга трассы с некоторыми дополнениями. Главной особенностью работы алгоритма является пропуск трансляции отдельных функций. Многие библиотечные функции имеют известные побочные эффекты, которые можно описать явно с помощью уравнений над входными и выходными параметрами. Такой подход позволяет не транслировать инструкции, принадлежащие известной библиотечной функции, а вместо этого обновлять состояние контекста интерпретации в соответствии с описанием побочных эффектов для этой библиотечной функции. Это позволяет значительно сократить сложность уравнений, а также реализовать возможность работы с буферами символьной длины. Кроме того, использование слайсинга позволяет значительно сократить количество обрабатываемых инструкций процессора за счёт отбора только тех инструкций, которые связаны с обработкой помеченных данных. В табл. 1 приведено сравнение количества обрабатываемых инструкций. Во втором столбце приведены значения количества инструкций, которые были бы обработаны без использования слайсинга. В третьем столбце приведено количество инструкций, отобранных в результате работы слайсинга, а в

четвёртом столбце – количество инструкций, отобранных в результате работы слайсинга с использованием интерпретации функций с известной семантикой.

Табл. 1. Сравнение количества обрабатываемых инструкций.

Программа	Количество инструкций	Размер слайса	Количество оттранслированных инструкций
httpdx	712029	12576	12367
GoldMP4Player	22009330	9353	9347
mysql_plugin	220710	8268	105

На примере программы mysql_plugin видно, что с помощью интерпретации функций можно значительно сократить число обрабатываемых инструкций.

Во время обработки вызова библиотечной функции выполняются проверки различных условий нарушения доступа к памяти. Для этого составляются уравнения предиката безопасности. После этого предикат пути и предикат безопасности объединяются и передаются SMT-решателю. Если система уравнений оказалась совместной, решение системы уравнений будет содержать длину входного буфера, при которой происходит нарушение доступа к памяти. Семантика обработки некоторых библиотечных функций описана в разделе 2. Все остальные инструкции, отобранные алгоритмом слайсинга, обрабатываются с помощью механизма, который был описан в работе [9].

Следует отметить, что обрабатываемые вызовы библиотечных функций в трассе могут не соответствовать вызовам этих же функций в исходном коде анализируемой программы. Чаще всего несоответствие возникает из-за оптимизаций компилятора, встраивающих код функции в программу в месте вызова этой функции. В этом случае вызов функции в трассе не будет обработан с помощью описанной выше семантики, а вместо этого произойдёт обработка отдельных инструкций, которые соответствуют библиотечной функции. Это, в свою очередь, может привести к добавлению дополнительных ограничений на размер входных данных и ложноотрицательным результатам работы алгоритма.

Кроме вызовов библиотечных функций, специальным образом обрабатываются вызовы всех остальных функций. Если для вызова функции известна информация о размере кадра стека, множество выделенных буферов пополняется буфером, который описывает область памяти, соответствующую кадру стека. При обработке возврата из этой функции, буфер, соответствующий кадру стека, удаляется из множества выделенных буферов.

3.4 Завершение работы алгоритма

В процессе работы алгоритма проверяется нарушение доступа к памяти. Если при очередной проверке устанавливается факт нарушения, алгоритм

завершается. Результатом работы алгоритма является длина входного буфера, а также значение префикса.

4. Реализация дополнительных алгоритмов

В данном разделе описаны особенности реализации алгоритма разметки памяти и дополнение алгоритма выделения вызовов функций в рамках системы анализа бинарного кода.

4.1 Разметка памяти

В бинарном коде информация о переменных и буферах в памяти в явном виде отсутствует, поэтому для поиска выходов за границы буферов сначала нужно восстановить эту информацию. При реализации данного метода восстанавливалась информация о динамической и автоматической памяти.

Разметка динамической памяти. Составление карты динамической памяти основывается на использовании моделей функций [15]. Под моделью функции будем понимать функцию с описанными входными и выходными параметрами в виде ячеек памяти и регистров. Задаются три модели, каждая из которых соответствует функциям выделения (*alloc*), освобождения (*free*), и изменения размера уже выделенной памяти (*realloc*). Для каждой модели задаются параметры, имеющие определенную семантику. Для модели *alloc* задаётся размер (входной параметр) и адрес выделенной памяти (выходной параметр). Для модели *free* задаётся адрес освобождаемой памяти (входной параметр). Для модели *realloc* задаётся адрес буфера, размер которого будет изменён (входной параметр), новый размер (входной параметр) и адрес нового буфера (выходной параметр). После того, как модели заданы для каждого экземпляра в трассе, происходит обновление карты динамической памяти в соответствии с семантикой модели. Стоит отметить, что в исследуемой программе может быть несколько вложенных менеджеров памяти, для работы с которыми используются разные наборы функций. Для отличия областей выделенной памяти используется идентификатор менеджера памяти. Карта динамической памяти реализована в виде последовательности кортежей *{идентификатор менеджера памяти, шаг трассы при создании буфера, шаг трассы при удалении буфера, идентификатор процесса, идентификатор потока, адрес начала буфера, размер буфера}*.

Модель функции, связанная с конкретным вызовом этой функции в трассе, называется экземпляром модели этой функции.

Обработка экземпляра модели *alloc* добавляет кортеж в карту памяти, инициализируя все значения, кроме шага трассы на котором происходит удаление буфера.

Обработка экземпляра модели *free* добавляет в кортеж шаг трассы, на котором происходит удаление буфера.

Обработка экземпляра модели *realloc* является комбинацией обработки предыдущих двух моделей. Сначала записывается шаг трассы, на котором входной буфер удаляется, затем создаётся кортеж, описывающий новый буфер. Адрес и размер нового буфера соответствуют параметрам модели *realloc*.

С помощью обработки всех экземпляров моделей по описанным выше правилам составляется разметка для динамической памяти.

Разметка автоматической памяти. В случае с автоматической памятью буфером является кадр стека соответствующей функции заданного исполняемого модуля. Создание карты автоматической памяти происходит в два этапа:

- получение информации о кадрах стека для каждого исполняемого модуля при помощи IDA Pro;
- отображение полученной информации на трассу.

Информация, полученная с помощью IDA Pro, представляет собой последовательность кортежей вида:

- смещение адреса функции относительно базового адреса модуля;
- размер кадра стека;
- размер параметров функции расположенных на стеке.

Для каждого вызова функции в трасе из заданного модуля создаётся кортеж в карте, аналогичный кортежу в карте динамической памяти. Шагом создания является шаг вызова функции, а шагом удаления является шаг возврата из функции.

Совокупность разметок автоматической и динамической памяти используется при дальнейшем анализе.

4.2 Разметка вызовов в Linux

Поиск ошибок в программах под ОС Linux усложняется из-за использования механизмов ленивого связывания. Во время первого вызова каждой библиотечной функции вызывается функция-заглушка из библиотеки ld.so, которая получает адрес вызываемой функции и изменяет код исполняемого файла таким образом, что при следующем вызове этой же библиотечной функции она будет вызвана напрямую. При этом выход из функции-заглушки происходит с помощью инструкции RET и приводит к передаче управления на код вызываемой библиотечной функции. Фактически, вызов библиотечной функции в этом случае происходит с помощью инструкции RET, что приводит к искажению результатов работы алгоритмов, выполняющих поиск вызовов функций. Это, в свою очередь, приводит к тому, что первый вызов каждой библиотечной функции не обрабатывается модулем символьного анализа. В то же время, многие ошибки, связанные с переполнением буфера на стеке в результате обработки параметров командной строки, происходят в результате копирования параметра с помощью строковых функций (strcpy, strcat) в самом

начале программы. Это приводит к ложноотрицательному результату при поиске ошибок в таких программах.

Для решения данной проблемы необходим более детальный анализ кадров стека. Рассмотрим пример вызова функции waitpid из библиотеки libc-2.19.so. Последовательность инструкций изображена на Рис. 2. Функция waitpid вызывается из программы bash с помощью пары инструкций CALL и JMP по адресам 0807F0C1 и 08059700 соответственно. Так как это первый вызов функции waitpid, управление передаётся на заглушку по адресу 08059706 и далее в библиотеку ld-2.19.so. Выход из библиотеки происходит с помощью инструкции RET 000Ch по адресу B7FF243B.

```
        bash 0807F0C1   CALL  08059700h
        bash 08059700   JMP   DWORD PTR [080F51B8h]
        bash 08059706   PUSH  00000358h
        bash 0805970B   JMP   08059040h
        bash 08059040   PUSH  DWORD PTR [080F5004h]
        bash 08059046   JMP   DWORD PTR [080F5008h]
  ld-2.19.so B7FF2420   PUSH  EAX
  ld-2.19.so B7FF2421   PUSH  ECX
                        …
  ld-2.19.so B7FF2434   MOV   DWORD PTR SS:[ESP], EAX
  ld-2.19.so B7FF2437   MOV   EAX, DWORD PTR SS:[ESP + 04h]
  ld-2.19.so B7FF243B   RET   000Ch
libc-2.19.so B7E398B0   CMP   DWORD PTR GS:[0Ch], 0
libc-2.19.so B7E398B8   JNZ   0B7E398DCh
```

Рис. 2. Последовательность инструкций при вызове функции waitpid.

Так как функция-заглушка сохраняет указатель стека, значение указателя стека после выполнения инструкции CALL (по адресу 0807F0C1), после выполнения инструкции JMP (по адресу 08059700) и после выполнения инструкции RET (по адресу B7FF243B) одинаковое. Анализ цепочки вызовов и инструкций RET в комбинации с анализом значений указателя стека позволяет установить факт вызова функции-заглушки и корректно определить вызов библиотечной функции.

5. Результаты практического применения

Предложенный метод был реализован в виде модуля-расширения среды анализа бинарного кода, он использует такие ее возможности, как повышение уровня представления, модель процессора общего назначения, слайс трассы.

Разработанный инструмент был опробован на 11 программах, работающих под управлением 32-разрядных ОС Windows XP SP2 и Arch Linux (по состоянию на март 2015). Среди программ были 4 с опубликованными уязвимостями. Список анализируемых программ приведён в табл. 2. Для

каждой тестовой программы была получена трасса нормального, безошибочного, выполнения на некотором наборе входных данных. Далее был запущен алгоритм поиска ошибок, с целью проверки, сможет ли он построить входные данные, реализующие дефект.

В табл. 3 приведены результаты работы алгоритма. Для всех исследуемых программ время работы SMT-решателя не превышало одной секунды. При получении результатов было добавлено дополнительное ограничение сверху на абстрактную длину буфера. Без этого ограничения, SMT-решатель может подобрать слишком большое значение для абстрактной длины буфера, которое не позволит создать в памяти буфер такого размера.

Табл. 2. Список анализируемых программ.

Операционная система	Программа	Версия приложения	CVE / OSVDB
Linux	iwconfig	iwconfig v26	CVE: 2003-0947
Linux	get_driver	sysfsutils 2.1.0	–
Linux	mkfs.jfs	jfsutils 1.1.15	–
Linux	alsa_in	jack 0.124.1	–
Linux	OpenSSL	openssl 1.0.0f	CVE: 2014-0160 (Heartbleed)
Windows XP SP2	httpdx	httpdx 1.5.4	OSVDB-ID: 84454
Windows XP SP2	GoldMP4Player	GoldMP4Player 3.3	OSVDB-ID: 103826
Linux	faad	faad2 2.7	–
Linux	gencnval	icu 54.1	–
Linux	lou_checktable	liblouis 2.5.2	–
Linux	mysql_plugin	mariadb 10.0.17	–

Табл. 3. Результаты работы алгоритма.

ОС	Программа	Размер префикса, байт	Размер входных данных	Тип доступа к памяти	Память	Время работы, с.
Linux	iwconfig	32	93	запись	стек	3
Linux	get_driver	14	489	запись	стек	4
Linux	mkfs.jfs	31	436	запись	стек	2
Linux	alsa_in	34	355	запись	стек	4
Linux	OpenSSL	18	25	чтение	стек	5
WinXP SP2	httpdx	329	330	запись	куча	338

WinXP SP2	GoldMP4Player	36	505	запись	куча	255
Linux	faad	13	302	запись	стек	<1
Linux	gencnval	13	574	запись	стек	<1
Linux	lou_checktable	15	527	запись	стек	8
Linux	mysql_plugin	16	578	запись	стек	14

В двух программах удалось обнаружить переполнение буфера на куче. Для многих из исследуемых программ были обнаружены ошибки доступа к памяти, связанные с записью входных данных, размер которых можно контролировать, в буфер фиксированного размера. Однако, для некоторых программ (OpenSSL и httpdx) были обнаружены ошибки доступа к памяти другого характера.

На примере OpenSSL было продемонстрировано обнаружение уязвимости Heartbleed, которая заключается в чтении данных за границами выделенного буфера. Для этого примера алгоритм автоматически подбирает два значения размера: размер пакета с данными и значение поля внутри пакета, которое описывает размер передаваемых (и запрашиваемых) данных. В табл. 3 приведен размер пакета, а размер запрашиваемых данных входит в префикс и составляет 247 байт.

В программе httpdx пользователь может контролировать не размер входных данных, а размер буфера выделенной памяти. Программа получает HTTP-запрос и выделяет буфер с помощью функции malloc. Размер буфера берётся из значения поля Content-Length, содержащегося в HTTP-заголовке. Затем происходит копирование тела запроса в выделенный буфер. При достаточно маленьком размере буфера происходит его переполнение. На этом примере также демонстрируется возможность автоматического определения соответствия между строковым значением размера, содержащимся в поле Content-Length и численным значением, передаваемым в функцию malloc.

Для проверки работы алгоритма был проведен следующий эксперимент. Для каждой исследуемой программы был сгенерирован новый набор входных данных на основе префикса и размера входных данных, которые были получены в результате работы описанного алгоритма. Если размер входных данных был больше размера префикса, оставшиеся байты, следующие за префиксом, принимались равными значению 0x41. Для всех исследуемых программ полученные наборы входных данных привели к аварийному завершению, что указывает на отсутствие ложноположительных результатов на данном наборе исследуемых программ. Следует отметить, что данный способ дополнения входных данных может привести к ложноположительному результату в случае, когда исследуемая программа проверяет формат входных данных.

6. Обзор близких работ

Наиболее близкие результаты были показаны в работах Splat [11] и LESE [16]. Инструмент Splat позволяет автоматически анализировать исходный код на языке Си и генерировать входные данные, приводящие к нарушению доступа к памяти. В данном инструменте используется понятие абстрактной длины и символьная интерпретация некоторых функций. Основным ограничением инструмента является то, что он предназначен только для исходных текстов программ на языке Си, в отличие от предлагаемого метода, который анализирует бинарный код. В работе LESE описывается метод обработки циклов, позволяющий проанализировать поведение программы при выполнении произвольного числа итераций цикла. В данном подходе считается, что входные данные определяются известной грамматикой. Для каждого цикла заводится специальная символьная переменная-счётчик *(trip counter*), которая отвечает за количество итераций в этом цикле. Все индуктивные переменные циклов выражаются через такие счётчики. Также счётчики связываются с атрибутами входных данных: длина поля или количество итераций поля-счётчика. В отличие от Splat, подход, описанный в LESE, применим также к программам, распространяемым в виде бинарного кода. К сожалению, описанные в данных работах инструменты недоступны.

Также стоит отметить отдельный класс программных инструментов, основанных на символьном исполнении, которые реализуют управляемый фаззинг для поиска дефектов: BitFuzz [17], FuzzBall [18], SAGE [19], Avalanche [20].

7. Заключение

В статье представлен метод поиска дефектов, приводящих к нарушению доступа к памяти. Метод основан на символьной интерпретации бинарной трассы, он позволяет абстрагироваться от конкретной длины входных данных и за счёт этого вычислять длину входных данных, при которой проявляется дефект. Метод был реализован в виде программного инструмента, являющегося частью среды анализа бинарного кода.

Входными данными для анализа выступает набор трасс, обеспечивающий достаточное покрытие кода. Анализ набора трасс производится автоматически, что позволяет совмещать его с другой деятельностью, например, восстановлением алгоритма из бинарного кода [3, 12].

Для последующей оценки критичности найденных ошибок следует воспользоваться методом, описанным в работе [9]. Входные данные, реализующие дефект, используются для получения новой трассы. Ее анализ позволяет оценить возможности эксплуатации найденного дефекта.

Дальнейшие работы предполагают автоматизацию определения точек получения входных данных и поддержку более широкого класса библиотечных функций.

Список литературы

[1]. Common Weakness Enumeration, a community-developed dictionary of software weakness types. https://cwe.mitre.org Дата обращения: 8.04.2015

[2]. К. Батузов, П. Довгалюк, В. Кошелев, В. Падарян. Два способа организации механизма полносистемного детерминированного воспроизведения в симуляторе QEMU. // Труды Института системного программирования РАН, том 22, 2012 г. Стр. 77-94.

[3]. Андрей Тихонов, Арутюн Аветисян, Вартан Падарян. Методика извлечения алгоритма из бинарного кода на основе динамического анализа. // Проблемы информационной безопасности. Компьютерные системы. №3, 2008. Стр. 66-71

[4]. Андрей Тихонов, Вартан Падарян. Применение программного слайсинга для анализа бинарного кода, представленного трассами выполнения. // Материалы XVIII Общероссийской научно-технической конференции «Методы и технические средства обеспечения безопасности информации». 2009. стр. 131

[5]. А.Ю.Тихонов, А.И. Аветисян. Комбинированный (статический и динамический) анализ бинарного кода. // Труды Института системного программирования РАН,том 22, 2012 г. стр. 131-152.

[6]. Alexander Getman, Vartan Padaryan, and Mikhail Solovyev. Combined approach to solving problems in binary code analysis. // Proceedings of 9th International Conference on Computer Science and Information Technologies (CSIT'2013), pp. 295-297.

[7]. Довгалюк П.М., Макаров В.А., Романеев М.С., Фурсова Н.И. Применение программых эмуляторов в задачах анализа бинарного кода. // Труды Института системного программирования РАН, том 26, 2014 г. Выпуск 1. Стр. 277-296. DOI: 10.15514/ISPRAS-2014-26(1)-9.

[8]. King J.C. Symbolic execution and program testing. // Commun. ACM. – 1976. – No 19.

[9]. Падарян В.А., Каушан В.В., Федотов А.Н. Автоматизированный метод построения эксплойтов для уязвимости переполнения буфера на стеке. // Труды Института системного программирования РАН, том 26, 2014 г. Выпуск 3. Стр. 127-144. DOI: 10.15514/ISPRAS-2014-26(3)-7.

[10]. E. J. Schwartz, T. Avgerinos, D. Brumley. // All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). // IEEE Symposium on Security and Privacy, May 2010, pp. 317–331.

[11]. Ru-Gang Xu, Patrice Godefroid, Rupak Majumdar. // Testing for Buffer OverFlows with Length Abstraction. // ISSTA, 2008

[12]. В.А. Падарян, А.И. Гетьман, М.А. Соловьев, М.Г.Бакулин, А.И. Борзилов, В.В. Каушан, И.Н. Ледовских, Ю.В. Маркин, С.С. Панасенко. // Методы и программные средства, поддерживающие комбинированный анализ бинарного кода. // Труды Института системного программирования РАН, том 26, 2014 г. Выпуск 1. Стр. 251-276. DOI: 10.15514/ISPRAS-2014-26(1)-8.

[13]. Падарян В. А., Соловьев М. А., Кононов А. И. // Моделирование операционной семантики машинных инструкций. // Программирование, No 3, 2011 г. Стр. 50-64.

[14]. Nikolaj Bjørner, Leonardo de Moura. // Z3: Applications, Enablers, Challenges and Directions/ // Sixth International Workshop on Constraints in Formal Verification Grenoble, 2009.

[15]. А. И. Аветисян, А. И. Гетьман. // Восстановление структуры бинарных данных по трассам программ. // Труды Института системного программирования РАН, том 22, 2012 г. Стр. 95-118.

[16]. Prateek Saxena, Pongsin Poosankam, Stephen McCamant, Dawn Song. // Loop-Extended Symbolic Execution on Binary Programs. // ISSTA, 2009.

[17]. J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song. Input generation via decomposition and re-stitching: Finding bugs in malware. In Proc. of the ACM Conference on Computer and Communications Security, Chicago, IL, October 2010.

[18]. L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. // In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems, London, UK, Mar. 2012.

[19]. P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. // In Proc. of the Network and Distributed System Security Symposium, Feb. 2008.

[20]. Исаев, И. К., Сидоров, Д. В., Герасимов, А. Ю., Ермаков, М. К. (2011). Avalanche: Применение динамического анализа для автоматического обнаружения ошибок в программах использующих сетевые сокеты. Труды Института системного программирования РАН, том 21, 2011 г., стр. 55-70.

Memory Violation Detection Method in Binary Code

V. V. Kaushan <korpse@ispras.ru>
A. YU. Mamontov <mamontov@ispras.ru>
V. A. Padaryan <vartan@ispras.ru>
A. N. Fedotov <fedotoff@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004

Abstract. In this paper memory violation detection method is considered. This method is applied to program binaries without requiring debug information. It allows to find such memory violations as out-of-bound read or write. The technique is based on dynamic analysis and symbolic execution. Instead of representing input buffer as a symbolic variable of fixed size, we track only the prefix of buffer symbolically and a special symbolic variable that represents the length of input buffer. The symbolic length variable allows to interpret functions with known semantics such as string library or memory allocation functions. While interpreting these functions using symbolic length variables we assert some constraints on buffer bounds. Such constraints allow to find memory violations. If violation is located, concrete values of buffer prefix and final input buffer length are provided. To apply this method to binary code we have to recover buffer bounds. So we developed some methods that recover buffer bounds in heap and stack memory. We present a tool implementing the method. We used this tool to find 11 bugs in both Linux and Windows programs, 7 of which were undocumented at the time this paper was written. This tool was able to detect known Heartbleed vulnerability which couldn't be found by simple fuzzers in crash absence.

Keywords: bug finding; symbolic execution; binary code; dynamic analysis.

DOI: 10.15514/ISPRAS-2015-27(2)-7

For citation: Kaushan V. V., Mamontov A. YU., Padaryan V. A., Fedotov A. N. Memory Violation Detection Method in Binary Code. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 105-126 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-7.

References

[1]. Common Weakness Enumeration, a community-developed dictionary of software weakness types. https://cwe.mitre.org Date of treatment: 8.04.2015

[2]. K. Batuzov, P. Dovgalyuk, V. Koshelev, V. Padaryan. Dva sposoba organizatsii mehanizma polnosistemnogo determinirovannogo vosproizvedeniya v simulyatore QEMU.[Two methods full-system deterministic replay in QEMU]// *Trudy ISP RAN [The Proceedings of ISP RAS]*, vol. 22, 2012, pp. 77-94 (in Russian)

[3]. Tikhonov A.Yu., Avetisyan A.I., Padaryan V.A., Metodika izvlecheniya algoritma iz binarnogo koda na osnove dinamicheskogo analiza [Methodology of exploring of an algorithm from binary code by dynamic analysis]. *Problemy informatsionnoj bezopasnosti. Komp'yuternye sistemy [Informations security aspects. Computer sistems]*, 2008, №3. pp. 66-71 (in Russian)

[4]. Tikhonov A.Yu., Padaryan V.A., Primenenie programmnogo slaysinga dlya analiza binarnogo koda, predstavlennogo trassami vyipolneniya.[Using program slicing for bynary code represented by execution traces] *Materialyi XVIII Obscherossiyskoy nauchno-tehnicheskoy konferentsii «Metodyi i tehnicheskie sredstva obespecheniya bezopasnosti informatsii». [The Proceedings of XVIII Russian science technical conference "Methods and technical information security tools"]* 2009. pp 131 (In Russian).

[5]. Tikhonov A.Yu., Avetisyan A.I. Kombinirovannyj (staticheskij i dinamicheskij) analiz binarnogo koda. [Combined (static and dynamic) analysis of binary code]. *Trudy ISP RAN [The Proceedings of ISP RAS]*, vol. 22, 2012, pp. 131-152 (in Russian).

[6]. Alexander Getman, Vartan Padaryan, and Mikhail Solovyev. Combined approach to solving problems in binary code analysis. // Proceedings of 9th International Conference on Computer Science and Information Technologies (CSIT'2013), pp. 295-297.

[7]. Dovgalyuk P.M., Makarov V.A., Romaneev M.S., Fursova N.I. Primenenie programmyih emulyatorov v zadachah analiza binarnogo koda.[Applying program emulators for binary code analysis] // *Trudy ISP RAN [The Proceedings of ISP RAS]*, vol. 26, issue 2014, pp. 277-296. DOI: 10.15514/ISPRAS-2014-26(1)-9.

[8]. King J.C. Symbolic execution and program testing. // Commun. ACM. – 1976. – No 19.

[9]. Padaryan V.A., Kaushan V.V., Fedotov A.N. Avtomatizirovannyiy metod postroeniya eksploytov dlya uyazvimosti perepolneniya bufera na steke.[Automated exploit generaton method for stack buffer overflow vulnerabilities] // *Trudy ISP RAN [The Proceedings of ISP RAS]*, vol. 26, issue 3, 2014, pp.. 127-144. DOI: 10.15514/ISPRAS-2014-26(3)-7

[10]. E. J. Schwartz, T. Avgerinos, D. Brumley. // All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). // IEEE Symposium on Security and Privacy, May 2010, pp. 317–331.

[11]. Ru-Gang Xu, Patrice Godefroid, Rupak Majumdar. // Testing for Buffer OverFlows with Length Abstraction. // ISSTA, 2008

[12]. V.A. Padaryan, A.I. Getman, M.A. Solovyev, M.G. Bakulin, A.I. Borzilov, V.V. Kaushan, I.N. Ledovskich, U.V. Markin, S.S. Panasenko. Metody i programmnye sredstva, podderzhivayushhie kombinirovannyj analiz binarnogo koda [Methods and software tools for combined binary code analysis]. *Trudy ISP RAN [The Proceedings of ISP RAS]*, 2014, vol. 26, no. 1, pp. 251-276 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-8

[13]. Padaryan V.A., Solov'ev M.A., Kononov A.I. Modelirovanie operatsionnoy semantiki mashinnyih instruktsiy. [Simulation of operational semantics of machine instructions]. *Programming and Computer Software*, May 2011, Volume 37, Issue 3, pp 161 – 170 , DOI 10.1134/S0361768811030030 (In Russian)

[14]. Nikolaj Bjørner, Leonardo de Moura. // Z3: Applications, Enablers, Challenges and Directions/ // Sixth International Workshop on Constraints in Formal Verification Grenoble, 2009.

[15]. Avetisyan A.I., Getman A.I. Vosstanovlenie struktury binarnykh dannykh po trassam program [Recovery the structure of binary data on the program traces]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 95-118 (in Russian)

[16]. Prateek Saxena, Pongsin Poosankam, Stephen McCamant, Dawn Song. // Loop-Extended Symbolic Execution on Binary Programs. // ISSTA, 2009.

[17]. J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song. Input generation via decomposition and re-stitching: Finding bugs in malware. In Proc. of the ACM Conference on Computer and Communications Security, Chicago, IL, October 2010.

[18]. L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. // In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems, London, UK, Mar. 2012.

[19]. P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. // In Proc. of the Network and Distributed System Security Symposium, Feb. 2008.

[20]. Isaev, I. K., Sidorov, D. V., Gerasimov, A. YU., Ermakov, M. K. (2011). Primenenie dinamicheskogo analiza dlya avtomaticheskogo obnaruzheniya oshibok v programmakh ispol'zuyushhikh setevye sokety [Using dynamic analysis for automatic bug detection in software that use network sockets]. *Trudy ISP RAN [The Proceedings of ISP RAS]*, 2011, vol. 21, pp. 55-70 (In Russian).

Методы повышения производительности обратной отладки

М.А. Климушенкова <maria.klimushenkova@ispras.ru>
П.М. Довгалюк <pavel.dovgaluk@ispras.ru>
Новгородский государственный университет имени Ярослава Мудрого
173003, Россия, г. Великий Новгород, ул. Большая Санкт-Петербургская, д. 41

Аннотация. Обратная отладка — это инструмент разработки ПО, позволяющий более эффективно справляться с ошибками, возникающими при недетерминированном поведении программы. Она позволяет изучать прошедшие состояния программы без ее повторного запуска. В работе описана реализация обратной отладки на основе детерминированного воспроизведения в симуляторе QEMU 2.0. Предлагаются несколько способов повышения производительности отладки за счет сокращения дополнительно записываемых данных, оптимального сохранения снимков системы, индексации и сжатия журнала событий. Симулятор может работать совместно с интерактивным отладчиком GDB, что позволяет использовать команды reverse-continue, reverse-nexti, reverse-stepi и reverse-finish в процессе отладки. Скорость работы этих команд зависит от периода сохранения состояний системы в процессе записи ее работы. В статье представлена оценка наилучшего периода для оптимальной скорости работы команды reverse-continue.

Ключевые слова: обратная отладка; детерминированное воспроизведение; QEMU; симулятор

DOI: 10.15514/ISPRAS-2015-27(2)-8

Для цитирования: Климушенкова М.А., Довгалюк П.М. Методы повышения производительности обратной отладки. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 127-144. DOI: 10.15514/ISPRAS-2015-27(2)-8.

1. Введение

Неотъемлемым этапом разработки каждой программы является отладка. Процесс отладки может оказаться очень трудоемким и занять непредсказуемое количество времени. Есть много инструментов, которые

помогают и облегчают программисту этот процесс, но сама процедура отладки, как правило, одина и та же.

Каждый раз поиск ошибки начинается с выполнения программы до момента ее первого проявления, после этого программист идет в обратном направлении по пути выполнения до причины ее возникновения. Классическим способом является циклическая отладка. Программист перезапускает программу, останавливает ее в более раннем моменте времени и изучает ее состояние с помощью отладчика и трассировки, так происходит несколько раз. Этот подход к отладке сложно использовать, если программа недетерминирована.

Основными трудностями являются:

- Недетерминированное поведение программы. Возникновение таких недетерминированных событий, как прерывания, переключения потоков, ввод от пользователя или получение данных из сети изменяет ход выполнения программы
- Программа работает длительное время до возникновения ошибки (часы, дни, месяцы)
- Сам факт отладки может повлиять на работу программы. При остановке или выполнении программы по шагам обмен сетевыми пакетами может быть нарушен из-за истечения таймаута соединения
- Непосредственное взаимодействие с аппаратной частью. Аппаратные устройства являются источниками недетерминизма, и алгоритм их работы может быть нарушен из-за задержек при отладке

Решение проблем, вызванных описанными трудностями, может быть достигнуто посредством использования механизма обратной отладки. Она позволяет программисту изучать уже прошедшие состояния программы без повторного ее выполнения с самого начала. Обратная отладка может быть реализована с помощью трассировки или детерминированного воспроизведения. В случае трассировки доступна только та информация о работе системы, которая была записана в трассу, для получения дополнительных данных необходимо повторно выполнить программу и записать новую трассу. В случае же детерминированного воспроизведения можно восстановить состояние всей системы и получить любую нужную информацию. По этой причине детерминированное воспроизведение является более предпочтительным способом реализации обратной отладки [1].

Для создания интерактивного обратного отладчика необходимо реализовать детерминированное воспроизведение работы отлаживаемой системы, механизм перехода на произвольное количество инструкций вперед и назад и возможность исследования состояния системы в заданный момент времени.

2. Детерминированное воспроизведение

2.1 Средства детерминированного воспроизведения

Путь выполнения любой программы зависит не только от передаваемых пользователем входных данных и окружения, в котором она запускается, но и от большого количества прочих факторов, которые сложно контролировать. Например, прерывания аппаратных устройств, моменты прихода сетевых пакетов и их содержимое, работа менеджера памяти системы и прочее. Во время отладки необходимо повторить тот же путь выполнения, при котором возникла ошибка. Сделать это вручную для сложных приложений реального времени крайне трудно. С помощью симулятора с поддержкой детерминированного воспроизведения можно повторить тот же самый путь [2]. Симулятор позволяет запустить внутри себя исследуемое приложение, которое будет функционировать так же, как если бы оно работало в реальной системе.

На рынке существует ряд программных симуляторов с функцией детерминированного воспроизведения. В своей основе все симуляторы используют механизмы виртуализации. Она может быть реализована как на аппаратном, так и на программном уровне.

Аппаратная виртуализация требует от хостовой платформы поддержки специальной процессорной архитектуры. При этом не требуется совпадения архитектур хостовой и гостевой систем, но требуется их совместимость. Например, возможна реализация 32-битных гостевых систем на 64-битных хостовых системах. Использование аппаратной виртуализации позволяет достичь высокой производительности, близкой к производительности невиртуализованной системы. Она используется в виртуальных машинах ReVirt [3], XenLR [4] и VMWare версии 7.0 [5], которые поддерживают детерминированное воспроизведение.

В части программной виртуализации можно выделить несколько основных подходов: паравиртуализация, динамическая трансляция и интерпретация. Техника паравиртуализации подразумевает внесение изменений в гостевую ОС, это возможно только, если ОС имеет открытый исходный код. При этом если необходимо эмулировать много аппаратных устройств, то количество правок будет велико. Паравиртуализация используется таким инструментом обратной отладки как TTVM[6], созданным на основе симулятора User-Mode Linux [7].

Динамическая трансляция представляет собой трансляцию машинного кода гостевой архитектуры в код архитектуры хостовой машины, при этом обе архитектуры никак между собой не связаны. Этот механизм реализован в симуляторе QEMU[8], который поддерживает широкий спектр аппаратных платформ. На основе QEMU был создан проект FREE [9] – еще один вариант детерминированного воспроизведения работы системы. Эта реализация поддерживает только гостевую архитектуру x86.

В основе полносистемного симулятора Bochs[10] лежит интерпретация гостевого кода, она медленнее динамической трансляции, однако более точно эмулирует поведение гостевой системы. На базе Bochs создан инструмент воспроизведения ExecRecoder [11], но сам симулятор эмулирует только архитектуру x86.

Чем меньше ограничений накладывает симулятор на возможную конфигурацию системы, тем шире спектр приложений, для которых можно применить обратную отладку, основанную на нем. Аппаратная виртуализация и паравиртуализация ограничивают архитектуру и ОС гостевой системы, что делает невозможным использование их, к примеру, при отладке мобильных приложений на настольных компьютерах. Динамическая трансляция и интерпретация позволяют выполнять код одной архитектуры на машине с другой архитектурой, то есть можно реализовать поддержку любой аппаратной платформы.

2.2 Корректность и производительность воспроизведения

Скорость работы системы внутри симулятора сравнима со скоростью ее работы на физической машине, поэтому симулятор, как правило, никак не влияет на ход выполнения приложения внутри него. Однако сохранение дополнительных данных, необходимых для воспроизведения, может существенно замедлить работу симулятора. Это будет критичным при отладке приложений, работающих в реальном времени, также может быть нарушено взаимодействие с сетью. Сохранение дополнительных данных включает в себя запись журнала недетерминированных событий и периодическое создание снимков системы, которые позднее используются при отладке. Журнал представляет собой файл, в который записываются недетерминированные события, происходящие в симуляторе [2]. Чем меньше информации будет сохраняться, тем меньше будет замедление. Для этого в журнал нужно записывать минимально возможное количество событий, в идеале только события от периферийных устройств, а снимки системы нужно получать уже во время воспроизведения ее работы, когда вызванное этим замедление ни на что не будет влиять.

В процессе воспроизведения порядок выполняющихся инструкций и состояние регистров процессора должны соответствовать порядку инструкций и состоянию регистров при выполнении программы в симуляторе во время записи ее работы.

3. Обратная отладка

3.1 Реализация функций отладчика

Для реализации функций отладки механизм детерминированного воспроизведения должен поддерживать совместную работу с интерактивным

отладчиком. Одним из наиболее подходящих является отладчик GDB [12], он доступен для большинства популярных платформ (Linux, Windows) и целевых архитектур, таких как Intel86, AMD64 и ARM.

Стандартные возможности отладчика позволяют останавливать программу в ходе ее выполнения, двигаться по пути выполнения и изучать состояние программы [13]. Большинство отладчиков могут двигаться только в прямом направлении выполнения. Однако, часто обратное направление движения будет более удобным для пользователя, чтобы найти причину ошибки.

Ядро отладчика GDB содержит поддержку таких функций обратной отладки, как reverse-continue, reverse-nexti, reverse-stepi и reverse-finish наравне с классическими командами continue, next, step, finish.

По команде step отладчик переходит к следующей инструкции, при необходимости заходя внутрь вызовов функций. Подобно ей, команда reverse-stepi переходит на предшествующую инструкцию, возможно заходя внутрь предыдущего вызова функции. Для работы этих команд при обратной отладке используется внутренний счетчик инструкций. При переходе вперед, процессор выполняет одну очередную инструкцию, а при переходе назад происходит загрузка предыдущего снимка системы, а затем воспроизведение кода до инструкции с порядковым номером на один меньше текущего.

Если же для программы доступен исходный код, а не только исполняемый файл, то можно выполнить команду reverse-step, которая переходит не к предыдущей инструкции, а к предыдущей строке кода.

Команда continue выполняет программу в прямом направлении до достижения какой-либо точки останова, аналогично ей, команда reverse-continue делает то же самое, только в обратном направлении. Традиционный отладчик вставляет в исполняемый код в точках останова программы инструкции-ловушки, которые передают управление отладчику при их срабатывании.

В случае обратной отладки симулятор не идет в обратном направлении исполнения кода в прямом смысле. По команде reverse-continue выполнение должно останавливаться в ближайшей предшествующей точке останова, поэтому симулятор загружает предыдущий снимок состояния системы и воспроизводит весь путь выполнения до текущего момента, чтобы найти все точки останова и определить последнюю. Далее он повторно загружает снимок и воспроизводит работу системы до уже известной точки. Если ни одной точки останова не нашлось, то весь процесс повторяется для следующего более раннего снимка.

Команда reverse-nexti аналогично команде reverse-stepi переходит на предыдущую инструкцию, однако в отличии от нее, не заходит внутрь вызовов функций. Если предыдущая инструкция являлась вызовом функции, то произойдет переход к состоянию предшествующему этому вызову, иначе команда работает также как reverse-stepi. По команде reverse-finish отладчик переходит к месту вызова функции, в которой он сейчас находится.

Все точки останова работают и при обратном направлении отладки, что позволяет, например, перейти к предыдущей точке, где была изменена переменная.

3.2 Скорость перехода на определенный шаг

Отладка программы часто происходит итеративно, практически невозможно с первого раза определить место, которое являлось причиной возникновения ошибки. В случае обратной отладки на это требуется много шагов назад. Если каждый из них будет приводить к длительному ожиданию, то весь процесс займет слишком много времени. Скорость перехода к шагу «в прошлом» зависит от удаленности ближайшего предшествующего снимка системы и времени на воспроизведение от этого снимка до интересующего шага.

Переход к предыдущему шагу во время обратной отладки происходит путем загрузки состояния системы, предшествующего интересующему шагу, с дальнейшим воспроизведением до нужного шага. От количества сохраненных состояний зависит скорость, с которой будет происходить переход назад, так как чем меньше интервал времени между ними, тем меньше инструкций нужно будет воспроизводить до искомого шага.

Простейший способ сохранения состояния виртуальной машины – это создание полной копии физической памяти, образа виртуального диска и состояния процессора и периферийных устройств. Однако этот способ является неэффективным по времени и по объему использующейся памяти. Решением этой проблемы может являться использование механизма копирования при записи, подобного применяемому в симуляторе TTVM [6]. Это позволяет существенно сократить накладные расходы на создание снимка виртуальной машины. Когда состояние сохраняется впервые, записывается все содержимое физической памяти, а для следующих состояний сохраняются только изменившиеся страницы. Они записываются в два лог-файла. Первый используется для получения «следующих состояний», то есть хранит изменения состояния n+1 относительно состояния n. Второй используется для получения «предыдущих состояний», хранит изменения состояния n-1 относительно состояния n. Образ виртуального диска хранится аналогичным образом, сохраняются только изменившиеся блоки.

Дополнительные состояния можно добавлять и удалять путем внесения правок в лог-файлы. Если журнал записывается в течение длительного времени, состояний накапливается очень много, так как изначально они сохраняются с фиксированным интервалом. Для экономии дискового пространства некоторые из них нужно постепенно удалять. По умолчанию, чем ближе период к моменту возникновения ошибки, тем больше состояний для него оставляется, так как он более интересен для отладки. Состояния для удаления выбираются таким образом, чтобы расстояние по времени между соседними возрастало экспоненциально в соответствии с удаленностью по времени от момента возникновения ошибки. Для ускорения переходов,

программист может определить более короткий интервал при сохранении состояний для выбранной части журнала, которую он отлаживает чаще всего [6].

Если состояния сохранялись редко, то скорость перехода к предыдущим шагам будет сильно зависеть от скорости воспроизведения. На нее влияют количество записываемых в журнал событий, накладные расходы от реализации механизма воспроизведения внутри симулятора и насколько полно воспроизводится работа всей системы (воспроизводится работа только процессора, воспроизводится работа периферийных, выводится ли изображение от видеокарты в окно симулятора).

Средняя скорость работы команды reverse-continue зависит от величины интервала сохранения снимков системы [14]. При слишком маленьком интервале негативное влияние начинают оказывать накладные расходы от загрузки снимка системы, а при слишком большом - происходит воспроизведение большого участка журнала и теряется преимущество от использования снимков. Оптимальный интервал можно найти эмпирическим путем для конкретной реализации детерминированного воспроизведения и конкретной отлаживаемой программы.

Время, необходимое для работы команде reverse-continue, зависит от количества инструкций до ближайшей точки останова. Симулятор загружает предыдущий снимок системы и воспроизводит журнал с этого состояния до текущего положения в поиске точки останова, если она не найдена, то загружает следующий снимок и снова воспроизводит журнал, процесс повторяется до тех пор, пока не будет найдена точка останова или до начала выполнения программы. Если ближайшая точка останова находится на расстоянии в m инструкций относительно текущей точки, то для ее поиска нужно будет загрузить d снимков системы и воспроизвести столько же участков журнала между соседними снимками состояния.

$$d = \left\lceil \frac{m}{n} \times k \right\rceil, \quad (1)$$

где m - количество инструкций от ближайшей точки останова до текущей точки;

n - количество инструкций от начала журнала на текущей точки;

k - количество сохраненных снимков с начала записи журнала до текущего момента;

Время для загрузки одного снимка и воспроизведение журнала до следующего снимка можно выразить формулой (2).

$$T_d = T_k + t_s, \quad (2)$$

где T_k - время воспроизведения журнала между двумя соседними снимками состояния;

t_s - время загрузки состояния;

$$T_k = \frac{n}{k} \times t_i \,, \tag{3}$$

где t_i - время воспроизведения одной инструкции;

Тогда общее время нахождения ближайшей точки останова можно выразить формулой (4).

$$T_m = d \times T_d = \left\lceil \frac{m}{n} \times k \right\rceil \times (T_k + t_s) = \left\lceil \frac{m}{n} \times k \right\rceil \times \left(\frac{n}{k} \times t_i + t_s \right) \tag{4}$$

После того, как точка найдена, происходит воспроизведение журнала с загруженного состояния до состояния срабатывания точки останова. Требующееся на это время можно посчитать по формуле (5).

$$T_r = \left(\frac{n}{k} \times d - m \right) \times t_i \tag{5}$$

В результате с учетом формул (4) и (5) время выполнения команды reverse-continue равно выражается формулой (6)

$$\begin{aligned} T_m = d \times T_d + T_r &= d \times (T_k + t_s) + \left(\frac{n}{k} \times d - m \right) \times t_i \\ &= d \times (T_k + t_s) + \frac{n}{k} \times d \times t_i - m \times t_i = \\ &= d \times \left(T_k + t_s + \frac{n}{k} \times t_i \right) - m \times t_i \\ &= \left\lceil \frac{m}{n} \times k \right\rceil \times \left(\frac{n}{k} \times t_i + t_s + \frac{n}{k} \times t_i \right) - m \times t_i \\ &= \left\lceil \frac{m}{n} \times k \right\rceil \times \left(2 \times \frac{n}{k} \times t_i + t_s \right) - m \times t_i \end{aligned} \tag{6}$$

Для каждой реализации из формулы (6) можно вычислить оптимальный период сохранения снимков системы.

3.3 Модификация журнала для повышения производительности

Как правило, при отладке удается локализовать проблему, и далее работа идет ни со всем сценарием работы программы, а только с его частью. Поэтому удобно выделить фрагмент сценария, и работать только с ним.

При выполнении команды reverse-continue воспроизведение программы происходит в обратном порядке до первой встретившейся точки останова. Заранее неизвестно когда последний раз выполнялась искомая инструкция или менялась искомая ячейка памяти, поэтому симулятор будет последовательно в обратном порядке загружать сохраненные состояния и выполнять код между ними, чтобы найти место, где ему следует остановиться.

Значительно ускорить этот процесс позволяет индексация выполненного кода. В процессе записи журнала дополнительно сохраняется информация о том, когда какая инструкция была выполнена и какие элементы памяти записала [15]. Весь процесс записи журнала разбивается на небольшие временные промежутки. Для поиска момента, когда был изменен регистр или выполнена

инструкция, используются промежутки, соответствующие выполнившимся базовым блокам машинных команд. Запись о каждом промежутке содержит диапазон адресов выполнившихся инструкций и битовую маску изменившихся в этом блоке регистров. Для поиска изменений в ячейке памяти, вся память разбивается на страницы, для каждой страницы сохраняется своя история из временных промежутков, записи в которой содержат битовую маску измененных ячеек. Когда нужно найти место последнего изменения переменной, вся история проходится в обратном порядке, проверяя битовые маски, а затем загружается нужное состояние.

Если до возникновения ошибки программа работает очень длительное время, то размер журнала станет достаточно большим и может достигать несколько сотен гигабайт. Уменьшить размер файла журнала можно с помощью его сжатия. При воспроизведении его части будут последовательно восстанавливаться. Другим способом сокращения размера журнала является выделение из него фрагмента, относящегося к исследуемой ошибке. Сохранение снимков системы позволяет получить из журнала фрагмент, начинающийся с произвольного снимка, и использовать его в качестве полноценного журнала.

4. Исследование гостевой системы

Использование полносистемного симулятора в качестве основы для реализации обратной отладки имеет как преимущества, такие как возможность отладки компонентов операционной системы и драйверов, так и недостатки. Симулятор выполняет код не только исследуемого приложения, а всей системы целиком. Если приложение собрано с отладочной информацией и доступен его исходный код, то можно указать отладчику путь к исполняемому и исходным файлам. В этом случае выполнять команды отладки и ставить точки останова можно в исходном коде. Если же эта информация отсутствует, то отлаживать приходится ассемблерный код, что более трудоемко.

Если ошибка возникает в каком-то компоненте операционной системы, для которого нет исходного кода, то получить дополнительную информацию о системы и ее объектах можно с помощью анализа дампа памяти и таких инструментов, как интерактивные отладчики. Симулятор с функцией детерминированного воспроизведения позволяет перейти в любое место выполнения программы, а затем с помощью дополнительных инструментов можно исследовать систему.

Дамп оперативной памяти, полученный в момент выполнения программы в симуляторе, можно исследовать с помощью инструмента Volatility Framework [16]. Он включает в себя широкий спектр плагинов для получения списка активных процессов и загруженных библиотек, соответствия физических и

виртуальных адресов, списка модулей/драйверов/потоков ядра и многого другого.

Если гостевой системой является Windows, то для отладки можно использовать отладчик WinDBG [17]. Он позволяет отлаживать пользовательские приложения и драйвера, анализировать аварийные дампы, автоматически загружать символьную информацию для модулей Windows. Поддерживается возможность отладки удаленной машины, для этого в отлаживаемой системе запускается отладочный сервер. Он работает внутри системы, поэтому при воспроизведении интерактивная отладка становится невозможной.

В SDK платформы Android используется симулятор QEMU, на его базе можно реализовать обратную отладку мобильных приложений. Большинство приложений реализованы на языке java, для их выполнения в систему входит виртуальная машина Dalvik. При отладке с использованием GDB симулятор выполняет код этой виртуальной машины, а не java код, что существенно осложняет поиск ошибки. Для отладки java кода в SDK включен отладчик DDMS [18]. Он подключается к порту симулятора или реального устройства и позволяет получить информацию о выполняющихся потоках, процессах и динамических объектах. Отладчик интерактивно взаимодействует с симулятором, посылая ему команды и получая запрашиваемую информацию, поэтому при воспроизведении получение информации, отличной от полученной при записи, невозможно. Воспользоваться DDMS при воспроизведении можно, если остановить симулятор, отключить воспроизведение, получить данные из симулятора, а далее продолжить воспроизведении с приостановленного состояния [19].

5. Предлагаемая реализация обратной отладки

Наиболее используемыми в настоящее время в персональных компьютерах и мобильных устройствах являются процессоры архитектуры i386, AMD 64 и ARM. Немного симуляторов поддерживают все эти платформы, одним из них является симулятор QEMU. Он использует динамическую трансляцию, поддерживает такие целевые платформы как i386, AMD 64, ARM (есть версия для запуска Andriod), PowerPC, MIPS и другие. Имеет открытый исходный код, что позволяет добавлять поддержку новых платформ и периферийных устройств.

Ранее в симуляторе QEMU были реализованы функции детерминированного воспроизведения и обратной отладки [2]. QEMU поддерживает симуляцию не только процессора, а всей системы, включая состояние памяти, периферийных устройств и экрана симулятора. Эта особенность позволяет отлаживать работу драйверов для периферийных устройств даже в том случае, когда само устройство еще не выпущено, для чего можно смоделировать его внутри симулятора.

В основе механизма обратной отладки лежит детерминированное воспроизведение. В процессе работы приложения записывается журнал недетерминированных событий, происходящих внутри системы, именно он позднее позволяет повторить весь сценарий ее работы. К недетерминированным событиям относятся события срабатывания системных таймеров, прерывания и ввод-вывод от периферийных устройств.

При воспроизведении работы системы, события считываются из журнала и выполняются в те же моменты (по отношению к выполнению гостевой программы), в которые они были записаны. Использование журнала событий позволяет в точности повторять сценарий работы системы любое количество раз. Всякое замедление, возникающее при воспроизведении, никак не влияет на работу гостевой системы, так как не происходит реального ее взаимодействия с внешними источниками данных. Это свойство позволяет использовать алгоритмы анализа или снятия трассы, которые обычно дают сильное замедление и не могут быть использованы во время реальной работы из-за возможного изменения поведения исследуемой программы.

Для обеспечения контроля точности используется запись в журнал при необходимости дополнительных событий типа assert. При записи такого рода событий сохраняются значения ряда регистров гостевого процессора. Во время воспроизведения при считывании этих событий происходит сравнение значений регистров из журнала и состояние процессора симулируемой системы в данный момент. Если значения отличаются, то происходит завершение работы симулятора с сообщением об ошибке. Это позволяет избежать зависания или зацикливания при неверном воспроизведении журнала.

В процессе записи журнала событий должно быть сохранено содержимое сетевых пакетов, которые были отправлены или получены от виртуальной сетевой карты. Все сетевые пакеты сохраняются в отдельных файлах pcap, их можно анализировать отдельно с помощью программ анализаторов сетевого трафика подобных Wireshark [20]

QEMU поддерживает совместную работу с отладчиком GDB. Так как отлаживается всегда работа всей системы целиком, исследуемое приложение необходимо загрузить в ОС, установленную в симуляторе. С помощью команд обратной отладки в GDB, можно перемещаться по пути выполнения приложения в обратном направлении. Любая из этих команд требует длительное время для выполнения, поэтому важно рассчитать оптимальный период сохранения состояний системы для конкретной реализации обратной отладки, чтобы максимально сократить время выполнения команд обратной отладки.

Используя формулу 6, для QEMU с функциями обратной отладки можно построить график, изображенный на рисунке 1, зависимости времени выполнения команды reverse-continue от количества сохраненных снимков. Все данные получены для тестового случая загрузки ОС WindowsXP, в их

числе общее количество выполнившихся инструкций от начала записи журнала $n \approx 5 \cdot 10^9$, время воспроизведения одной инструкции $t_i \approx 6 \cdot 10^{-8}$ и время загрузки состояния $t_s \approx 3{,}5$.

На рис. 1 изображена зависимость времени выполнения команды reverse-continue от количества сохраненных снимков для симулятора QEMU. Графики построены для разного значения m – количество шагов до ближайшей точки останова.

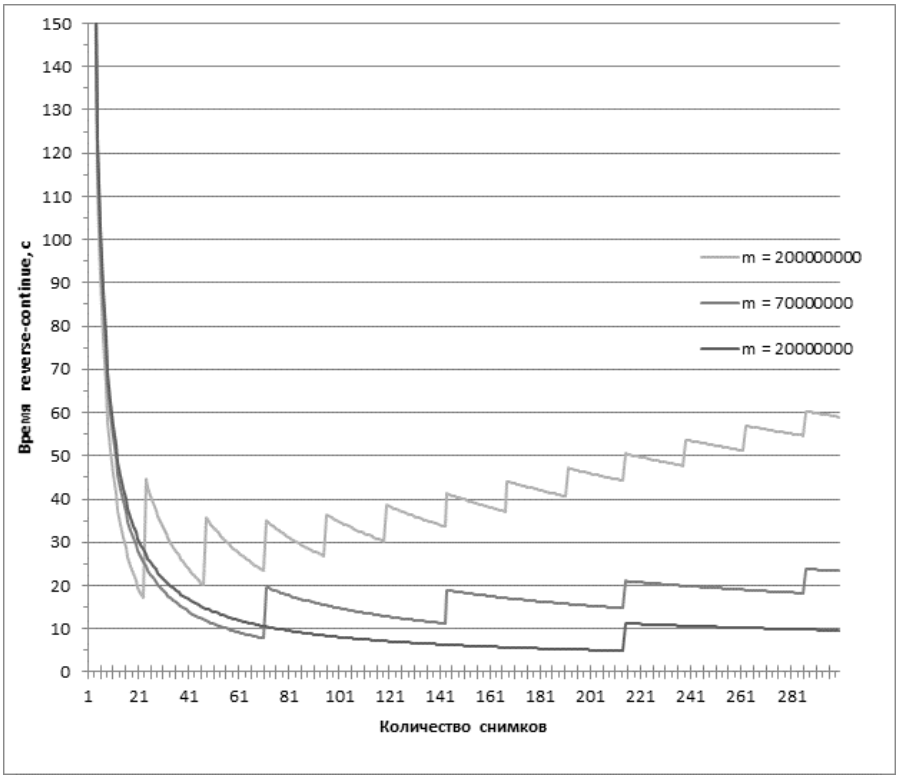

Рис. 1. Зависимость времени работы команды reverse continue от числа снимков состояния виртуальной машины, сделанных при записи ее работы.

Шаг, на котором встретится ближайшая точка останова, не известен, поэтому изображено три графика для различного среднего количества шагов, на которое необходимо вернуться команде reverse-continue.

В табл. 1 приведены результаты вычислений оптимального периода сохранения снимков при записи журнала, при котором время работы команд обратной отладки будет наименьшим.

Табл. 1 Оптимальный период сохранения снимков виртуальной машины при различном расстоянии до точки останова.

Количество шагов до точки останова	Количество сохраненных снимков	Период сохранения снимков, с
$2 \cdot 10^8$	23	5
$7 \cdot 10^7$	71	1,5
$2 \cdot 10^7$	214	0,5

6. Направления дальнейших исследований

Дальнейшая работа над обратной отладкой в симуляторе QEMU планируется в следующих направлениях.

Работа со снимками состояний, перенос их сохранения из этапа записи журнала на этап его воспроизведения, что позволит повысить скорость работы симулятора при записи журнала. Когда есть уже записанный журнал, пользователь сможет менять период сохранения снимков без перезаписи журнала. Использование копирования при записи для сохранения снимков и добавление возможности удаления лишних снимков позволит сократить общий объем хранимых снимков.

Протокол отладчика GDB на данный момент не позволяет эффективно использовать обратную отладку с симулятором QEMU. Когда пользователь хочет выполнить команду reverse-step n, то отладчик посылает симулятору n раз команду reverse-step, что подразумевает загрузку предыдущего состояния и воспроизведение журнала до предыдущей команды. Если бы симулятор получил сразу команду reverse-step n, то за один проход перешел на нужную инструкцию. При выполнении команды reverse-continue можно кэшировать все найденные точки останова, чтобы при повторном ее выполнении, избежать лишнего прохода. GDB имеет открытый исходный код, если его доработать, то связка GDB и QEMU работала бы намного эффективней.

Эффективность обратной отладки напрямую зависит от скорости ее работы, поэтому важно повышать скорость как записи, так и воспроизведения работы программы. Этого можно добиться за счет уменьшения количества событий, записываемых в журнал.

Планируется также добавить возможность исследовать внутреннее состояние системы с помощью инструмента Volatility Framework. Он работает только с файлами дампов памяти, но можно доработать его и интерактивно получать необходимую информацию, подключаясь к работающему симулятору.

7. Заключение

В работе рассматривается обратная отладка с использованием многоплатформенного симулятора QEMU. Применение обратной отладки при создании программных продуктов упрощает нахождение трудновоспроизводимых ошибок, что может существенно сократить время и

затраты на разработку. В настоящий момент на рынке нет реализации обратной отладки, сравнимой по производительности с прямой отладкой.

Описываемый в статье вариант обратной отладки имеет в своей основе симулятор QEMU 2.0 с функцией детерминированного воспроизведения. Для повышения эффективности отладки в работе были предложены методы оптимального сохранения снимков системы, индексации и сжатия журнала событий. Реализованный метод обратной отладки работает с пользовательскими приложениями, ядрами операционных систем. Поддерживается работа с периферийными устройствами из «реального мира», что позволяет отлаживать разрабатываемые и сопровождаемые драйверы устройств.

Список литературы

[1]. Довгалюк П.М. Обратная отладка программного обеспечения: монография. НовГУ им. Ярослава Мудрого. Великий Новгород 2013. 72 стр.

[2]. Довгалюк П.М. Детерминированное воспроизведение процесса выполнения программ в виртуальной машине. Труды ИСП РАН, т. 21, 2011 г. стр. 123-132, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print)

[3]. Dunlap, George W. and King, Samuel T. and Cinar, Sukru and Basrai, Murtaza A. and Chen, Peter M. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation, vol. 36, 2002, pp. 211-224.

[4]. Haikun Liu, Hai Jin, Xiaofei Liao, Zhengqiu Pan. XenLR: Xen-based Logging for Deterministic Replay. In proc. of Japan-China Joint Workshop on Frontier of Computer Science and Technology (2008). pp. 149-154.

[5]. Integrated Virtual Debugger for Visual Studio Developer's Guide. http://www.vmware.com/pdf/ws7_visualstudio_debug.pdf. Дата обращения: 11.06.2015

[6]. Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. Proceedings of the 2005 USENIX Annual Technical Conference, USENIX, 2005, pp. 1-15

[7]. J. Dike. A user-mode port of the Linux kernel. InProceedings of the 2000 Linux Showcase and Conference, 2000

[8]. QEMU - open source processor emulator. http://wiki.qemu.org/Main_Page. Дата обращения: 11.06.2015

[9]. Chia-Wei Hsu, Shiuhpyng Shieh. FREE: A Fine-grain Replaying Executions by Using Emulation. The 20th Cryptology and Information Security Conference (CISC 2010), Taiwan, 2010.

[10]. Bochs -the cross platform IA-32 emulator. http://bochs.sourceforge.net. Дата обращения: 11.06.2015

[11]. Daniela A. S. de Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T. Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. Proc. of the 1st workshop on Architectural and system support for improving software dependability (ASID '06), 2006. pp. 66-71

[12]. GDB and Reverse Debugging. http://sourceware.org/gdb/news/reversible.html. Дата обращения: 11.06.2015
[13]. Bob Boothe. Efficient Algorithms for Bidirectional Debugging. Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2000, pp. 299-310
[14]. Marc Rittinghaus, Konrad Miller, Marius Hillenbrand, and Frank Bellosa. Simuboost: Scalable parallelization of functional system simulation. In 11th International Workshop on Dynamic Analysis, WODA '03, Houston, Texas, March 2013
[15]. Robert O'Callahan. Efficient Collection and Storage of Indexed Program Traces. https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/Amber.pdf. Дата обращения: 11.06.2015
[16]. Volatility - an advanced memory forensics framework. https://code.google.com/p/volatility/. Дата обращения: 11.06.2015
[17]. Standalone Debugging Tools for Windows (WinDbg). http://msdn.microsoft.com/en-us/windows/hardware/hh852365. Дата обращения: 11.06.2015
[18]. Using DDMS. http://developer.android.com/tools/debugging/ddms.html. Дата обращения: 11.06.2015
[19]. Фурсова Н.И., Довгалюк П.М., Васильев И.А., Климушенкова М.А., Макаров В.А. Способы обратной отладки мобильных приложений. Проблемы информационной безопасности. Компьютерные системы, номер 3, Санкт-Петербург 2014 г, стр. 50-56
[20]. Wireshark User's Guide. https://www.wireshark.org/docs/wsug_html/. Дата обращения: 11.06.2015

Methods to Improve Reverse Debugging Performance

M.A. Klimushenkova <maria.klimushenkova@ispras.ru>
P.M. Dovgalyuk <pavel.dovgaluk@ispras.ru>
Novgorod State University
st. B. St. Petersburgskaya, 41, Veliky Novgorod, 173003, Russia

Abstract. Reverse debugging is software development technique that effectively helps to fix bugs caused by nondeterministic program behavior. Bug elimination usually includes multiple reruns of the program Program executions may be non-deterministic and the debugger may affect program's behavior. Reverse debugging allows inspecting past program's states without re-executing it. The paper describes implementation of software reverse debugging using deterministic replay based on the QEMU emulator. Our implementation of deterministic replay records high-level events (user and network input, CPU interrupts, USB and audio input). Record/replay subsystem saves these events into the log at the recording phase and reads them at the replaying phase. Therefore virtual machine is not connected to the real world in the replaying phase. We present ways to improve

debugging performance by reducing saved data, using copy-on-write snapshots' format and indexing/compressing of replay log. QEMU supports a common user interface for reverse debugging in GDB debugger which allows using reverse-continue (going back to the previous breakpoint or watchpoint), reverse-nexti, reverse-stepi (going back to the previous instruction), and reverse-finish (finding the point when function was called) commands. Time required for these commands' execution depends on taking snapshots frequency in recording replay log. We evaluate shapshotting frequency to get the best reverse debugging performance. In our implementation optimal period for taking snapshots is 3.5 seconds. This paper also presents assessment of snapshots frequency for better performance.

Keywords: reverse debugging; deterministic replay; QEMU; emulator.

DOI: 10.15514/ISPRAS-2015-27(2)-8

For citation: Klimushenkova M.A., Dovgalyuk P.M. Methods to Improve Reverse Debugging Performance. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 127-144 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-8.

References

[1]. Dovgalyuk P.M. Obratnaya otladka programmnogo obespecheniya: monografiya. Novgorod State University. Velikij Novgorod 2013. 72 p. (in Russian)

[2]. Dovgalyuk P.M. Determinirovannoe vosproizvedenie processa vypolnenija programm v virtual'noj mashine. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 21, 2011, pp. 123-132, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print) (in Russian)

[3]. Dunlap, George W. and King, Samuel T. and Cinar, Sukru and Basrai, Murtaza A. and Chen, Peter M. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation, vol. 36, 2002, pp. 211-224.

[4]. Haikun Liu, Hai Jin, Xiaofei Liao, Zhengqiu Pan. XenLR: Xen-based Logging for Deterministic Replay. In proc. of Japan-China Joint Workshop on Frontier of Computer Science and Technology (2008). pp. 149-154.

[5]. Integrated Virtual Debugger for Visual Studio Developer's Guide. http://www.vmware.com/pdf/ws7_visualstudio_debug.pdf. Access date: 11.06.2015

[6]. Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. Proceedings of the 2005 USENIX Annual Technical Conference, USENIX, 2005, pp. 1-15

[7]. J. Dike. A user-mode port of the Linux kernel. InProceedings of the 2000 Linux Showcase and Conference, 2000

[8]. QEMU - open source processor emulator. http://wiki.qemu.org/Main_Page. Access date: 11.06.2015

[9]. Chia-Wei Hsu, Shiuhpyng Shieh. FREE: A Fine-grain Replaying Executions by Using Emulation. The 20th Cryptology and Information Security Conference (CISC 2010), Taiwan, 2010.

[10]. Bochs -the cross platform IA-32 emulator. http://bochs.sourceforge.net. Access date: 11.06.2015

[11]. Daniela A. S. de Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T. Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. Proc. of the 1st workshop on Architectural and system support for improving software dependability (ASID '06), 2006. pp. 66-71

[12]. GDB and Reverse Debugging. http://sourceware.org/gdb/news/reversible.html. Access date: 11.06.2015

[13]. Bob Boothe. Efficient Algorithms for Bidirectional Debugging. Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2000, pp. 299-310

[14]. Marc Rittinghaus, Konrad Miller, Marius Hillenbrand, and Frank Bellosa. Simuboost: Scalable parallelization of functional system simulation. In 11th International Workshop on Dynamic Analysis, WODA '03, Houston, Texas, March 2013

[15]. Robert O'Callahan. Efficient Collection and Storage of Indexed Program Traces. https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/Amber.pdf. Access date: 11.06.2015

[16]. Volatility - an advanced memory forensics framework. https://code.google.com/p/volatility/. Access date: 11.06.2015

[17]. Standalone Debugging Tools for Windows (WinDbg). http://msdn.microsoft.com/en-us/windows/hardware/hh852365. Access date: 11.06.2015

[18]. Using DDMS. http://developer.android.com/tools/debugging/ddms.html. Access date: 11.06.2015

[19]. Fursova N.I., Dovgalyuk P.M., Vasil'ev I.A., Klimushenkova M.A., Makarov V.A. Sposoby obratnoj otladki mobil'nyh prilozhenij. Problemy informacionnoj bezopasnosti. Komp'juternye sistemy, vol. 3, Saint Petersburg 2014, pp. 50-56 (in Russian)

[20]. Wireshark User's Guide. https://www.wireshark.org/docs/wsug_html/. Access date: 11.06.2015

Тестирование реализаций клиента протокола TLS

А.В.Никешин <alexn@ispras.ru>
Н.В.Пакулин <npak@ispras.ru>
В.З. Шнитман <vzs@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. При разработке протоколов безопасности особую роль играют задачи обеспечения качества, надёжности и отказоустойчивости реализаций. Разработчики стандартов для таких протоколов ответственно подходят к разработке спецификаций и стараются предусмотреть множество возможных атак для того, чтобы минимизировать риск успешных атак на реализации. Поэтому для надежности реализаций очень важно соответствовать требованиям спецификации протокола, прежде всего в части обработки ошибок во входящих сообщениях или ошибки в последовательности сообщений, так как именно такие ситуации являются основным средством для осуществления атак на реализации протоколов.

Одним из основных инструментов для проверки соответствия реализации стандарту является тестирование. Данная статья продолжает серию работ авторов по автоматизации тестирования соответствия спецификациям протоколов безопаности интернета. В статье описаны результаты работы по созданию тестового набора для тестирования соответствия реализаций клиента протокола TLS спецификациям Интернета. В качестве базы для построения тестов мы использовали технологию UniTESK и программный пакет JavaTesK, реализующий эту технологию. Для построения атакующих воздействий на целевую реализацию использовался подход мутационного тестирования, при котором вместо корректных сообщений строятся их искаженные по определённым правилам копии. Были разработаны операторы мутации для некоторых основных типов данных, которые используются в формальной модели протокола. Подход был апробирован на ряде известных открытых реализаций клиентов протокола TLS. Этот подход доказал свою эффективность, поскольку обеспечил обнаружение отклонений от спецификации и других ошибок во всех выбранных реализациях клиента протокола.

Ключевые слова: тестирование; верификация; формальные методы; формальные спецификации; тестирование с использованием моделей; TLS; SSL; UniTESK; мутационное тестирование.

DOI: 10.15514/ISPRAS-2015-27(2)-9

Для цитирования: Никешин А.В., Пакулин Н.В., Шнитман В.З. Тестирование реализаций клиента протокола TLS. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 145-160. DOI: 10.15514/ISPRAS-2015-27(2)-9.

1. Введение

Технология UniTESK поддерживает нотацию формальных спецификаций, автоматическую динамическую генерацию тестовых воздействий и автоматический анализ результатов. UniTESK предоставляет средства для реализации формальных спецификаций протоколов в виде контрактных спецификаций и переходов расширенного конечного автомата [1]. Наш подход к применению технологии UniTESK для тестирования соответствия, в частности, вопросы построения формальной модели из текстовой спецификации и построения тестовых сценариев подробно изложены в [2,3]. Однако кроме воздействий, определяемых спецификацией, для более качественной проверки реализаций необходимо проводить тестирование на входных данных, противоречащих спецификации или не определяемых ею. Такие ситуации постоянно возникают из-за ошибок пользователей или из-за целенаправленных действий злоумышленника. Особенно это актуально для систем безопасности. Поэтому мы расширили наш тестовый набор средствами мутационного тестирования.

Наши предыдущие работы, относящиеся к протоколу TLS, были сосредоточены вокруг серверной части этого протокола [4]. В то же время, широко применяются клиентские приложения, использующие этот протокол (в том числе почтовые клиенты, web-браузеры и др.). Сегодня известны многочисленные уязвимости, найденные, например, в различных web-сервисах [5,6] (особенно использующих сценарии Java/Java scripts), демонстрирующие, что неадекватное поведение клиентских приложений может приводить к проблемам безопасности со стороны клиента. Кроме того, по объективным причинам тестированию клиентских приложений уделяется меньше внимания: многие серверные приложения являются коммерческими продуктами, в то время как, соответствующие клиентские решения часто предлагаются бесплатно. Поэтому тестирование клиентских приложений является актуальной задачей. Нами был разработан новый тестовый набор для TLS-клиентов, дополненный операторами мутации. Ниже дано краткое описание разработанного тестового набора и приведены результаты его апробации.

2. Тестовый стенд

Тестовый стенд для тестирования клиентов протокола TLS состоит из двух сетевых узлов: инструментального и целевого. На инструментальном узле исполняется основной поток управления тестовой системы, обход тестового автомата и верификация наблюдаемых реакций. На целевом узле функционирует тестируемая реализация и дополнительный агент, через

который инструментальный узел осуществляет управление реализацией. Инструментальный и целевой узлы могут располагаться в разных сегментах сети.

Стимулами в разработанном тестовом наборе являются сообщения от инструментального узла, а реакциями - сообщения со стороны тестируемого узла. Основная часть требований спецификации TLS проверяется в постусловиях реакций.

3. Инструментальные средства

Разработка тестового набора велась с использованием инструмента автоматизированного тестирования JavaTesK [7], который реализует технологию UniTESK на базе объектно-ориентированного языка с автоматической сборкой мусора и обеспечивает значительное упрощение модели протокола и тестовых сценариев.

4. Формальная спецификация

Формальная спецификация TLS состоит из следующих компонентов:

- модельного состояния, которое содержит набор структур данных, моделирующих концептуальные структуры данных из стандарта TLS;
- спецификационных стимулов, которые формализуют требования к изменению состояния реализации TLS при внешнем воздействии на систему;
- спецификационных реакций, которые формализуют требования к реакциям реализации TLS на внешние воздействия.

В спецификации каждое отправленное целевой системе TLS-сообщение рассматривается как последовательность стимулов. Каждый стимул в этой последовательности соответствует обработке отдельного блока данных в TLS-сообщении (TLS-сообщение может содержать несколько структур данных конкретного типа).

Каждое TLS-сообщение от целевой системы рассматривается как последовательность реакций. Каждая реакция в этой последовательности соответствует отдельному блоку данных в TLS-сообщении.

Параметрами спецификационных методов служат сообщения TLS в модельном представлении, т.е. объекты Java. Наборы полей соответствующих классов следуют структуре сообщений, описанных в стандарте TLS [8,9].

5. Модельное состояние

Модельное состояние представлено множеством TLS-соединений и множеством TLS-сессий на узле, моделирующем состояние целевой реализации.

TLS-соединение содержит параметры безопасности конкретного соединения, такие как криптографические алгоритмы, ключи. TLS-сессия содержит данные, необходимые для повторного использования согласованных ранее параметров безопасности, такие как сертификаты, криптографические алгоритмы, мастер-ключ (master secret).

Полное описание этих структур приведено в RFC 5246 и RFC 5746 [8,9].

TLS-соединение модели протокола включает следующие блоки данных:

selector селекторы трафика (адреса и порты TCP соединения), являющиеся идентификатором соединения;

current read state текущее состояние чтения;

current write state текущее состояние записи;

pending read state ожидаемое состояние чтения;

pending write state ожидаемое состояние записи.

Для каждого соединения TLS определяются четыре состояния: текущие состояния чтения и записи и ожидаемые (pending) состояния чтения и записи. Все сообщения обрабатываются, используя параметры текущего состояния. Параметры ожидаемых состояний устанавливаются с помощью обмена рукопожатия (TLS Handshake Protocol).

Модельный тип TLS-состояния включает:

sessionID идентификатор сессии, соответствующей данному соединению;

keys ключи криптографических алгоритмов;

sequence number порядковый номер сообщений;

security parameters параметры безопасности.

Модельный тип параметров безопасности соответствует структурам, описанным в стандарте:

```
struct {
    ConnectionEnd         entity;
    PRFAlgorithm          prf_algorithm;
    BulkCipherAlgorithm   bulk_cipher_algorithm;
    CipherType            cipher_type;
    uint8                 enc_key_length;
```

```
        uint8               block_length;
        uint8               fixed_iv_length;
        uint8               record_iv_length;
        MACAlgorithm        mac_algorithm;
        uint8               mac_length;
        uint8               mac_key_length;
        CompressionMethod   compression_algorithm;
        opaque              master_secret[48];
        opaque              client_random[32];
        opaque              server_random[32];
    } SecurityParameters;
```

connectionEnd данный флаг определяет, является узел сервером или клиентом для данного соединения;

prf_algorithm алгоритм, используемый для создания криптографических ключей из мастер-ключа;

bulk_cipher_algorithm,
enc_key_length,
block_length,
fixed_iv_length,
record_iv_length

алгоритм шифрования и соответствующие ему параметры;

mac_algorithm,
mac_length,
mac_key_length

алгоритм защиты целостности сообщений и соответствующие ему параметры;

compression_algorithm алгоритм сжатия;

master_secret мастер-ключ;

client_random, server_random одноразовые массивы байт клиента и сервера;

Модельный тип TLS-сессии включает:

id идентификатор сессии;

peer_certificate сертификат партнера, если такой используется;

compression_method метод сжатия;

cipher_spec криптографические алгоритмы;

master_secret мастер-ключ;

isResumable данный флаг определяет, может ли сессия использоваться для инициализации новых соединений.

6. Модель TLS-сообщений

Протокол TLS для передачи данных использует структуры, называемые TLS-записями (TLS Records), инкапсулирующие весь TLS-трафик. Спецификация определяет четыре типа передаваемых данных:

Обмен Рукопожатия (TLS Handshake Protocol), который используется для согласования новых параметров безопасности;

Протокол изменения состояния (Change Cipher Spec Protocol), единственное сообщение ChangeCipherSpec заменяет параметры текущего состояния параметрами соответствующего ожидаемого состояния;

Протокол Оповещения (Alert Protocol), предназначенный для передачи информационных сообщений и сообщений об ошибках;

Данные протокола верхнего уровня, использующего TLS в качестве транспорта.

Каждое TLS сообщение может содержать данные только одного из перечисленных выше типов, однако структур данных этого типа может быть

несколько (например, TLS-сообщение может содержать несколько сообщений протокола Рукопожатия). Тип данных указывается в заголовке TLS-записи.

Разработанная библиотека модельного представления сообщений TLS позволяет создавать различные варианты таких сообщений.

7. Тестирование TLS-клиента

В роли TLS-клиента реализация генерирует запросы. Дополнительный агент на целевом узле позволяет передавать команды от тестовой системы к реализации. В начале каждого теста через агента инициируется TLS-запрос от клиента. Далее стимулами являются сообщения от инструментального узла в качестве ответов клиенту.

В тестовом сценарии создается TLS-соединение, в рамках которого формируются ответы реализации клиента в модельном представлении, передаваемые затем функции отправки сообщений. В предусловии спецификационных функций стимулов проверяется правильность структуры тестового сообщения и его своевременность, и на основании этого делается вывод о том, должен ли на него быть ответ, сообщение об ошибке или реализация должна его проигнорировать. Из модельного представления тестового сообщения строится реализационное, которое и отправляется в сеть.

Сборщик реакций в течение заданного времени собирает ответные сообщения целевой системы. Из реализационных TLS-сообщений строятся их модельные представления. Последовательность блоков данных в полученных сообщениях рассматривается как последовательность реакций целевой системы.

В постусловии реакций данные проверяются на соответствие требованиям спецификации. Проверка разделена на несколько стадий. Сначала проверяется допустимость такого сообщения от реализации и его своевременность, затем - структура самого сообщения (присутствующие поля и их значения должны соответствовать текущему обмену).

После проверки всех требований, результат передается тестовому сценарию, где в зависимости от плана сценария, принимается решение о продолжении или завершении информационного обмена. В случае выявления нарушения требований принимается решение о критичности ошибки и возможности отправки следующих сообщений.

В случае успешного завершения обмена рукопожатия, создается новая TLS-сессия (за исключением сокращенного варианта обмена, в котором используется уже существующая сессия), параметры которой могут в дальнейшем использоваться для инициализации других TLS-соединений.

8. Выбор реализации для тестирования

В качестве клиентов были выбраны семь открытых реализаций:

Библиотечные реализации протокола:

- реализация TLS в виртуальной машине Java 1.7.0_05 (JSSE - Java Secure Socket Extension) [10];
- встроенная реализация клиента TLS библиотеки openssl-1.0.1j [11].

Интернет браузеры:

- интернет браузер Mozilla Firefox 34.0.5 [12];
- интернет браузер Opera 12.17 [13];
- интернет браузер SRWare Iron 42.0.2250.1 [14].

Почтовые клиенты:

- Mozilla Thunderbird 31.7.0 [15];
- TheBat 6.8.2 [16].

9. Результаты тестирования реализаций клиента протокола TLS на соответствие требованиям спецификаций RFC 5246 и RFC 5746

При выполнении разработанного нами тестового набора был выявлен ряд особенностей, нарушений требований RFC 5246/5746 и ошибок реализаций.

Реализация JSSE:

- принимает сообщения большей длины, чем допускается спецификацией;
- сообщения RecordLayer с неизвестным типом данных (спецификация определяет 4 типа данных: change_cipher_spec, alert, handshake, application_data), игнорируются клиентом (не вызывают ошибку);
- в сообщениях Alert, ServerHelloDone, HelloRequest, Finished обрабатывает первые 2, 4, 4, 12 байт соответственно (т.е. поле длины сообщения не проверяется) и только они учитываются в различных контрольных суммах, поэтому если длина сообщения больше установленной спецификацией вычисляется не правильное контрольное значение (HelloRequest не учитывается при вычислении контрольных сумм);

- принимает дубликаты сообщений ServerHello, Certificate, ServerKeyExchange, CertificateRequest, следующие друг за другом (т.е. ServerHello- ServerHello, Certificate- Certificate), если при этом не нарушается остальной порядок обмена;
- при получении сообщения Finished от сервера после сообщений ServerHello, ServerKeyExchange, ServerCertificate, CertificateRequest (без предварительного сообщения ChangeCipherSpec) создает TLS сессию, через которую позволяет передавать пользовательские данные (данная особенность поведения является серьезным нарушением требований спецификации);
- при получении после Finished (и создании TLS сессии) сообщения ServerHello создает еще одну TLS сессию, но после получения ServerHelloDone возвращает ошибку и закрывает сразу обе сессии;
- при отсутствии подходящего сертификата в большинстве случаев вместо пустого списка сертификатов отправляет сообщение об ошибке;
- при выборе подходящего сертификата не проверяет допустимые значения алгоритмов SignatureAndHashAlgorithm из сообщения CertificateRequest, но использует их в сообщении CertificateVerify. В случае несовпадения отправленного сертификата и доступных алгоритмов, отправляется сообщение об ошибке (вместо того, чтобы сразу выбрать правильный сертификат);
- если тип сертификата сервера не совпадает с алгоритмом обмена ключами, но при этом сообщение ServerKeyExchange корректно построено и соответствует сертификату, то соединение устанавливается.

Реализация Openssl-1.0.1j:

- принимает сообщения ServerHello с расширением “Signature Algorithms”, с неизвестными расширениями, которые клиент не

запрашивал, а также дубликаты этих расширений;

- после завершения обмена рукопожатия (как основного, так и возобновленного) на сообщение ServerHello (без дополнительного запроса со стороны клиента) отвечает сообщением ClientHello; на другие сообщения протокола рукопожатия (Handshake) отвечает сообщением ClientHello и следующим за ним сообщением об ошибке;
- при возобновлении сессии реализация всегда отправляет расширение "renegotiation_info" в сообщении ClientHello (RFC 5746), даже если сервер его не поддерживает;
- если сервер не использует расширение "renegotiation_info" при первом обмене Рукопожатия (указывая, что он его не поддерживает), а при последующих – использует с правильными контрольными данными, то клиент разрешает такие согласования (согласно спецификации клиент должен отправить сообщение об ошибке);
- если сервер использует пустое расширение "renegotiation_info" при первом обмене Рукопожатия (соглашаясь на безопасное переустановление параметров соединения), а при последующих – не использует совсем, то клиент разрешает такие согласования (что является критичным нарушением требований спецификации, и делает бесполезным использование данного расширения).

Реализация Mozilla Firefox:

- принимает сообщения ServerHello с расширением “Signature Algorithms”, а также дубликаты этого расширения (согласно спецификации данное расширение отправляется только клиентом, сервер не должен использовать его в ответе; также не допускается присутствие дубликатов расширений в одном сообщении);
- выбирает личный сертификат по своим правилам не используя данные из CertificateRequest;
- если тип сертификата сервера не совпадает с алгоритмом обмена

ключами, но при этом сообщение ServerKeyExchange корректно построено и соответствует сертификату, то соединение устанавливается;

- разрешает только безопасную переустановку криптографических параметров без разрыва соединения (session resume) (используется расширение TLS Renegotiation Indication Extension, RFC 5746), нарушая некоторые требования обратной совместимости и проверки наличия в сообщениях данного расширения. Такое поведение можно назвать особенностью реализации, поскольку оно направлено на усиление безопасности соединений.

Реализация Opera:

- если сервер использует пустое расширение "renegotiation_info" при первом обмене Рукопожатия (соглашаясь на безопасное переустановление параметров соединения), а при последующих – не использует совсем, то клиент разрешает такие согласования (что по сути делает бесполезным использование данного критичного расширения);
- не проверяет поле версии TLS в сообщении ServerHello (кроме значения 3.0), используя поле версии заголовка RecordLayer (данная особенность поведения является серьезным нарушением требований спецификации);
- информационное сообщение Alert (1,*) воспринимает как критическую ошибку (для критических ошибок должен использоваться шаблон Alert (2,*));
- при завершении передачи данных и закрытии соединения не отправляет сообщение Alert (1,0).

Реализация SRWare Iron 42.0.2250.1

- принимает сообщения ServerHello с расширением “Signature

Algorithms" и с неизвестными расширениями, которые клиент не запрашивал;

- после завершения обмена Рукопожатия на сообщение ServerHello отвечает сообщением ClientHello (вместо ошибки), на другие сообщения - сообщением ClientHello и следующим за ним сообщением об ошибке.

Реализация Mozilla Thunderbird 31.7.0:

- спрашивает у пользователя какой сертификат отправить серверу;
- если тип сертификата сервера не совпадает с алгоритмом обмена ключами, но при этом сообщение ServerKeyExchange корректно построено и соответствует сертификату, то соединение устанавливается;
- принимает сообщения ServerHello с расширением "Signature Algorithms".

Реализация TheBat 6.8.2:

- поддерживает только версию TLS1.1; считает TLS1.2 устаревшей версией и советует обновить ее до более новой.

10. Мутационное тестирование

Дополнительно с тестированием на соответствие спецификации с использованием формальных моделей сетевого протокола TLS, мы использовали методы мутационного тестирования, при котором на вход тестируемой реализации подаются заведомо некорректные данные. Мутационное тестирование основано на реализованной ранее формальной модели протокола, при котором в правильно построенные сетевые сообщения вносятся изменения. Данное тестирование использовалось на всех указанных выше реализациях клиентов TLS. Некоторые отклонения были найдены в реализациях JSSE (некорректная обработка длины некоторых сообщений, принятие дубликатов сообщений) и Opera (некорректная обработка поля версии в сообщении ServerHello) и указаны выше в перечне ошибок и особенностей реализаций.

11. Заключение

В данной статье представлены результаты работы по созданию тестового набора для тестирования соответствия реализаций клиента протокола TLS спецификациям Интернета. Наш подход для автоматизации тестирования использует технологию UniTESK: тестовая последовательность строится динамически как обход некоторого автомата теста, для построения тестовых воздействий и вынесения вердикта о корректности наблюдаемого поведения реализации используется модель протокола.

Кроме того, мы использовали метод расширения тестовых сценариев на основе моделей средствами мутационного тестирования. Такая комбинация тестовых методов позволяет улучшить качество тестирования реализаций и, как показала апробация предложенного подхода, обеспечивает обнаружение отклонений от спецификации и других ошибок во всех выбранных реализациях протокола.[1]

Список литературы

[1]. Bourdonov I., Kossatchev A., Kuliamin V., and Petrenko A. UniTesK Test Suite Architecture. //Proceedings of FME 2002. LNCS 2391, pp. 77-88, Springer-Verlag, 2002

[2]. Н.В.Пакулин, В.З.Шнитман, А.В. Никешин. "Разработка тестового набора для верификации реализаций протокола безопасности TLS // Труды Института системного программирования РАН, том 23, 2012 г. Стр. 387-404.

[3]. Н.В. Пакулин, В.З. Шнитман, А.В. Никешин. Автоматизация тестирования соответствия для телекоммуникационных протоколов. // Труды Института системного программирования РАН Том 26. Выпуск 1. 2014 г. Стр. 109-148.

[4]. Никешин А.В., Пакулин Н.В., Шнитман В.З. Автоматизация тестирования соответствия реализаций стандарту протокола безопасности транспортного уровня TLS // Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление. Выпуск 2(193)/2014. стр. 180-188. ISSN 2304-9766.

[5]. OWASP Top Ten Project, https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[6]. Дэйв Уайтлегг “Проверьте ваши приложения на уязвимости из списка OWASP Top 10 за 2013 год”, https://www.ibm.com/developerworks/ru/library/se-owasp-top10/

[7]. JavaTESK - http://www.unitesk.ru/content/category/5/25/60/

[8]. IETF RFC 5246, T. Dierks and E. Rescorla “The Transport Layer Security (TLS) Protocol Version 1.2”, August 2008.

[9]. IETF RFC 5746, E. Rescorla, M. Ray, S. Dispensa, N. Oskov “Transport Layer Security (TLS) Renegotiation Indication Extension”, February 2010.

[10]. JavaTM Secure Socket Extension (JSSE), http://docs.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html

[11]. OpenSSL Project, https://www.openssl.org/

[1] Проект выполняется при поддержке РФФИ, проект № 13-07-00869 «Разработка метода оценки устойчивости протоколов безопасности к атакам»

[12]. Mozilla Firefox, https://www.mozilla.org/ru/
[13]. Opera, http://www.opera.com/ru/
[14]. SRWare Iron, http://www.srware.net/ru/software_srware_iron.php
[15]. Mozilla Thunderbird, https://www.mozilla.org/ru/
[16]. TheBat, https://www.ritlabs.com/ru/products/thebat/

TLS Clients Testing

A.V. Nikeshin <alexn@ispras.ru>
N.V. Pakulin <npak@ispras.ru>
V.Z. Shnitman <vzs@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004

Abstract. Quality assuarance, reliability, fault tolerance are of major concern for developers of security protocols. Authors of specifications for those protocols take responsible approach to specification development and undertake significant efforts to study potential attacks and minimize the risk of effective exploits. Therefore it is vitally important for an implementation to conform to the corresponding protocol specification, especially in the context of error prcessing in inbound meesages or sequence of messages since such kinds of errors are the major facility for implementation of attacks against protocol implementations.
Testing is one of the primary tools for evaluation whether an implementation conforms to the specification. This paper continues the series of other publications of the authors dedicated to specification-based conformance testing for Internet security protocols. The paper presents a test suite for conformance testing of TLS protocol clients. The test suite is based on UniTESK technology of test construction and JavaTESK toolkit that implements the technology. The attacking inputs are constructed using mutation testing, building malformed test packets from correct originals following specific rules called "mutation operators". We developed mutation operators for a number of primary data types used in the formal model of the protocol. The approach was applied to a number of open-source well-known implementations of TLS. The approach proved to be feasible: a number of deviations from protocol specification and other errors were identified in all selected implementations of the protocol.

Keywords: testing, verification, formal methods, formal specifications, Model Based Testing, TLS, SSL, UniTESK, Fuzz Testing.

DOI: 10.15514/ISPRAS-2015-27(2)-9

For citation: Nikeshin A.V., Pakulin N.V., Shnitman V.Z. TLS Clients Testing. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 145-160 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-9.

References

[1]. Bourdonov I., Kossatchev A., Kuliamin V., and Petrenko A. UniTesK Test Suite Architecture. //Proceedings of FME 2002. LNCS 2391, pp. 77-88, Springer-Verlag, 2002

[2]. Nikeshin A.V., Pakulin N.V., Shnitman V.Z. Razrabotka testovogo nabora dlya verifikatsii realizatsiy protokola bezopasnosti TLS [Development of a test suite for the verification of implementations of the TLS security protocol], Trudyi Instituta sistemnogo programmirovaniya RAN [Proceedings of IPS RAS], 2012, Vol. 23, pp. 387–404 (in Russian).

[3]. Nikeshin A.V., Pakulin N.V., Shnitman V.Z. Avtomatizaciya testirovaniya sootvetstviya dla telecommunicacionnyh protocolov [Conformance testing automation for telecommunication protocols] // Trudyi Instituta sistemnogo programmirovaniya RAN [Proceedings of IPS RAS], 2014, Vol. 26 (Issue 1), pp. 109-148 (in Russian).

[4]. Nikeshin A.V., Pakulin N.V., Shnitman V.Z. Avtomatizacija testirovanija sootvetstvija realizacij standartu protocola bezopasnosti transportnogo urovnja TLS [Conformance testing automation for Transport Layer Security Protocol TLS], Naucno-tehnicheskie vedomosti SPbGPU. Informatika. Telekommunikacii. Upravlenie, Vol. 2(193)/2014, Pp. 180-188. ISSN 2304-9766 (in Russian).

[5]. OWASP Top Ten Project, https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[6]. Dave Whitelegg “Scan your app to find and fix OWASP Top 10 2013 vulnerabilities”, http://www.ibm.com/developerworks/security/library/se-owasp-top10/

[7]. JavaTESK - http://www.unitesk.ru/content/category/5/25/60/

[8]. IETF RFC 5246, T. Dierks and E. Rescorla “The Transport Layer Security (TLS) Protocol Version 1.2”, August 2008.

[9]. IETF RFC 5746, E. Rescorla, M. Ray, S. Dispensa, N. Oskov “Transport Layer Security (TLS) Renegotiation Indication Extension”, February 2010.

[10]. JavaTM Secure Socket Extension (JSSE), http://docs.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html

[11]. OpenSSL Project, https://www.openssl.org/

[12]. Mozilla Firefox, https://www.mozilla.org/ru/

[13]. Opera, http://www.opera.com/ru/

[14]. SRWare Iron, http://www.srware.net/ru/software_srware_iron.php

[15]. Mozilla Thunderbird, https://www.mozilla.org/ru/

[16]. TheBat, https://www.ritlabs.com/ru/products/thebat/

Конечные автоматы в теории алгебраических схем программ

Р.И. Подловченко <rimma.podlovchenko@gmail.com>
Московский Государственный Университет им. М.В. Ломоносова,
Россия, Москва, Ленинские горы, 1, стр. 52

Аннотация. Рассматриваемые в статье алгебраические модели программ обобщают две модели программ, введённые А.А. Ляпуновым и А.А. Летичевским. Показано, что алгебраические модели программ аппроксимируют компьютерные программы, через промежуточную формализацию. Центральное место в теории таких моделей занимает проблема эквивалентности схем программ. Существует достаточно много классов моделей, для которых эта проблема разрешима. Большинство разрешающих алгоритмов следуют структуре алгоритма проверки эквивалентности конечных автоматов. Целью данной статьи является выявление этой связи. Вводится эквивалентное представление моделей программ, называемое матричными схемами. Такое представление структурно ближе к конечным автоматам, и показано, что проблема эквивалентности в подклассе матричных схем программ сводится к таковой для конечных автоматов. Рассматривается алгоритм, решающий проблему эквивалентности КА; он формулируется в терминах требований к конечным участкам путей выполнения автоматов. В результате удаётся сформулировать более общий метод, применимый к другим моделям. Приводятся необходимые требования, предъявляемые к моделям для применимости к ним указанного метода. Приводятся две модели, уравновешенная полугрупповая модель с левым сокращением и коммутативная модель с монотонными операторами, в который разрешимость проблемы эквивалентности установлена указанным методом.

Ключевые слова: схема программ, алгебраическая модель программ, проблема эквивалентности, разрешающий алгоритм, конечный автомат.

DOI: 10.15514/ISPRAS-2015-27(2)-10

Для цитирования: Подловченко Р.И. Конечные автоматы в теории алгебраических схем программ. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 161-172. DOI: 10.15514/ISPRAS-2015-27(2)-10.

1. Введение

Рассматриваемые нами алгебраические модели программ введены в [1] как обобщение моделей, исследованных в [2] и [3]. Эти модели предназначены для изучения семантических свойств реальных программ и, в первую очередь, решения для них проблемы эквивалентности. Она состоит в поиске алгоритма, который, получив на свой вход две программы, распознаёт, эквивалентны они функционально или нет.

Принципиальным отличием алгебраических моделей программ от моделей из [2] и [3] является то, что они введены для программ, предварительно формализованных. Центральное место в теории таких моделей принадлежит проблеме эквивалентности схем программ, заменивших формализованные программы. В [4] доказана разрешимость этой проблемы в широком классе алгебраических моделей программ. Обозревая методику, которой получен данный результат, можно прийти к выводу: проверяемые ею свойства алгебраических моделей подсказаны теми свойствами конечных автоматов [5], которыми обеспечивается разрешимость их эквивалентности.

Задача статьи состоит в следующем: отметить эти свойства конечных автоматов и перевести их на язык требований, предъявляемых к алгебраическим моделям программ и прогнозирующих разрешимость в них проблемы эквивалентности. В совокупности эти требования определят метод исследований, названный нами автоматным. Он действительно открыл один из путей, которыми распознаётся эквивалентность схем в алгебраических моделях программ.

Статья начинается экскурсом в теорию алгебраических моделей программ. В разделе 1описывются формализованные программы и определяются сами модели. Здесь же осуществляется отбор моделей, пригодных для распознавания семантических свойств формализованных программ (теорема 1). Отобранные модели называются строго аппроксимирующими, и только они рассматриваются в теории. В разделе 2 формулируется теорема 2, сводящая проблему эквивалентности в алгебраической модели программ к родственной ей модели, элементами которой являются матричные схемы. Демонстрируется, что по своей структуре матричная схема – это конечный автомат. Предварительно воспроизводится определение последнего. Раздел 3 посвящён описанию и анализу алгоритма, разрешающего эквивалентность конечных автоматов. Отмечаются те свойства автоматов, которые играют принципиальную роль в работе алгоритма. Сам автоматный метод изложен в разделе 4, а заключительный раздел 5 отводится применению метода для двух видов алгебраических моделей программ, что говорит о его действенности в вопросе разрешимости проблемы эквивалентности.

2. Формализованные программы и построенные для них модели программ.

При формализации понятия программы исходной является следующая установка: в программе, описанной на алгоритмическом языке типа Паскаль, сохранить лишь отдел операторов; это означает, что устраняется ввод начальных данных и вывод полученных результатов. Кроме того, предполагается, что все данные имеют общий тип. Вместе с тем, сохраняются все обычные композиции операторов за исключением аппарата процедур.

Согласно такой установке, формализованная программа строится над двумя конечными алфавитами Y и P; элементами алфавита Y являются обозначения операторов, а элементами алфавита P – обозначения логических условий.

Синтаксически *программа* представляет собой конечный ориентированный и размеченный граф; в нём выделены две вершины: *вход* без входящих в него дуг и с одной исходящей дугой и *выход* – вершина без исходящих из неё дуг. Остальные вершины, если они имеются, подразделяются на *преобразователи* и *распознаватели*; преобразователь помечается символом из Y, и из него исходит одна дуга; распознаватель помечается символом из P, и из него исходят две дуги с метками 0 и 1 соответственно.

Пример программы дан на рис.1; здесь y_1, y_2 – символы из Y, приписанные преобразователям, а p_1, p_2 – символы из P, сопоставленные распознавателям.

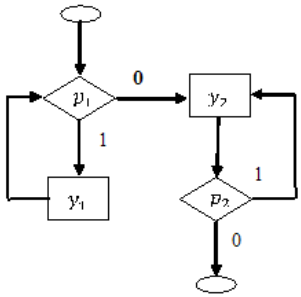

Рис. 1

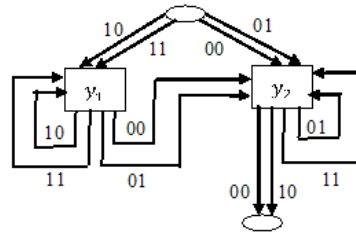

Рис. 2

Семантика формализованной программы (далее – просто программы) задаётся *семантикой* σ базиса Y, P; её составными частями являются:

- предметное множество Θ, элементы которого называются *состояниями*;
- отображения $\sigma_y: \Theta \to \Theta$, где $y \in Y$;
- отображения $\sigma_p: \Theta \to \{0,1\}$, где $p \in P$.

Для заданной семантики σ вводится процедура выполнения программы π на начальном состоянии ξ_0 из Θ. Она представляет собой детерминированный

обход программы π, который начинается в её входе при состоянии ξ_0 и подчиняется следующим правилам. Пусть при состоянии ξ достигнута вершина v программы π; если v – выход, то обход прекращается, а ξ воспринимается как *результат выполнения* π на состоянии ξ_0; если v – преобразователь с приписанным ему символом y, то состояние ξ перерабатывается в состояние $\sigma_y(\xi)$, и обход продолжается по дуге, исходящей из v; если же v – распознаватель с приписанным ему символом p, то состояние ξ сохраняется, а обход продолжается по дуге из v, помеченной числом $\sigma_p(\xi)$.

Таким образом, программе π сопоставляется реализуемая ею функция $\sigma\pi$, отображающая множество Θ в себя, в общем случае, частично. Программы π', π'' над Y, P называются *эквивалентными* при семантике σ, если $\sigma\pi' = \sigma\pi''$. Так возникает -класс программ, в котором рассматриваются проблема эквивалентности и проблема построения эквивалентных преобразований (э.п.) программ.

Перейдём к определению алгебраической модели программ над Y, P. Её элементами являются *схемы программ*. Чтобы всякое преобразование структуры схемы одновременно было бы преобразованием структуры программы, для которой построена схема, принимается решение: *синтаксически схема определяется так же, как программа*. Таким образом, предстоит определить лишь семантику схем программ.

Пусть G – схема над Y, P. Введём процедуру выполнения G на функции, которая отображает множество всех слов над алфавитом Y в множество X, где $X = \{x \mid x: P \to \{0,1\}\}$. Первое множество обозначим Y^*, а его элементы будем называть *цепочками*. Элементы второго множества назовём *наборами*, а саму функцию – *функцией разметки*, построенной над базисом Y, P. Обозначим $\mathcal{L}$ множество всех таких функций. Процедура выполнения схемы G на функции μ из $\mathcal{L}$ представляет собой детерминированный обход схемы G, который начинается в её входе с пустой цепочкой из Y^* и подчиняется следующим правилам. Допустим, что с цепочкой h достигнута вершина v схемы G. Если v – выход схемы, то обход её прекращается, а цепочка h называется *результатом выполнения схемы на* μ. Если h – преобразователь с приписанным ему символом y, то цепочка h трансформируется в цепочку hy, а обход схемы продолжается по дуге, исходящей из v. В случае, когда v – распознаватель, а p – сопоставленный ему символ, цепочка h остаётся неизменённой, а обход схемы продолжается по дуге, исходящей из v и помеченной числом, равным значению p в наборе $\mu(h)$. Таким образом схеме G приписывается отображение множества $\mathcal{L}$ в множество Y^*, частичное в общем случае.

Эквивалентность схем над Y, P индуцируется параметрами ν и L, где ν – отношение эквивалентности в Y^*, а L – подмножество множества $\mathcal{L}$, и определяется так: две схемы G_1, G_2 эквивалентны тогда и только тогда, когда,

какой бы ни была функция μ из L, если одна из схем останавливается на μ, то другая останавливается тоже, и результаты их выполнения на μ – это -эквивалентные цепочки. Множество схем над Y, P с введённой в нём эквивалентностью схем назовём *алгебраической моделью программ* над Y, P, а ν, L – её параметрами.

Напомним теперь, что введённые модели предназначены для исследования семантических свойств формализованных программ и для разработки на схемах эквивалентных преобразований программ. Это обязывает отобрать пригодные модели.

Пусть M – модель над Y, P, а σ – семантика базиса Y, P. Говорим, что M *аппроксимирует* –класс программ, если, какими бы ни были схемы G_1, G_2 из M, из их эквивалентности в M следует равенство функций $\sigma\pi_1, \sigma\pi_2$, где π_i – программа той же структуры, что и структура G_i, $i = 1,2$. Очевидно, что аппроксимирующая модель пригодна для объявленной выше задачи. Однако для отбора их пришлось предъявить дополнительные требования.

Модель M назовём *строго аппроксимирующей*, если существует такое множество S семантик базиса Y, P, что схемы из M эквивалентны тогда и только тогда, когда для любой семантики σ из S модель M аппроксимирует -класс программ. Достаточные условия строгой аппроксимируемости даёт

Теорема 1. Алгебраическая модель программ является строго аппроксимирующей, если её параметры ν и L удовлетворяют следующим требованиям: ν – это полугрупповая эквивалентность в Y^*, а множество L состоит из -согласованных функций разметки и является замкнутым по операции сдвига.

Теорема 1 доказана в [4]. Опишем используемые ею понятия. Эквивалентность ν называется *полугрупповой*, если, какими бы ни были цепочки h_1, h_2, h_3, h_4 из Y^*, из эквивалентности h_1, h_2 вместе с эквивалентностью h_3, h_4 следует эквивалентность цепочек h_1h_3 и h_2h_4. Функция разметки μ из $\mathcal{L}$ называется -*согласованной*, если она сохраняет своё значение на каждом классе ν-эквивалентных цепочек из Y^*. *Сдвигом функции* μ на цепочку h называется функция μ_h, обладающая свойством: для любой цепочки h' из Y^* верно равенство $\mu_h(h') = \mu(hh')$. Множество L, по определению, *замкнуто по операции сдвига*, если для любых μ из L и h из Y^* функция μ_h принадлежит L.

Как было отмечено во введении, рассматриваются только алгебраические модели, параметры которых удовлетворяют требованиям теоремы 1.

3. Матричные схемы и конечные автоматы

Связь между алгебраическими моделями программ и конечными автоматами, введёнными в [5], проявилась на основе теоремы 2, доказанной, в частности, в [4].

Теорема 2. Какой бы ни была алгебраическая модель программ M над базисом Y, P, проблема эквивалентности в ней сводится к родственной ей модели M_0 над тем же базисом.

Опишем модель M_0. Её элементами являются *матричные схемы*, построенные над Y, P. Структура матричной схемы выглядит так. Это – конечный ориентированный и размеченный граф, в котором выделены три вершины – *вход* без входящих в него дуг, *выход* и вершина loop, каждая без исходящих из неё дуг. Остальные вершины называются *преобразователями*. Каждому преобразователю приписан свой символ из Y. Из всякой вершины, отличной от выхода и loop, исходят дуги в количестве, равном числу наборов в X, причём каждая дуга помечена своим набором. Структура матричной схемы описана. На рис.2 приведена матричная схема, построенная по теореме 2 для схемы с рис. 1 и эквивалентная ей в любой модели M.

Матричной схеме сопоставляется отображение множества $\mathcal{L}$ функций разметки над Y, P в множество Y^*. Оно осуществляется процедурой выполнения схемы на функциях из $\mathcal{L}$. Описание процесса выполнения схемы из M_0 на функции μ отличается от того, как выполняется на μ схема из M, следующими деталями:

- обход схемы из её входа идёт по дуге, помеченной набором $\mu(\Lambda)$, где Λ – пустая цепочка из Y^*;
- из преобразователя с символом y, достигнутом при обходе с цепочкой h из Y^*, он продолжается по дуге, помеченной набором $\mu(hy)$;
- при достижении вершины loop обход прекращается без результата.

Параметрами модели M_0 являются параметры модели M, а отношение эквивалентности матричных схем в M_0 вводится также, как отношение эквивалентности схем в M. Поскольку далее будут рассматриваться модели типа M_0, их элементы называются просто схемами.

Модель M_0, параметрами которой являются тождество цепочек в Y^* и всё множество $\mathcal{L}$, называется *максимальной* в силу того, что из эквивалентности схем в этой модели следует эквивалентность их в любой модели из числа рассматриваемых нами. Имеет место лемма 1, доказанная в [6]. Она даётся в редакции, используемой в [7].

Лемма 1. Проблема эквивалентности в максимальной модели M_0 над базисом Y, P сводится к проблеме эквивалентности конечных автоматов над алфавитом $(Y \cup \{y_0\}) \times X$, где y_0 – добавочный символ.

Доказательству её предпошлём напоминание о том, как определяются конечные автоматы и отношение их эквивалентности.

Автоматом над алфавитом Σ называется конечный ориентированный граф с размеченными дугами. В нём одна вершина называется *инициальной*, и некоторые вершины (возможно, пустое множество) – *финальными*. Из каждой вершины исходят дуги в количестве, равном числу символов в Σ, и каждая

дуга помечена своим символом. Всякое слово над алфавитом Σ прокладывает в автомате маршрут, начинающийся в инициальной вершине и составленный дугами, метки которых при просмотре маршрута слагаются в это слово. По определению, оно *принимается автоматом*, если маршрут оканчивается в финальной вершине. Автоматы называются *эквивалентными*, если они принимают равные множества слов.

Доказательство леммы 1 состоит в построении алгоритма, который, получив на свой вход схему G из M_0, строит автомат $A(G)$, и в установлении корректности этого алгоритма. Ограничимся описанием алгоритма. Вершинами автомата $A(G)$ он объявляет образы всех вершин схемы G, при этом образ входа называет инициальной вершиной, образ выхода – единственной финальной вершиной, а образом loop – вершину, именуемую мёртвой, ибо все исходящие из неё дуги ведут в неё же.

Пусть v – вершина схемы G, а $\bar{v}$ – её образ в $A(G)$. Если v – это вход, то всякая дуга из него, помеченная набором x, порождает дугу из $\bar{v}$, помеченную парой (y_0, x) и ведущую в образ той вершины схемы, в которую ведёт первая дуга. Если v – это преобразователь с меткой y, то любая дуга из него, несущая метку x, порождает дугу из $\bar{v}$ с меткой (y, x), оканчивающуюся в образе той вершины схемы, в которой оканчивается первая дуга. Все остальные дуги, долженствующие быть в автомате $A(G)$, алгоритм направляет в мёртвую вершину. Этим автомат $A(G)$ построен. Обозревая описанный алгоритм, легко установить, что структура схемы из M_0 подобна структуре конечного автомата.

4. Разрешимость эквивалентности конечных автоматов.

Пусть конечные автоматы строятся над алфавитом Σ, где $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_\ell\}, \ell \geq 2$. Автомат над Σ назовём *приведённым*, если любая вершина в нем, за исключением мёртвой, находится на маршруте через него, то есть начинающимся в инициальной вершине и оканчивающимся в финальной. Мертвой называется вершина, в которую возвращаются все исходящие из неё дуги. Справедливо

Утверждение 1. Существует алгоритм, который поступивший на его вход автомат над Σ трансформирует в эквивалентный ему приведённый автомат над Σ.

Действительно, такой алгоритм сначала оставляет в автомате все вершины, достижимые из инициальной, затем все вершины, из которых достижима какая-либо финальная вершина, заменяя при этом прочие вершины мёртвой.

Воспользуемся алгоритмом, распознающим эквивалентность приведённых автоматов над Σ и известным как алгоритм Мура. Обозначим его для определённости символом ρ. Пусть A_1, A_2 – приведённые автоматы над Σ, поступившие на вход алгоритма ρ. Он строит таблицу, в которой, кроме ведущего столбца, имеется ℓ столбцов. Элементами таблицы будут пары

a_1, a_2 где a_i – вершина автомата $A_i, i = 1,2$. Элементы ведущего столбца таблицы называются *ведущими*, а позиция в строке таблицы, находящаяся в -ом столбце, – j-ой позицией. .

Сначала алгоритм ρ записывает в ещё пустую таблицу в качестве первого ведущего элемента пару (a_{10}, a_{20}), где a_{i0} – инициальная вершина автомата $A_i, i = 1,2$, и приступает к заполнению первой строки.

Предположим, что алгоритм ρ добрался до заполнения i-ой позиции строки с ведущим элементом (a_1, a_2). Тогда он строит пару (b_1, b_2), где b_i – вершина, в которую идёт дуга с меткой σ_j из вершины $a_i, i = 1,2$. В случае, когда вершины b_1, b_2 разнотипны (типы – финальная, мёртвая, прочая), алгоритм ρ останавливается, справедливо заявляя, что автоматы A_1, A_2 не эквивалентны. В ином случае он проверяет, имеется ли в ведущем столбце пара (b_1, b_2), и, если её нет, записывает её в первую свободную позицию ведущего столбца и продолжает свою работу, переходя к заполнению следующей позиции той же строки, если таковая имеется. Когда вся строка заполнена, а в таблице имеются незаполненные строки, алгоритм ρ работает со следующей строкой, останавливаясь с ответом «A_1, A_2 эквивалентны» в случае, когда таблица полностью построена. Этим алгоритм ρ описан.

Проанализируем работу алгоритма ρ, введя предварительно нужные понятия. Легко видеть, что всякий маршрут в автомате является *реализуемым*, то есть прокладывается некоторым словом над Σ. Равновеликие по длине маршруты в автоматах назовём *сочетаемыми*, если они прокладываются общим для них словом. Если в сочетаемых маршрутах повторяется пара равноудалённых от их начала вершин, то отрезки маршрутов, соединяющие эти вершины, назовём *сочетаемыми интервалами*.

Обратимся теперь к алгоритму ρ. Из его описания следует, что он строит сочетаемые маршруты в автоматах A_1, A_2, постепенно увеличивая их длину. При этом алгоритм ρ учитывает два свойства, присущие автоматам, а именно:

– если маршруты оканчиваются в вершинах разного типа, то автоматы не эквивалентны;

– в эквивалентных автоматах сочетаемые интервалы несут равные слова, и сокращение маршрутов на эти интервалы приводит к сочетаемым маршрутам.

Второе свойство позволяет не повторять пары в ведущем столбце, что и обеспечивает завершаемость построения таблицы.

5. Автоматный метод распознавания эквивалентности схем программ

Этот метод, отталкиваясь от свойств, на которые опирается алгоритм ρ, предписывает требования к модели с матричными схемами программ, выполнение которых прогнозирует разрешимость в ней проблемы эквивалентности. Областью его применимости являются только те модели, в

которых ν-эквивалентные цепочки над Y равны по длине; здесь ν – первый параметр модели.

Пусть M_0 – модель программ из этой области. Изложению автоматного метода предпошлём описание некоторых характеристик схем из M_0.

Маршрутом в схеме называется ориентированный путь в ней, начинающийся во входе схемы; если путь завершается в выходе схемы, то он называется *маршрутом через схему*. Всякому конечному маршруту в схеме сопоставим *несомую им цепочку* над Y, которая строится выписыванием друг за другом меток преобразователей при просмотре маршрута от начала к концу; если маршрут оканчивается в преобразователе, то его символ не учитывается. Маршрут в схеме называется *реализуемым*, если он прокладывается при выполнении схемы на некоторой функции разметки, допустимой для M_0. Маршруты в двух схемах называются *сочетаемыми*, если они прокладываются общей для них функцией разметки. В двух сочетаемых маршрутах, в которых повторяется пара равноудалённых от их начала вершин, отрезки их, соединяющие повторяющиеся вершины, будем называть *сочетаемыми интервалами*.

Необходимость в рассмотрении сочетаемых маршрутов в схемах из M_0 обоснована самим определением эквивалентности схем, которое можно дать в следующей редакции: схемы в M_0 эквивалентны тогда и только тогда, когда, каким бы ни был маршрут через одну из них, всякая функция разметки из L, прокладывающая этот маршрут, вместе с тем прокладывает в другой схеме маршрут через неё, несущий цепочку, -эквивалентную цепочке, несомой первым маршрутом.

Поскольку в модели M_0 имеются схемы с нереализуемыми маршрутами, автоматный метод выставляет

Требование 1. Доказать, что проблема эквивалентности в модели M_0 сводится к множеству принадлежащих ей схем со свойствами: любой маршрут в схеме реализуем, а каждый преобразователь находится на маршруте через неё.

Схема с указанным здесь свойством называется *свободной в* M_0.

Предположим, что это требование выполнено, и обозначим M_1 подмодель модели M_0, состоящую из всех свободных в ней схем. Теперь проблема эквивалентности рассматривается в M_1, и возникает

Требование 2. Сформулировать алгоритмически проверяемый критерий сочетаемости маршрутов в схемах из M_1.

Наконец, формулируется

Требование 3. Выбрать параметры модели M_1 такими, чтобы необходимыми условиями эквивалентности схем в ней являлись два условия:

1) равновеликие сочетаемые маршруты в схемах оканчиваются в вершинах общего типа;

2) сочетаемые интервалы в схемах несут -эквивалентные цепочки, и сокращение маршрутов на сочетаемые интервалы приводит к сочетаемым маршрутам.

Автоматный метод, которым прогнозируется разрешимость эквивалентности в модели M_0, описан.

6. Заключение

Первым примером применимости автоматного метода являются уравновешенные полугрупповые модели программ с левым сокращением, рассмотренные в [4]. Опишем их параметры ν и L.

Эквивалентность ν является полугрупповой, а множество L состоящим из всех -согласованных функций разметки. Таким образом, модель является аппроксимирующей. Уравновешенность модели трактуется как требование: -эквивалентные цепочки равновелики по длине. Это позволило применить к модели автоматный метод. Свойство левого сокращения формулируется так: какими бы ни были цепочки h_1, h_2, h_3, h_4 из Y^*, выполняется импликация $(h_1h_3 \sim h_2h_4) \wedge (h_1 \sim h_2) \Rightarrow h_3 \sim h_4$.

Доказательство разрешимости проблемы эквивалентности в этих моделях выполнено автоматным методом. Таблица, аналогичная той, что строится для конечных автоматов алгоритмом ρ, состоит из пар так называемых сопряжённых вершин схем: такие вершины достижимы равновеликими сочетаемыми маршрутами, которые несут -эквивалентные цепочки.

Другим примером является коммутативная модель программ с монотонными операторами, исследованная в [8]. В ней -эквивалентными объявляются цепочки, любая из которых получается из другой перестановками соседних символов. А множество L состоит из всех таких -согласованных функций разметки μ, которые удовлетворяют требованию : какими бы ни были цепочка h из Y^* и символ y из Y, верно соотношение $\mu(h) \leq \mu(hy)$; по определению, наборы x_1, x_2 находятся в отношении $\leq$, если для любого p из P выполняется: $x_1(p) \leq x_2(p)$.

Описанная модель является аппроксимирующей. Очевидно, что она входит в область применимости автоматного метода. При распознавании эквивалентности схем из этой модели таблицу, аналогичную таблице алгоритма ρ, приходится строить для каждого из маршрутов через схемы, ограничиваясь маршрутами, длина которых определяется размерами самих схем.

В [4] и [8] обсуждается и сложность алгоритма, распознающего эквивалентность схем.

Список литературы

[1]. Подловченко Р.И. Иерархия моделей программ. Программирование, 1981, №2, стр. 3-14.
[2]. Ляпунов А.А. О логических схемах программ. Проблемы кибернетики. М.: Физматгиз, Вып.1, 1958, стр. 46-74.
[3]. Глушков В.М., Летичевский А.А. Теория дискретных преобразователей. Избранные вопросы алгебры и логики. Новосибирск. ИМ СО АН СССР, 1973, стр. 5-39.
[4]. Подловченко Р.И. Об одной методике распознавания эквивалентности в алгебраических моделях программ. Программирование, 2011, № 6, с. 33-43.
[5]. Rabin M.O., Scott D. Finite automata and their decision problems. IBM Journal of Research and Development, 1959, vol.3, №2, p. 114-125.
[6]. Rutledge J.D. On Ianovs program schemata. Journal of the ACM, 1964, Vol. 11, № 1, p. 1-9.
[7]. Подловченко Р.И. Эквивалентные преобразования в математических моделях вычислений. Учебное пособие. М.: Издат. отдел факультета ВМиК МГУ им. М.В.Ломоносова. МАКС-Пресс, 2011, 72 с.
[8]. Подловченко Р.И. О схемах программ с перестановочными и монотонными операторами. Программирование, 2003, №5, стр. 46-54.

Finite State Automata in the Theory of Algebraic Program Schemata

R. I. Podlovchenko <rimma.podlovchenko@gmail.com>
Lomonosov Moscow State University,
1/52, Leninskie Gory, Moscow, Russia

Annotation. Algebraic program models considered in this paper generalize two models of programs introduced by A.A. Lyapunov and A.A. Letichevsky. The algebraic models are shown to approximate real-world programs via an intermediate formalization. The theory of program models focuses on the equivalence problem for program schemata. There are many results of decidability of this problem for various classes of algebraic program models. Most of these deciding algorithms derive from the equivalence checking algorithm for finite state automata. The aim of this paper is to reveal this relationship. An equivalent representation of schemes called matrix schemes is introduced. This representation is structurally closer to FSA, and it is proven that the equivalence problem in a subclass of program models represented as matrix models is reducible to one in FSA. The algorithm that checks an equivalence of FSA is studied and stated in terms of requirements applied to finite execution paths in automata. This results in a more general method which can be applied to other models, and necessary requirements models must satisfy for this method to be applicable are stated. Two models, namely the balanced semigroup model with left cancellation and the monotonous commutative model, are shown to have the equivalence problem decidable by the proposed method.

Keywords: program scheme, algebraic model of program, equivalence checking problem, decision procedure, finite state automaton.

DOI: 10.15514/ISPRAS-2015-27(2)-10

For citation: Podlovchenko R. I. Finite State Automata in the Theory of Algebraic Program Schemata. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 161-172 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-10.

References

[1]. Podlovchenko R.I. Ierarchiya modeley program [Hierarchy of program models]. Programmirovanije [Programming and Computer Software], 1981, № 2, p. 3-14 (in Russian).

[2]. Lyapunov A.A. O logicheskih shemah program [On the logical program schemata]. Problemy Kibernetiki [Problems of Cybernetics] Mosocow. Physmathgiz, vol. 1, 1958, p. 46-74 (in Russian).

[3]. Glushkov V.M., Letichevsky A.A. Teoriya discretnyh preobrazovateley [Theory of discrete transducers] Izbranniye voprosy algebry i logiki [Selected issues in algebra and logics] Novosibirsk, 1973, p. 5-39 (in Russian).

[4]. Podlovchenko R.I. On one equivalence checking technique for algebraic models of programs Programming and Computer Software, 2011, vol. 37, № 6, p. 292-298.

[5]. Rabin M.O., Scott D. Finite automata and their decision problems. IBM Journal of Research and Development, 1959, vol.3, №2, p. 114-125.

[6]. Rutledge J.D. On Ianovs program schemata. Journal of the ACM, 1964, vol. 11, № 1, p. 1-9.

[7]. Podlovchenko R.I. Equivalent transformations in mathematical models of computations. Tetxbook, Moscow, CMC Faculty, Lomonosov Moscow State University, 2011, 72 p. (in Russian)

[8]. Podlovchenko R.I. On program schemes with commuting and monotone operators. Programming and Computer Software, 2003, vol. 29, № 5, p. 270-276.

Разрешимость проблемы эквивалентных преобразований в классе примитивных схем программ

А. Э. Молчанов <gurux13@gmail.com>
Московский государственный университет, факультет вычислительной математики и кибернетики, 119333 Москва, Ленинские Горы, 1с52.

Аннотация. Статья принадлежит теории схем программ. Схемы программ - это объекты, созданные для анализа формализованных программ. Одной из основных задач теории является создание полных конечных систем эквивалентных преобразований (Э.П.). В статье рассматриваются схемы программ с процедурами, с ограничением на перегородчатые модели. Такие модели тесно связаны с моделями программ без процедур. Даётся краткое описание систем Э.П., и описывается методология решения проблемы Э.П. Выделяется подкласс перегородчатых моделей программ, называемый примитивными моделями. Они находятся ещё ближе к моделям без процедур. Указанная методология успешно применяется для построения полной конечной системы Э.П. в примитивном подклассе перегородчатых уравновешенных полугрупповых моделей программ с левым сокращением. Этот класс является широко изученным классом моделей программ, и при построении системы Э.П. используется известная конечная полная система Э.П. для похожих моделей без процедур. Используется вспомогательный вид схем, называемый многовыходными. В результате, строится канонический представитель класса эквивалентности схем программ, а также последовательность преобразований, приводящая произвольную схему этой модели в её каноническую форму. Такой способ решения считается полным решением проблемы Э.П. в классе моделей программ. В заключение приводятся направления для дальнейшего исследования.
Ключевые слова: алгебраические модели программ с процедурами; система эквивалентных преобразований; уравновешенная модель программ с левым сокращением**.**

DOI: 10.15514/ISPRAS-2015-27(2)-11

Для цитирования: Молчанов А.Э. Разрешимость проблемы эквивалентных преобразований в классе примитивных схем программ. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 173-188. DOI: 10.15514/ISPRAS-2015-27(2)-11.

1. Введение

Статья относится к теории алгебраических моделей программ, предназначенной для разработки эквивалентных преобразований программ на создаваемых для них схемах программ, и обобщает результат, полученный в [1].

Концепции, на основе которых строится теория, подробно изложены в [2]. Коротко перечислим их.

Программы берутся изначально формализованными. Схема программы по своей структуре совпадает с программой, для которой она создаётся. Отсюда всякое преобразование схемы одновременно является и преобразованием самой программы.

Моделью называется множество схем, построенных над некоторым базисом операторов, с введённым в нем отношением эквивалентности схем. Последнее вводится параметрически, что приводит к иерархии моделей. В теории рассматриваются только аппроксимирующие модели, т.е. такие, каждая из которых обладает свойством: из эквивалентности схем в модели следует эквивалентность программ, принадлежащих некоторому их классу, и совпадающих по структуре со схемами. В [2] описаны условия, налагаемые на параметры модели, при выполнении которых она является аппроксимирующей.

Обратимся теперь к вопросу о том, какое место в теории алгебраических моделей программ занимает рассматриваемая в статье задача.

Изучаемые модели подразделяются на модели программ, не использующих процедур, и модели программ, использующих их. Первые называются простыми, вторыми они обобщаются. Ключевой является проблема эквивалентных преобразований (э.п.) схем в модели. Она состоит в поиске системы э.п. схем, обладающей свойством: какими бы ни были две эквивалентные схемы из этой модели, конечной цепочкой преобразований, принадлежащих системе, одна из схем алгоритмически транслируется в другую. Естественно, что исследование проблемы э.п. проводятся в моделях с разрешимой эквивалентностью схем.

Первоначально рассматривались простые модели, и для них получен богатый арсенал результатов как по проблеме эквивалентности в модели, так и по проблеме э.п. в модели. Возникла задача – использовать эти результаты в моделях программ с процедурами.

В этих целях в [3] среди последних выделены так называемые перегородчатые модели программ. Их характерной особенностью является то, что параметры такой модели индуцируются параметрами её простой подмодели. Существенно и то, что если простая подмодель – аппроксимирующая, то и индуцируемая ей модель тоже аппроксимирующая.

В [4] формулируется условие, когда проблема эквивалентности в перегородчатой модели сводится в её свободную подмодель; так называется множество принадлежащих модели схем, в которых нет бездействующих элементом (схемы этого множества называются свободными).

Далее в [4] описан класс так называемых примитивных схем, принадлежащих свободной подмодели, и получен следующий результат: эквивалентность в этом классе разрешима, если она разрешима в индуцирующей его простой модели.

Рассматриваемая нами задача состоит в следующем: выбрав в качестве индуцирующей простую модель, называемую уравновешенной полугрупповой моделью программ с левым сокращением (она рассматривалась в [5]), решить проблему э.п. в классе примитивных схем программ. Отметим, что выбранная простая модель является аппроксимирующей, и в ней разрешимы проблема эквивалентности и проблема э.п.

В [1] требуемый результат получен для подкласса примитивных схем, обсуждающихся в данной статье. Это полностью определяет план её изложения.

В разделе 2 описывается общий вид алгебраической модели программ с процедурами и их частный случай – перегородчатые модели.

Примитивные схемы программ определены в разделе 3. Здесь же формулируются факты, установленные для них в [4], и описываются свойства, присущие эквивалентным примитивным схемам.

Раздел 4 посвящён методу решения проблемы э.п. в алгебраических моделях программ – воспроизводится его изложение, данное в [1].

В разделе 5 даётся определение используемой нами уравновешенной полугрупповой модели программ с левым сокращением, формулируются полученные в [5] результаты и описываются свойства принадлежащих им эквивалентных схем.

Построение системы э.п., полной в классе примитивных схем, индуцируемых уравновешенными полугрупповыми моделями с левым сокращением, осуществляется в разделе 6. Описываются канонические формы примитивных схем и алгоритм, транслирующий эквивалентные примитивные схемы в каноническую форму. При изложении алгоритма выделяются э.п., входящие в полную систему. Они описываются содержательно со ссылкой на их формальное определение в [5].

2. Рассматриваемые алгебраические модели программ

2.1 Общего вида алгебраические модели программ с процедурами

Модели программ с процедурами строятся над четырьмя конечными и непересекающимися алфавитами Y, C, R и X. Элементы первых трёх алфавитов называются символами операторов, вызовов и возвратов соответственно, элементы алфавита X именуются наборами. Алфавиты Y, C, R, X называются базисом.

Элементы модели называются схемами программ.

Схема программы по своей структуре представляет размеченный конечный ориентированный граф следующего строения. Граф распадается на подграфы с непересекающимися множествами вершин. Один из подграфов называется главным, остальные – процедурными. В главном подграфе выделены вершина вход без входящих в неё дуг и вершина выход без исходящих из неё дуг. В любом процедурном подграфе тоже выделены две вершины – инициальная и

финальная. В каждом подграфе имеется специальная вершина $loop$ без исходящих из неё дуг. Перечисленные вершины не имеют меток, а все остальные помечены символами из Y, C и R, называясь соответственно преобразователями, вызовами и возвратами. Всякому вызову соответствует свой персональный возврат, именуемый парным вызову; тот и другой принадлежат общему для них подграфу. Все вызовы перенумерованы. Из всякой вершины, отличной от выхода схемы, вызова, финальной и вершины $loop$, исходят дуги, каждая из которых помечена своим набором из X, в количестве, равном числу наборов в X. Из вызова исходит единственная дуга, которая ведёт в инициальную вершину некоторого подграфа (он называется ассоциированным с данным вызовом), и тогда из финальной вершины этого подграфа идёт дуга в парный вызову возврат, и это – единственная приходящая в него дуга. В инициальную вершину могут входить дуги только из вызовов, а из финальной вершины могут исходить дуги только в возвраты. Так осуществляется связь между подграфами, ибо начало и конец дуги иного типа находятся в общем для них подграфе.

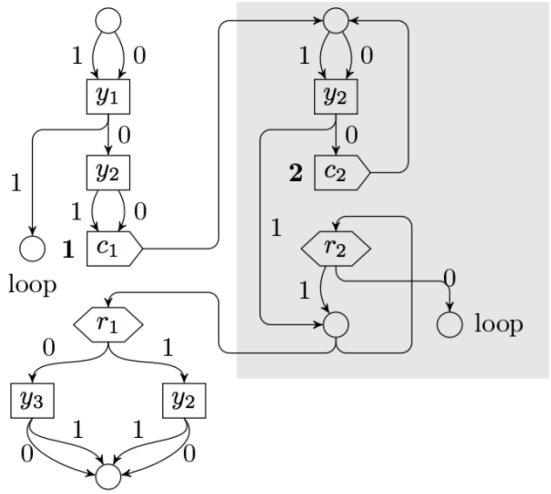

Рис. 1. Пример схемы программы

На рис. 1 приведён пример схемы программы. Здесь $\{y_1, y_2, y_3\} \subseteq Y, \{c_1, c_2\} \subseteq C, \{r_1, r_2\} \subseteq R, X = \{0,1\}$. Схема содержит один процедурный подграф.

Функционирование схемы осуществляется на функциях разметки. *Функцией разметки* называется отображение множества всех слов, построенных над $Y \cup C \cup R$ (оно обозначается H), в множество X. Множество всех функций разметки обозначается L.

Пусть $\mu \in L$. *Выполнением схемы на функции* μ называется процесс, состоящий в путешествии по схеме и сопровождающийся построением слова из H; оно называется *цепочкой*.

Для однозначности выбора пути, кроме μ, используется магазин, в который загружаются номера вызовов в схеме. Путешествие начинается по дуге, исходящей из входа, при пустых магазине и цепочке. Переход через вершину с сопоставленным ей символом сопровождается приписыванием этого символа к текущей цепочке справа. Если переходимая вершина – вызов, то в магазин загружается его номер. При переходе через инициальную вершину и финальную вершину текущая цепочка не изменяется. Во втором случае из макушки магазина, который заведомо не пуст, извлекается номер, и путешествие продолжается по дуге, ведущей к возврату, парному вызову с этим номером. Из входа схемы путь идёт по дуге, помеченной набором $\mu(\Lambda)$, где Λ – пустая цепочка. При переходе через преобразователь, возврат, инициальную вершину тоже используется функция μ: в качестве следующей берётся дуга, несущая метку $\mu(h)$, где h – построенная к этому моменту цепочка. При достижении выхода путешествие прекращается; говорим, что *схема остановилась на* μ, и построенную цепочку называем *результатом её выполнения на* μ. Альтернативным является случай бесконечного путешествия по схеме или случай попадания в вершину $loop$, когда путешествие прекращается без результата.

Пусть ν – эквивалентность в множестве H, и $L \subseteq L$. Схемы G_1, G_2 назовём (ν, L)-*эквивалентными* (обозначим: $G_1 \overset{\nu,L}{\sim} G_2$) тогда и только тогда, когда, какой бы ни была функция μ из L, всякий раз, как на ней останавливается с результатом одна из схем, останавливается с результатом и другая, и в этом случае результатами их выполнения являются ν-эквивалентные цепочки.

Множество всех схем над выбранным базисом, рассматриваемое вместе с (ν, L)-эквивалентностью схем, называется (ν, L)-*моделью*, а ν, L – её *параметрами*.

Схему из модели, не содержащую процедурных подграфов, назовём *простой*. Также *простой* именуем модель, построенную над базисом, в котором C и R пусты.

2.2 Перегородчатые модели программ

Определение перегородчатой модели программ в точности соответствует данному в [2].

Перегородчатая модель программ с процедурами представляет собой тот частный случай модели над базисом Y, C, R, X, в которой параметры ν, L индуцируются параметрами τ, l некоторой простой модели над Y, P; последняя называется *индуцирующей первую*.

Эквивалентность ν вводится следующим образом. Цепочки h_1 и h_2 из H считаем ν-эквивалентными в том и только том случае, когда совпадают их проекции на множество $C \cup R$, и, кроме того, если представить цепочки h_i, $i = 1,2$, в виде

$$h_i = h_{0i} b_1 h_{1i} b_2 \dots b_k h_{ki}, \quad k \geq 0,$$

где $b_1 b_2 \dots b_k$ есть общая проекция цепочек h_1, h_2 на множество $C \cup R$, то при всех $j, j = 0, \dots, k$, имеет место соотношение

$$h_{j1} \overset{\tau}{\sim} h_{j2}.$$

Здесь τ – эквивалентность цепочек g_1, g_2 над алфавитом Y записывается в виде $g_1 \overset{\tau}{\sim} g_2$.

Опишем, как строится множество L функций разметки.

Начнём с того, что задание отдельной функции разметки из L фактически сводится к разметке наборами из X вершин бесконечного дерева, в котором из всякой вершины исходят дуги в количестве, равном общему числу символов в алфавитах Y, C, R, причём каждая дуга помечена своим символом. Таким образом, всякой вершине дерева соответствует цепочка, полученная выписыванием друг за другом символов, метящих дуги пути, идущего из корня дерева в данную вершину. Набор из X, сопоставленный этой вершине, воспринимается как значение функции разметки именно на этой цепочке, а разметка всех вершин дерева наборами из X определяет функцию разметки.

Заметим теперь, что всякая вершина дерева является корнем его поддерева, все дуги которого помечены только символами из Y и всеми такими символами. Выделим поддеревья описанного типа, вырастающие из корня дерева или из вершины его, в которую ведёт дуга с меткой из C или R. Всякое выделенное поддерево разметим наборами из X так, чтобы это определило функцию разметки из l. По определению, выполненная разметка даёт функцию разметки из L, и иных функций в L нет.

Итак, перегородчатая модель программ с параметрами ν, L построена.

Легко доказуемо следующее утверждение.

Утверждение 1 *Перегородчатая модель программ является аппроксимирующей, если этим свойством обладает индуцирующая её простая модель.*

Затронем проблему либеризации для перегородчатой модели программ. Схема из неё называется *свободной*, если в ней любая вершина, кроме $loop$, принадлежит реализуемому маршруту через схему.

Пусть G – схема из рассматриваемой модели. *Маршрутом* в G назовём ориентированный путь w, начинающийся во входе схемы и заканчивающийся в некоторой вершине. Если последняя – это выход схемы G, то w именуем *маршрутом через схему*.

Маршрут в G назовём *реализуемым*, если в L существует функция разметки, которая его прокладывает при выполнении на ней схемы G.

Проблема либеризации – поиск алгоритма, который, получив на свой вход схему из рассматриваемой модели, трансформирует её в эквивалентную ей свободную схему из той же модели.

В [2] доказана

Теорема 1 *Проблема либеризации разрешима в перегородчатой модели программ, если она разрешима в индуцирующей её простой модели.*

Перегородчатая модель называется *специальной*, если она индуцируется простой моделью с разрешимой проблемой либеризации.

В [2] доказана

Теорема 2 *В специальной перегородчатой модели программ разрешима проблема либеризации.*

Назовём множество всех свободных схем в специальной перегородчатой модели программ её *свободной подмоделью*.

3. Примитивные схемы программ

Обозначим M свободную подмодель специальной перегородчатой модели над базисом Y, X, C, R; M_0 – индуцирующую её простую модель с параметрами τ, l.

Пусть G – схема из M. *Опорной* в G назовём вершину типа вход, вызов, возврат, выход схемы. *Преемником опорной вершины* v, отличной от выхода, назовём опорную вершину u, достижимую из v ориентированным путём, который не содержит внутри опорных вершин. Отметим, что в силу свободности схемы G любая её опорная вершина имеет алгоритмически определяемые преемники.

Свободная схема из M называется *примитивной*, если преемники любой её опорной вершины не имеют повторяющихся меток. Легко увидеть, что класс примитивных схем из M является разрешимым, то есть всегда можно установить, принадлежит ему или нет схема из M. В [4] установлен факт, следствием которого является теорема 3, самостоятельно доказанная [1].

Теорема 3 *Если в модели M_0 разрешима проблема эквивалентности схем, то она разрешима в классе примитивных схем из M.*

Опираясь на доказательство теоремы 3, приведённое в [1], опишем необходимые признаки эквивалентности примитивных схем.

Пусть G – примитивная схема из M, v – опорная в ней вершина, u – преемник вершины v. Обозначим $G(v,u)$ простую схему, построенную по следующим правилам. Сначала в ней оставляются все вершины, принадлежащие ориентированным путям из v в u. Если v – вход схемы G, то он объявляется входом создаваемой схемы, если u – выход схемы G, то он становится выходом создаваемой схемы. Если v – возврат, то он заменяется входом создаваемой схемы, если u – вызов, то он заменяется её выходом. Инициальную вершину всегда считаем входом создаваемой схемы, устраняя вызов v, из которого идёт дуга в данную инициальную вершину, а финальную – выходом, устраняя возврат u, в который из неё идёт дуга. Полагаем, что все оставленные преобразователи и наследуемые из G дуги сохраняют свои метки. Теперь введём в создаваемую схему вершину $loop$, если её там не было, устраним дуги, приходящие в оставленные вершины извне, а дуги, ведущие в

вершины наружу, направим в вершину $loop$. Схема $G(v,u)$ построена. Именуем её вырастающей из вершины v.

Маршруты в двух схемах из M назовём *сочетаемыми*, если они прокладываются при выполнении схем на общей и допустимой функции разметки. В эквивалентных примитивных схемах их опорные вершины назовём *сочетаемыми*, если ими оканчиваются сочетаемые маршруты, которые несут цепочки с равными проекциями на $C \cup R$. *Несомая* путём цепочка строится выписыванием друг за другом символов, метящих вершины этого пути при просмотре его от начала к концу. Отметим, что сочетаемые вершины, если помечены, то общим для них символом.

Пусть v_1, v_2 – сочетаемые вершины эквивалентных примитивных схем G_1, G_2 соответственно, и u_i – преемник вершины $v_i, i = 1,2$. Если преемники имеют общую для них метку в случае, когда они отличны от выходов схем, то схемы $G_1(v_1, u_1), G_2(v_2, u_2)$ называем *сочетаемыми*.

При доказательстве теоремы 3 установлен факт, формулируемый здесь как лемма 1.

Лемма 1 *В эквивалентных примитивных схемах из модели M каждой опорной вершине в одной из них соответствует единственная сочетаемая с ней опорная вершина в другой, а вырастающие из них сочетаемые схемы эквивалентны.*

Всякой опорной вершине v схемы из M сопоставим так называемую *многовыходную схему* $\hat{G}(v)$, построенную над базисом Y, X, D, где D – конечный алфавит, который будет описан ниже.

Многовыходная схема по своей конструкции отличается от простой схемы из M_0 только тем, что в ней может быть больше одной вершины без исходящих из неё дуг и называемых ей выходами; при этом выходы помечаются символами из D без повторений.

Такая схема выполняется на функции разметки, допустимой для M_0, по тем же правилам, что и простая схема из M_0. Если выполнение завершается в одном из выходов схемы, и только в этом случае, оно признаётся результативным, причём результатом объявляется цепочка, несомая проложенным маршрутом и завершаемая символом, сопоставленным достигнутого выхода.

Эквивалентными, по определению, являются две схемы, удовлетворяющие требованию: какой бы ни была допустимая для M_0 функция разметки, если выполнение на ней одной из схем результативно, то результативно и выполнение другой, и при этом результаты оканчиваются общим для них символом, а в остальном это τ-эквивалентные цепочки.

Для опорной вершины v схемы из M сопоставляемая ей многовыходная схема строится объединением в один граф всех простых схем, вырастающих из v, сопровождаемая приписыванием выходу схемы метки преемника, для которого она построена; если же преемником является выход самой схемы из M, то в многовыходной схеме он помечается специальным символом y_0. Таким образом, в качестве алфавита D берётся сумма C и R, дополняемая этим

специальным символом. Будем говорить, что построенная многовыходная схема вырастает из v.

В [1] установлена справедливость утверждения 2

Утверждение 2 *Какими бы ни были сочетаемые опорные вершины эквивалентных приведённых схем, вырастающие из них многовыходные схемы эквивалентны.*

4. Методика решения проблемы э.п. в алгебраической модели программ

Для решения проблемы э.п. схем строится специального вида формальное исчисление, то есть описываются его элементы и вводятся действующие в нём правила вывода и аксиомы.

Здесь мы изложим концепции, которыми руководствуется такое построение, обобщая при этом результаты исследований, выполненных в [2] и [3].

Итак, пусть S – рассматриваемая алгебраическая модель программ или рассматриваемый класс принадлежащих ей схем программ, и в S разрешима проблема эквивалентности.

В самом начале построения формального исчисления для S независимо от типа выполняется следующее:

- вводится понятие фрагмента схемы, и фрагменты объявляются элементами создаваемого исчисления;
- определяется единственное в исчислении правило вывода, состоящее в допустимой подстановке во фрагмент вместо его подфрагмента некоторого другого фрагмента; при этом подстановка является обратимой операцией.

Далее используется отношение эквивалентности схем из S и, опираясь на него, выполняется следующее:

- определяется отношение эквивалентности фрагментов с классификацией на безусловную и условную эквивалентность;
- объявляется, что аксиомой исчисления может быть разрешимое множество пар эквивалентных фрагментов, причём состоящее либо из пар безусловно эквивалентных, либо из пар условно эквивалентных фрагментов;
- вводится конечное множество аксиом, и формальное исчисление считается построенным.

Отметим, когда оно служит решению проблемы э.п. в S.

Применяя правило подстановки, каждой аксиоме сопоставляется множество э.п. схем, реализуемых её использованием; оно именуется индуцируемым аксиомой и является разрешимым. Таким образом, конечным числом аксиом в

исчислении индуцируется в совокупности разрешимое множество э.п. схем. Если доказывается его полнота в S, то в S решена проблема э.п. схем.

Теперь – о поиске нужных для S аксиом. Он подчиняется стратегии, рекомендации которой состоят в следующем:

- ввести каноническую форму схем из S, подчинив это требованию, чтобы каноническая форма была алгоритмически распознаваема и была единственной с точностью до изоморфизма в своём классе эквивалентных схем из S;
- построить алгоритм, который, получив на свой вход две эквивалентные схемы из S, трансформирует каждую в каноническую, применяя только эквивалентные преобразования их структуры;
- проанализировать этот алгоритм, переводя выполняемые им преобразования на язык фрагментных, и скомпоновать таким образом аксиомы.

Заметим, что анализ алгоритма сопровождает его построение.

Если изложенные рекомендации реализованы, и получено конечное число аксиом, то в силу обратимости выполненных алгоритмом преобразований будет получено решение проблемы э.п. схем в S.

Определения фрагмента, его вхождения в схему, согласованных фрагментов и подстановки фрагмента подробно представлены в [1]. Приведём здесь лишь содержательное описание.

Фрагментом называется часть схемы, для которой указана связь с остальной частью схемы.

Вхождение фрагмента F в схему G – это фрагмент схемы G, изоморфный F.

Согласованность фрагментов обеспечивает корректную выполнимость правила подстановки вместо одного фрагмента согласованного с ним; корректность трактуется так: результатом подстановки является фрагмент, в частности, схема, если подставляемый фрагмент входил в неё.

Описанная операция подстановки объявляется единственным правилом вывода в строящемся исчислении.

Согласованные фрагменты F_1, F_2 называются *безусловно эквивалентными*, если выполняется следующее: какой бы ни была схема из S и каким бы ни было вхождение в неё одного из фрагментов F_1, F_2, подстановка вместо этого вхождения другого фрагмента является эквивалентным в S преобразованием данной схемы. *Условная эквивалентность* этих *фрагментов* имеет место в том случае, когда существует алгоритм, определяющий допустимые вхождения подставляемого фрагмента.

В заключение отметим, что в разделе 6 данной статьи изучается решение проблемы э.п. в классе примитивных схем в случае, когда индуцирующая модель M_0 – это уравновешенная полугрупповая модель программ с левым сокращением.

5. Уравновешенные полугрупповые модели программ с левым сокращением

Описывая эти модели, мы фактически даём аналитический обзор статьи [5].

Простая алгебраическая модель программ над базисом Y, P называется *уравновешенной полугрупповой моделью программ с левым сокращением*, если её параметры τ и l удовлетворяют следующим требованиям.

Эквивалентность τ является *полугрупповой*, то есть, какими бы ни были цепочки h_1, h_2, h_3, h_4 над Y, выполняется импликация

$$(h_1 \overset{\tau}{\sim} h_2) \& (h_3 \overset{\tau}{\sim} h_4) \Rightarrow h_1 h_3 \overset{\tau}{\sim} h_2 h_4;$$

и обладающей *левым сокращением*, если для любых цепочек h_1, h_2, h_3, h_4 над Y выполняется импликация

$$(h_1 h_3 \overset{\tau}{\sim} h_2 h_4) \& (h_1 \overset{\tau}{\sim} h_2) \Rightarrow h_3 \overset{\tau}{\sim} h_4;$$

Кроме того, эквивалентность τ является уравновешенной, то есть τ-эквивалентные цепочки равны по длине.

Множество l состоит из всех τ-согласованных функций разметки.

По-прежнему обозначаем M_0 описанную модель программ.

Легко доказуемо, что для модели M_0 разрешима проблема либеризации. В [3] установлены факты, излагаемые ниже как теорема 4 и теорема 5.

Теорема 4 *В модели M_0 разрешима проблема эквивалентности схем.*

Теорема 5 *В модели M_0 разрешима проблема э.п. схем.*

Полагаем схемы в M_0 свободными.

Остановимся на описании необходимых свойств эквивалентности схем из модели M_0, формулируя их как утверждения 3-5

Утверждение 3 *Равновеликие сочетаемые маршруты в эквивалентных схемах оканчиваются в вершинах общего типа.*

Утверждение 4 *Из сопряжённых и различно помеченных вершин эквивалентных схем вырастают кусты общего маркера.*

Пусть G – схема из M_0, и v – преобразователь в ней. *Кустом, произрастающим из* v, называется фрагмент схемы G, состоящий из преобразователей, включая v, и удовлетворяющий следующим требованиям. Все вершины фрагмента распределены по уровням, на каждом из которых находятся равноудалённые от v преобразователи, при этом, если k – число уровней, то дуги, исходящие из вершин, принадлежащих уровню $i, i = 0,1, \dots, k-1$, ведут в вершины $i+1$-ого уровня; таким образом, из вершины v на k-ый уровень ведут простые ориентированные пути; требуется, чтобы все они несли эквивалентные цепочки. Класс эквивалентности этих цепочек называется *маркером* данного *куста*, а вершина v – его *корнем*.

Здесь *сопряжёнными* в двух схемах называются преобразователи, в которых завершаются сочетаемые маршруты, несущие цепочки, которые являются τ-эквивалентными по устранении из них последнего символа.

Кусты, о которых говорится в утверждении 4, также называются *сопряжёнными*.

Утверждение 5 *В эквивалентных схемах для любого набора $x \in X$ x–выходы сопряжённых кустов эквивалентны.*

По определению, *x–выходом куста* в схеме называется вершина, в которую из последнего уровня куста ведёт дуга с меткой x; $x \in X$.

Введём отношение эквивалентности вершин в схеме из M_0.

Любой вершине v схемы G из M_0 сопоставим простую и свободную схему $G(v)$, построенную по следующим правилам. Берётся фрагмент схемы G, порождённый всеми её вершинами, достижимыми из v ориентированными путями. Среди них, в силу свободности схемы G, находится её выход, который будем считать выходом схемы $G(v)$. Реконструируем этот фрагмент, внеся в него вершину *loop*, если её не было, затем устраним все входящие во фрагмент извне его дуги и направим в вершину *loop* все исходящие из него наружу дуги. В заключение добавим вершину, именуемую входом схемы $G(v)$, отправив в вершину v все исходящие из неё дуги. Схема $G(v)$ построена.

Вершины v_1, v_2 схемы G из M_0 назовём *эквивалентными*, если эквивалентны схемы $G(v_1), G(v_2)$.

Отметим, что доказательство теоремы 4 выполнимо проверкой утверждений 3-5 при просмотре пар сопряжённых вершин испытуемых схем.

6. Решение проблемы э.п. в классе примитивных схем, индуцированных моделью M_0

Следуя методике, изложенной в разделе 4, определим каноническую схему в классе K примитивных схем, индуцированных моделью M_0.

Схему G из K назовём *канонической*, если выполнены условия:

- в G отсутствуют эквивалентные опорные вершины;
- какой бы ни была опорная вершина v схемы G, многовыходная схема $\hat{G}(v)$ не содержит кустов и сильно эквивалентных вершин;
- для разных опорных вершин v_1, v_2 многовыходные схемы $\hat{G}(v_1), \hat{G}(v_2)$ не имеют общих вершин, кроме опорных.

Определим используемые здесь понятия. Пусть G – схема из K.

Опорной вершине v, являющейся вызовом, сопоставим схему $\nabla G(v)$. Эта схема строится следующим образом. Главный подграф G удаляется и заменяется на главный подграф следующего вида. Он содержит входную и выходную вершины, все дуги из входной вершины ведут в единственный вызов, который помечен тем же символом, что и v, и инцидентен тому же

подграфу, что и v. Единственный возврат, принадлежащий главному подграфу, парен этому вызову, и все дуги из него ведут в выход. Других вершин в главном подграфе нет. В остальном схема $\nabla G(v)$ совпадает с G.

Вершины-вызовы v_1, v_2 схемы G назовём *эквивалентными*, если эквивалентны схемы $\nabla G(v_1), \nabla G(v_2)$.

Схемы G_1, G_2 из M_0 назовём *сильно эквивалентными*, если они эквивалентны, и сочетаемые маршруты через них несут равные цепочки.

Вершины v_1, v_2 схемы G из M_0 назовём *сильно эквивалентными*, если сильно эквивалентны схемы $G(v_1), G(v_2)$.

Лемма 2 *Каноническая схема из K алгоритмически распознаваема и единственна с точностью до изоморфизма в своём классе эквивалентности.*

Proof. Очевидно, что в силу разрешимости отношений эквивалентности и сильной эквивалентности, свойства 1-2 канонической схемы алгоритмически распознаваемы.

Свойство 3 распознаваемо обходом графа схемы из каждой опорной вершины. Следовательно, каноническая форма является алгоритмически распознаваемой.

Пусть существует две канонические формы $G_1, G_2, G_1 \sim G_2$, не являющиеся изоморфными. Пусть в G_1 существует вершина v, не являющаяся опорной, которая не имеет сопряжённой в G_2. В этом случае, в силу свободы схем, они не эквивалентны. Если вершина v схемы G_1 сопряжена более чем с одной вершиной G_2, то эти вершины в G_2 эквивалентны (тогда G_2 – не каноническая форма), либо принадлежат различным схемам вида $\widehat{G_2}(v_1), \widehat{G_2}(v_2)$ (тогда G_1 не является канонической формой, поскольку содержит пересекающиеся схемы, вырастающие из различных опорных вершин).

Если же схемы G_1, G_2 отличаются опорными вершинами, то они неэквивалентны в силу примитивности схем.

Лемма доказана.

Теперь опишем алгоритм ρ, который, получив на свой вход две эквивалентные схемы G_1, G_2, строит эквивалентную им каноническую схему G.

Сначала алгоритм ρ в каждой из схем G_1, G_2 просматривает пары одинаково помеченных вызовов и устанавливает, эквивалентны они или нет. В случае эквивалентности алгоритм ρ выполняет склеивание этих вершины путём переброса на оставляемую вершину дуг, приходящих в другую.

Это преобразование поддерживается условной аксиомой $B1$, формальное определение которой совпадает с определением условной аксиомы $A6$ в [5]. Здесь и далее мы будем ссылаться на аксиомы из [5], определяющие эквивалентные преобразования системы э.п., полной в M_0, ограничиваясь лишь содержательным описанием самих преобразований.

После применения аксиомы $B1$ схема перестаёт быть свободной, и поэтому применяется преобразование, описываемое безусловной аксиомой $B2$. Оно

состоит в удалении из схемы вершин, не достижимых из её входа; это преобразование обобщает описанное в [5] аксиомой $A7$.

Обозначим $G_{1,}, G_{2,}$ схемы, эквивалентные исходным схемам G_1, G_2, и не содержащие эквивалентных вызовов.

Далее, алгоритм ρ работает с каждой парой многовыходных схем, вырастающих из сочетаемых опорных вершин схем $G_{1,}, G_{2,}$. При этом, если существует преобразователи, принадлежащие нескольким многовыходным схемам, они *расклеиваются*, то есть, создаются копии преобразователей так, чтобы каждая копия принадлежала своей многовыходной схеме. Это преобразование поддерживается безусловной аксиомой $A1$ из [5].

Обозначим ρ_1 выполняемый для каждой пары многовыходных схем алгоритм.

Пусть v_1, v_2 – рассматриваемая пара вершин схем $G_{1,}, G_{2,}$ соответственно. Поскольку многовыходные схемы $\widehat{G_{1,}}(v_1)$ и $\widehat{G_{2,}}(v_2)$ эквивалентны, то известны принадлежащие им сопряжённые вершины, следовательно, и сопряжённые кусты в них.

На основании утверждения 5 справедлива

Лемма 3 *Если F_1, F_2 – сочетаемые кусты из схем $\widehat{G_{1,}}(v_1)$ и $\widehat{G_{2,}}(v_2)$, то при любом $x \in X$ все x–выходы этих кустов находятся в сочетаемых схемах, вырастающих из v_1, v_2 соответственно.*

Для каждой пары сочетаемых кустов проводится трансформация, подробно описанная в [1]. Приведём её кратко.

В первую очередь, сопряжённые кусты переводятся в *чистые* кусты, то есть, кусты, в которых все входящие извне дуги ведут в корень куста. Эта трансформация проводится путём *отклеивания* подкуста с формированием новой пары сопряжённых кустов. Это преобразование обеспечивается аксиомой $A1$.

Далее обеспечивается следующее свойство сопряжённых кустов: для каждого $x \in X$ все x–выходы куста ведут в одну и ту же вершину. Это требование выполняется путём склеивания x–выходов кустов при помощи аксиомы $B1$.

После этого кусты трансформируется в *линейные*, то есть, просто в цепочки преобразователей, несущие операторные символы, соответствующие маркерам кустов. Преобразование проводится при помощи безусловной аксиомы $A7$ из [5].

В процессе преобразований могли получаться несвободные схемы, поэтому алгоритм повторяет трансформацию схем в свободные согласно аксиоме $B2$.

В результате имеет место следующее утверждение.

Утверждение 6 *Преобразованные многовыходные схемы сильно эквивалентны.*

Здесь сильно эквивалентными называются эквивалентные схемы, составными частями которых являются сильно эквивалентные простые схемы.

Осталось применить аксиому $A6$, склеивающую сильно эквивалентные вершины.

Алгоритм ρ_1 описан.

Применением алгоритма ρ пара схем приводится к паре изоморфных канонических схем, следовательно, верна

Теорема 6 *Система э.п., индуцируемая аксиомами*

$$B1, B2, A1 - A7,$$

является полной в классе K.

7. Заключение

Дальнейшие исследования в этой области можно вести в следующих направлениях:

- исследование других простых моделей программ, в которых разрешима проблема эквивалентных преобразований, для применения решения к многовыходным схемам и, соответственно, к перегородчатым моделям программ;
- поиск представлений схем из перегородчатых моделей, которые могут быть альтернативой многовыходным схемам; результаты в этой области помогут решать проблему эквивалентных преобразований для схем, которые не являются примитивными;
- исследование многовыходных схем как самостоятельного объекта.

Список литературы

[1]. Подловченко Р.И. Исследование примитивных схем программ с процедурами // Моделирование и анализ информационных систем, т. 21, 4, 2014, с. 116–131.

[2]. Подловченко Р.И., Молчанов А.Э. О теории алгебраических моделей программ с процедурами // Моделирование и анализ информационных систем, т.19, №5, 2012, с.100–114.

[3]. Подловченко Р.И. Специальные перегородчато-автоматные модели рекурсивных программ // Программирование, 1994, №3, с.3 - 26.

[4]. Молчанов А.Э. Разрешимость эквивалентности в двухпараметрических перегородчатых моделях программ // Моделирование и анализ информационных систем, т.21, №4, 2014, с. 104-115.

[5]. Подловченко Р.И. Полные системы эквивалентных преобразований в уравновешенных полугрупповых моделях программ с левым сокращением // Программирование, 2010, № 3, с. 3-18.

A Solution to the Equivalent Transformation Problem in a Class of Primitive Program Schemes

A. Molchanov <gurux13@gmail.com>

Faculty of Computatational Mathematics and Cybernetics of Moscow State University, 1b52, Leninskie Gory str., Moscow, 119333, Russian Federation

Abstract. The article belongs to the theory of program schemes. Program schemes are objects designed for the analysis of formalized programs. One of the key tasks of the theory is to create a finite full system of equivalent transformations (E.T.). The article deals with program schemes with procedures, limited only to gateway program models. Such models derive highly from program models without procedures. A brief description of E.T. systems is given, followed by a methodology for the construction of full systems of E.T. A subclass of gateway program models, called primitive schemes, is introduced. Such schemes are closer to procedure-free schemes. The methodology is then successfully applied to construct a full finite system of E.T. in the primitive subclass of balanced gateway semigroup program models with left cancellation, which is a well-studied class of program models. The construction is based on the known system of E.T. for similar program models without procedures. An auxiliary type of schemes, called multiexit schemes, is used. As a result, canonical schemes for classes of equivalence within the models are defined, and a sequence of transformations resulting in the transition of any scheme from this model into its canonical form is stated. This is considered a complete E.T. problem solution for a class of programs. Further research topics are given in the conclusion.

Keywords: algebraic program models; equivalent transformations; gateway program models; primitive program models; left cancellation.

DOI: 10.15514/ISPRAS-2015-27(2)-11

For citation: Molchanov A. A Solution to the Equivalent Transformation Problem in a Class of Primitive Program Schemes. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 173-188 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-11.

References

[1]. Podlovchenko R.I. Primitive program schemata with procedures. *Automatic Control and Computer Sciences, Allerton Press*, 2014, vol. 48, N 7, pp. 615-622.

[2]. Podlovchenko R.I., Molchanov A.E. About algebraic program models with procedures. *Automatic Control and Computer Sciences, Allerton Press*, 2013, vol. 47, N 7, pp. 385-392.

[3]. Podlovchenko R.I. [Special gateway-automaton models of recursive programs]. *Programmirovanie [Programming],* 1994, No. 3, pp. 3-26 (in Russian).

[4]. Molchanov A.E. [Equivalence Problem Solvability in Biparametric Gateway Program Models]. *Modelirovanie I analiz informacionnikh sistem [Information systems analysis and modelling],* 2014, vol.21, No. 4, pp. 104-115 (in Russian).

[5]. Podlovchenko R.I. Complete Systems of Equivalent Transformations in Balanced Semigroup Models of Programs with Left Cancellation. *Programming and Computer Software, 2010, vol. 36, No. 3, pp. 125-137*

Игорь Бурдонов, Александр Косачев. Параллельные вычисления на динамически меняющемся графе. Труды ИСП РАН, том 27, вып. 2, 2015 г., с. 189-220

Параллельные вычисления на динамически меняющемся графе

Игорь Бурдонов <igor@ispras.ru>
Александр Косачев <kos@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Рассматривается задача параллельного вычисления значения функции от мультимножества значений, записанных в вершинах ориентированного сильно-связного графа. Вычисление выполняется автоматами, находящимися в вершинах графа. Автомат имеет локальную информацию о графе: он «знает» только о дугах, выходящих из вершины, в которой он находится, но «не знает», куда (в какие вершины) эти дуги ведут. Автоматы обмениваются сообщениями, передаваемыми по дугам графа, которые играют роль каналов передачи сообщений. Вычисление инициируется сообщением, приходящим извне в автомат выделенной начальной вершины графа. Этот же автомат в конце работы посылает вовне вычисленное значение функции. Для решения этой задачи предлагаются два алгоритма. Первый алгоритм выполняет исследование графа, целью которого является разметка графа с помощью изменения состояний автоматов в вершинах. Такая разметка используется вторым алгоритмом, который и производит вычисление значения той или иной функции. Это вычисление основано на алгоритме пульсации: сначала от автомата начальной вершины по всему графу распространяются *сообщения-вопросы*, которые должны достигнуть каждой вершины, а затем от каждой вершины «в обратную сторону» к начальной вершине двигаются *сообщения-ответы*. Алгоритм пульсации, по сути, вычисляет агрегатные функции, для которых значение функции от объединения мультимножеств вычисляется по значениям функции от этих мультимножеств. Однако показано, что любая функция $F(x)$ имеет агрегатное расширение, то есть может быть вычислена как $H(G(x))$, где G агрегатная функция. Заметим, что разметка графа не зависит от той функции, которая будет вычисляться. Это означает, что разметка графа выполняется один раз, после чего может многократно использоваться для вычисления различных функций. Поскольку автоматы в вершинах графа работают параллельно, как разметка графа, так и вычисление функции выполняются параллельно. Это первая особенность работы. Вторая особенность – вычисления выполняются на динамически меняющемся графе: его дуги могут исчезать, появляться или менять свои конечные вершины. На изменения графа налагаются такие минимальные ограничения, которые позволяют решать эту задачу за ограниченное время. Приводится оценка времени работы обоих предлагаемых алгоритмов.

Ключевые слова: ориентированные графы, исследование графа, взаимодействующие автоматы, параллельная работа, агрегатные функции, динамически меняющиеся графы.

DOI: 10.15514/ISPRAS-2015-27(2)-12

Для цитирования: Бурдонов Игорь, Косачев Александр. Параллельные вычисления на динамически меняющемся графе. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 189-220. DOI: 10.15514/ISPRAS-2015-27(2)-12.

1. Введение

Данная работа находится на пересечении двух направлений исследования, результаты которых опубликованы в статьях [1] и [2].

Первое направление [1] – это параллельные вычисления на ориентированном графе. Такие вычисления являются, по сути, исследованием графа, «нагруженного» дополнительными значениями в вершинах графа.

Исследование ориентированных графов – корневая задача во многих приложениях. Достаточно указать исследование сетей связи, в том числе сети интернета и GRID, и тестирование программных и аппаратных систем, моделируемых графами переходов. Исследование графа, как правило, базируется на его обходе, а это уже старая классическая задача обхода лабиринта. Эта задача нетривиальна, если граф ориентирован, то есть в лабиринте «улицы с односторонним движением».

Обход ориентированного сильно-связного графа требует времени порядка nm, где n – число вершин графа, а m – число дуг. Такое время обхода достигается многими хорошо известными алгоритмами: обход в глубину, обход в ширину, «жадный» алгоритм и др. [3,4,5].

В 1966 г. М.О. Рабин поставил задачу обхода ориентированного графа конечным автоматом [6]. Автомат на графе аналогичен машине Тьюринга: ячейке ленты соответствует вершина графа, а движение влево или вправо по ленте заменяется переходом по одной из дуг, выходящих из текущей вершины графа. На сегодняшний день наиболее быстрый алгоритм предложен в [7], он имеет оценку $nm+n^2 loglogn$. При повторном обходе, когда автомат может использовать пометки, оставленные им же после первого обхода, оценка уменьшается до $nm+n^2 l(n)$, где $l(n)$ — число логарифмирований, при котором достигается соотношение $1 \le log(log...(n)...) < 2$ [8]. Отличие от нижней оценки nm объясняется тем, что автомату бывает нужно «вернуться» в начало только что пройденной дуги.

За последние годы размер реально используемых систем и сетей и, следовательно, размер исследуемых графов непрерывно растёт. Проблемы возникают тогда, когда исследование графа одним автоматом (компьютером) либо требует недопустимо большого времени, либо граф не помещается в памяти одного компьютера, либо и то, и другое. Поэтому возникает задача

параллельного и распределённого исследования графов. Эта задача формализуется как задача исследования графа коллективом автоматов.

В [9] и [10] предложены алгоритмы работы такого коллектива автоматов. При этом предполагается, что автоматы не могут ничего писать в вершины графа или читать из них, но могут обмениваться между собой сообщениями с помощью сети связи, ортогональной графу, а также генерировать новые автоматы. Наилучшая полученная оценка $m+nD$, где D – диаметр графа, т.е. длина максимального пути (маршрута без самопересечений) в графе.

В [1] мы рассматривали классическую задачу исследования графа автоматами, обмен информацией между которыми происходит только через память вершин графа. Это эквивалентно исследованию графа с помощью сообщений, которыми обмениваются между собой автоматы, неподвижно «сидящие» в вершинах графа, а дуги графа играют роль каналов передачи сообщений. Автомат, находящийся в вершине, посылает сообщение по одной из дуг, выходящих из этой вершины, и через какое-то время такое сообщение принимается автоматом в конце дуги. Оценка времени работы алгоритма зависит от числа сообщений, которые могут одновременно передаваться по дуге. Такое число называется ёмкостью дуги и обозначается k. Предполагается, что исследуемый граф сильно связен, а время передачи сообщения по дуге ограничено сверху 1 тактом.

Как алгоритмы исследования графа, так и оценка времени их работы, существенно зависят от того, имеют ли автоматы в вершинах графа какую-то информацию о графе, или каждый автомат находится в начальном состоянии и «ничего не знает» о графе. В [1] мы предложили два алгоритма: алгоритм разметки и алгоритм пульсации. Алгоритм разметки выполняет первичный обхода графа с помощью пересылаемых сообщений, когда в начальный момент времени все автоматы находятся в начальном состоянии. После завершения обхода автоматы остаются в некоторых конечных состояниях, вообще говоря, отличающихся от начального, что, по сути, задаёт разметку графа. Такая разметка позволяет быстрее выполнять параллельное вычисление требуемой функции с помощью алгоритма пульсации. Разметка графа выполняется за время порядка $n/k+D$, а вычисление функции – за время порядка D.

В алгоритме пульсации сначала от автомата выделенной начальной вершины (корня) по всему графу распространяются сообщения-вопросы, которые должны достигнуть каждой вершины. А затем от каждой вершины «в обратную сторону» к корню двигаются сообщения-ответы. С помощью алгоритма пульсации можно параллельно вычислять любую функцию от мультимножества значений, записанных в памяти автоматов по всем вершинам графа (мы будем говорить «записанных в вершинах»). Вот лишь несколько примеров таких функций:

1. Максимум чисел, записанных в вершинах графа.

2. В более общем виде вместо максимума можно использовать любую коммутативную и ассоциативную операцию над числами: минимум, сложение, произведение и т.д.
3. Частные случаи: число вершин в графе, если в каждой вершине записать «1», и число дуг в графе, если в каждой вершине записать число выходящих дуг.
4. Дизъюнкция логических значений, записанных в вершинах графа.
5. В более общем виде вместо дизъюнкции можно использовать любую коммутативную и ассоциативную операцию над логическими значениями: конъюнкцию, эквивалентность и т.д.
6. В ещё более общем виде вместо чисел или логических значений можно использовать любые значения и любые коммутативные и ассоциативные операции над ними.
7. Среднее арифметическое, среднее геометрическое или среднее квадратичное от чисел, записанных в вершинах графа.

Второе направление [2] – это исследование динамически меняющегося графа. Исследование проводится также с помощью автоматов, находящихся в вершинах графа и обменивающихся между собой сообщениями, передаваемыми по дугам графа в направлении их ориентации. Ёмкость дуги предполагается равной 1. Особенность в том, что дуги графа могут динамически изменяться: появляться, исчезать или менять свою конечную вершину. При исчезновении дуги передаваемое по ней сообщение теряется, а при смене конца дуги сообщение попадает в новую конечную вершину.

Понятно, что если время существования дуги до её исчезновения или смены конца слишком мало, сообщение, передаваемое по ней, потеряется или не будет передано в «нужную» вершину. Поэтому хотя бы некоторые дуги должны быть «долгоживущими», чтобы за время существования такой дуги по ней могло пройти хотя бы одно сообщение. Для того чтобы можно было исследовать весь граф, достаточно, чтобы в каждый момент времени такие долгоживущие дуги образовывали сильно связный суграф.

В [2] целью исследования графа назван сбор полной информации о графе в корне графа. Такая информация представляет собой набор описаний всех дуг графа. Однако, поскольку граф динамически меняется, мы не можем гарантировать, что описания текущего состояния всех его дуг отражены в корне: сообщения о последних изменениях дуг могут просто не дойти до корня. Поэтому требуется только, чтобы через время, ограниченное сверху величиной T_0, после изменения дуги корень графа «узнал» об этом или более позднем изменении дуги. Если после данного изменения дуга больше не меняется, по крайней мере, в течение времени T_0, то в корне будет правильное описание этой дуги. Если в какой-то момент времени все изменения в графе прекращаются, то через время, ограниченное сверху величиной T_1, в корне

окажется полное описание состояния графа после прекращения изменений. В [2] предложен алгоритм исследования графа с оценками $T_0=\boldsymbol{O}(n)$ и $T_1=\boldsymbol{O}(D)$, где D – диаметр графа после прекращения изменений.

В данной работе мы попытались совместить идею параллельного вычисления на графе из [1] и идею исследования динамически меняющегося графа из [2]. В разделе 2 кратко излагается теория агрегатных функций и агрегатных расширений функций. Раздел 3 содержит постановку задачи, в том числе, все предположения и ограничения на устройство графа, работу автоматов и передачу сообщений. При этих предположениях и ограничениях в разделе 4 описывается общая идея алгоритма. Формальное описание алгоритма приведено в разделе 5, а доказательство правильности работы алгоритма, а также оценка времени работы алгоритма, размеров памяти автомата и сообщений приведены в разделе 6.

2. Агрегатные функции и агрегатные расширения функций

Алгоритм пульсации, по сути, вычисляет агрегатные функции, для которых значение функции от объединения мультимножеств вычисляется по значениям функции от этих мультимножеств. В [1] мы дали формальное определение агрегатной функции, доказали критерий агрегатности и показали, что любая функция F имеет агрегатное расширение, то есть может быть вычислена как композиция функций HG, где G агрегатная функция. Также показали, что существует и единственное минимальное агрегатное расширение, вычисляющее минимум информации, по которой еще можно восстановить функцию F. Эта теория агрегатных функций является модификацией теории индуктивных функций в [11]. Здесь мы повторим определения и утверждения из [1], опустив доказательства.

Далее рассматриваются функции на конечных мультимножествах из элементов базового множества X. Множество всех конечных мультимножеств из элементов X обозначается, как и выше, через $X^{\#}$. Под операциями объединения, пересечения, дополнения и пр. далее подразумеваются операции на мультимножествах, т.е., учитывающие кратности элементов. Через $\boldsymbol{N}$ обозначим множество натуральных чисел.

Агрегатная функция $G:X^{\#}\to B$ – это такая функция, что

$\exists E:B\times B\to B\ \forall a,b\in X^{\#}\ G(a\cup b) = E(G(a),G(b))$.

Замечание 2.1: Для агрегатной функции G выполнено следующее: $\forall r\in\boldsymbol{N}\ \exists E_r : B^r\to B\ \forall a_1,\ldots,a_r\in X^{\#}\ G(\cup\{a_i \mid 1\le i\le r\}) = E_r(G(a_1),\ldots,G(a_r))$.

Утверждение 2.1: Функция $G:X^{\#}\to B$ является агрегатной тогда и только тогда, когда $\forall a,b\in X^{\#}\ \forall x\in X\ G(a) = G(b) \Rightarrow G(a\cup\{x\}) = G(b\cup\{x\})$.

Замечание 2.2: Из утверждения 2.1 следует, что $G : X^{\#} \to B$ является агрегатной тогда и только тогда, когда $\exists G` : B \times X \to B \ \forall a \in X^{\#} \ x \in X \ G(a \cup \{x\}) = G`(G(a), x)$. Действительно, достаточно определить $G`(G(a), x) = E(G(a), G(\{x\}))$.

Агрегатным расширением функции $F:X^{\#} \to A$ назовём агрегатную функцию $G:X^{\#} \to B$, такую, что $\exists H : B \to A \ \forall a \in X^{\#} \ F(a) = H(G(a))$.

Агрегатное расширение $G : X^{\#} \to B$ функции $F : X^{\#} \to A$ назовём *минимальным*, если $G(X^{\#}) = B$ и $\forall G` : X^{\#} \to C$ являющего агрегатным расширением $F$ имеет место $\exists i : C \to B \ G = iG`$.

Замечание 2.3: Агрегатное расширение G функции F представляет собой агрегатную функцию, по которой можно вычислить функцию F. При этом возможны такие расширения, которые на практике не помогают в этом, например, можно взять в качестве G тождественную функцию на $X^{\#}$, а в качестве H – саму функцию F. Чтобы избежать такого, используется минимальное агрегатное расширение; интуитивно, это агрегатная функция, вычисляющая минимум информации, по которой еще можно восстановить F.

Утверждение 2.2: Минимальное агрегатное расширение любой функции $F : X^{\#} \to A$ существует и единственно с точностью до взаимно однозначных отображений.

Замечание 2.4: Примеры минимального агрегатного расширения (π_i – проекция кортежа на i-ый компонент).

- Для функции вычисления среднего арифметического $F(a_1,\dots,a_n) = (a_1+\dots+a_n)/n$ минимальным агрегатным расширением является функция $G(a_1,\dots,a_n) = (a_1+\dots+a_n, n)$; $F = \pi_1 G / \pi_2 G$.
- Для функции вычисления среднего геометрического $F(a_1,\dots,a_n) = \sqrt[n]{(a_1 \cdot \dots \cdot a_n)}$ минимальным агрегатным расширением является функция $G(a_1,\dots,a_n) = (a_1 \cdot \dots \cdot a_n, n)$; $F = \sqrt[\pi_2 G]{\pi_1 G}$.
- Для функции вычисления среднего квадратичного $F(a_1,\dots,a_n) = \sqrt{((a_1^2+\dots+a_n^2)/n)}$ минимальным агрегатным расширением является функция $G(a_1,\dots,a_n) = (a_1^2+\dots+a_n^2, n)$; $F = \sqrt{(\pi_1 G / \pi_2 G)}$.

Итак, каждую функцию $F:X^{\#} \to A$ можно представить в виде $F(a) = H(G(a))$ и $\forall b,c \in X^{\#} \ G(b \cup c) = E(G(b),G(c))$. *Разбиением* мультимножества $b \in X^{\#}$ называется набор $b_1,\dots,b_r$ его подмножеств, объединение которых совпадает с b: $b=b_1 \cup \dots \cup b_r$. Будем говорить, что задано *вложенное разбиение* мультимножества $b \in X^{\#}$, если задано его разбиение $b_1,\dots,b_r$ и для каждого не синглетонного (содержащего более одного элемента) мультимножества b_i также задано вложенное разбиение. Тогда, если задано вложенное разбиение мультимножества $a \in X^{\#}$, то вычисление функции $F(a)$ можно выполнить следующим образом. Сначала вычисляются значения $G(x)$ для каждого $x \in a$ (без учёта кратности). Далее, учитывая замечание 2.1, для каждого элемента

$b=b_1\cup...\cup b_r$ вложенного разбиения вычисляется значение $G(b) = E_r(G(b_1),...,G(b_r))$. При этом сама функция E_r может вычисляться итеративно с помощью функции E: для $r>2$ имеем $E_r(G(b_1),...,G(b_r)) = E(E_{r-1}(G(b_1),...,G(b_{r-1})),G(b_r))$. После того, как будет получено значение $G(a)$, вычисляется искомый результат $F(a) = H(G(a))$.

3. Постановка задачи

Постановка задачи совмещает предположения и ограничения, сформулированные в [1] и [2], с тремя отличиями, которые мы специально оговорим.

Рассматривается ориентированный граф с n вершинами и m дугами. Множество вершин обозначим через V, одна из вершин $v_0\in V$ выделена и называется *корнем*. Пусть также задан некоторый алфавит X, множество всех конечных мультимножеств из элементов X обозначим через $X^\#$. Будем считать, что задана функция $q : V\to X$. Мультимножество, образуемое значениями $q(v)$ во всех вершинах графа, обозначим через $q(V)\in X^\#$. Для любой функции $F:X^\#\to A$ требуется вычислить значение $F(q(V))$. Как показано в разделе 2 функцию можно представить в виде $F=HG$, где $G : X^\#\to B$ – минимальное агрегатное расширение, и $H : B\to A$. При этом существует такая функция $E : B\times B\to B$, что $\forall a,b\in X^\#\ G(a\cup b) = E(G(a),G(b))$.

Вычисления выполняются автоматами, находящимися в вершинах графа и обменивающихся между собой сообщениями по дугам графа. Иногда для краткости мы будем вместо «автомат вершины» писать просто «вершина», если это не приводит к недоразумениям. Автомат в вершине может послать сообщение по любой из выходящих дуг и принять сообщение по любой из входящих дуг. Автомат начинает работать, получая первое сообщение по входящей дуге. Корень получает первое сообщение «*Старт*» извне графа, это сообщение инициирует работу алгоритма разметки; в ответном сообщении «*Готов к работе*» корень извещает о завершении разметки и готовности к вычислению функции.

Вычисление функции выполняется в алгоритме пульсации. Оно также инициируется извне с помощью сообщения «*Вопрос*», направляемого в корень и содержащего описание функции F: H,G,E. Когда корень вычисляет значение функции $F(q(V))$, он посылает вовне сообщение «*Ответ*», содержащее вычисленное значение и означающее готовность принять следующий вопрос. Мы будем предполагать, что сообщения извне приходят только в корень, а не в какую-нибудь другую вершину.

Автомат в вершине v может узнать значение $q(v)$ с помощью примитива «*Дай значение*». Каждая вершина имеет уникальный (среди вершин) *идентификатор вершины*. Автомат может узнать идентификатор своей вершины с помощью примитива «*Дай идентификатор*».

Все дуги, выходящие из вершины, перенумерованы, начиная с номера 1 до наибольшего номера, не превосходящего некоторого числа *s*. Очевидно, $s \le m$. Дуга однозначно идентифицируется парой (идентификатор начала дуги, номер дуги), которую мы будем называть *идентификатор дуги*. Число дуг *m* – это число различных идентификаторов дуг в графе.

Автомат, посылая сообщение по дуге, указывает её номер. В отличие от [1] и так же как в [2], мы будем считать, что ёмкость дуги *k=1*. Это первое отличие. Оно объясняется тем, что граф может динамически меняться, и гарантируется передача по дуге только одного сообщения, после чего дуга может измениться, и даже это гарантируется не для всех дуг, как описано ниже. Поэтому в наихудшем случае, для которого и делается оценка времени работы алгоритма, случаи *k>1* и *k=1* не отличаются. Иными словами, даже если *k>1*, автомат не будет посылать по дуге сообщение до тех пор, пока предыдущее сообщение продолжает передаваться по дуге. Будем говорить, что дуга занята, если по ней передаётся сообщение. Иначе дуга свободна. Сообщение посылается только по свободной дуге. В самом начале все дуги свободны. Когда сообщение передано по дуге, начало дуги извещается об этом сигналом *освобождения* дуги (*сигнал О*) с параметром номер дуги.

Граф может динамически меняться, причём вершины не меняются, а дуги могут изменяться следующим образом:

- Дуга может появиться, о чём начало дуги извещается сигналом *появления* дуги (*сигнал П*) с параметром номер дуги.
- Дуга может исчезнуть. Если по дуге передавалось сообщение, то оно пропадает, и в этом случае начало дуги извещается сигналом *исчезновения* дуги (*сигнал И*) с параметром номер дуги. Заметим, что если по дуге никакого сообщения не передавалось, то сигнала И не будет; однако если вершина попытается послать сообщение по исчезнувшей дуге, сообщение не будет передано, а вершина получит сигнал И.
- Дуга может поменять свой конец; в этом случае никаких сигналов не предусмотрено.

Из-за того, что дуги могут исчезать и появляться, номера дуг, выходящих из одной вершины и имеющихся (появившихся и не исчезнувших) в данный момент времени, могут идти не подряд, т.е. не образовывать отрезок натурального ряда, но в любом случае они располагаются на отрезке от 1 до *s*.

Итак, если не считать идентификатора вершины *v* и значения *q(v)*, которые автомат получает с помощью соответствующих примитивов, входными символами автомата, находящегося в вершине, являются сигналы от дуг, выходящих из вершины, и сообщения, получаемые по дугам, входящим в вершину. Срабатывание автомата – это обработка одного такого входного символа. Мы будем предполагать, что входные символы, как правило,

поступают в автомат в порядке их возникновения. Это означает, что существует очередь входных символов автомата.

Сигнал О вырабатывается не тогда, когда сообщение ставится в очередь входных символов, а тогда, когда оно выбирается из этой очереди автоматом конца дуги. Поэтому только после того, как сообщение будет принято автоматом конца дуги, по этой дуге будет послано следующее сообщение, а ещё позже оно будет поставлено в очередь в конце дуги. Следовательно, в очереди входных символов может быть не более одного сообщения для каждой входящей дуги.

Если сигнал от некоторой выходящей дуги вырабатывается в тот момент времени, когда предыдущий сигнал от этой дуги ещё находится в очереди входных символов, то новый сигнал не ставится в конец очереди, а замещает собой старый сигнал. Можно отметить, что таким новым сигналом может быть только сигнал П.

Исключением является случай, когда старый сигнал – это сигнал О. В отличие от [2] сигнал О не замещается сигналом П: в очереди остаётся сигнал О, а сигнал П игнорируется. Это второе отличие. В [2] замещение сигнала О сигналом П информировало автомат начала дуги о том, что дуга изменилась: исчезла и снова появилась. Это было важно, поскольку ставилась цель мониторинга динамического графа, т.е. нужно было отслеживать все изменения его дуг. В данной работе у нас нет такой цели, зато нужно, чтобы автомат в начале дуги мог узнать, дошло сообщение, посланное по дуге, до её конца или пропало. Это используется в алгоритме разметки для того, чтобы определить момент завершения сбора информации о всех вершинах графа (подробнее смотри ниже в подразделе 4.3). Если как в [2] сигнал О замещается сигналом П, автомат в начале дуги не различает следующие два случая. 1) Сообщение дошло до конца дуги, сгенерирован сигнал О; до его обработки дуга исчезает (сигнала И нет, так как на дуге нет сообщения), а затем появляется: сигнал П замещает сигнал О. 2) Сообщение пропадает, так как дуга исчезает. Генерируется сигнал И. До его обработки дуга снова появляется, и сигнал П замещает собой сигнал И. В обоих случаях автомат начала дуги получит сигнал П, однако в 1-ом случае сообщение дошло до конца дуги, а во 2-ом случае пропало. Итак, в [2] замещение сигналов происходит по схеме «П+П→П, И+П→П, О+П→П», а в данной работе – по схеме «П+П→П, И+П→П, О+П→О».

Заметим, что если бы нам нужно было отслеживать все изменения дуг как в [2] и гарантированно определять прохождение сообщения по дуге, как в данной работе, то нужно было бы сохранить оба сигнала: О и П. Сигнал О при появлении сигнала П нужно было бы замещать новым сигналом «*освобождение+появление*», который информировал бы начало дуги как о том, что сообщение дошло до конца дуги, так и о том, что после этого дуга изменилась: исчезла и снова появилась.

Из-за замещения сигналов от одной выходящей дуги в очереди входных символов может быть не более одного сигнала для каждой выходящей дуги. Тем самым, длина очереди входных символов автомата не превосходит степени вершины, то есть не больше $2m$ ($2m$ достигается для $n=1$, когда все дуги – петли, поскольку петля считается два раза: как входящая – для сообщений и как выходящая – для сигналов).

Обозначим через T_{ab} максимальное время, за которое информация, имеющаяся в некоторой вершине a, доходит до вершины b, и $T = max\{T_{ab} \mid a,b \in V\}$. Для того чтобы время T было конечным, нужно, чтобы сообщения могли распространяться по графу, доходя от каждой вершины a до каждой вершины b. Для этого нужно, чтобы время x пересылки сообщения по дуге и время y срабатывания автомата были конечными. Кроме того, *время z существования* дуги, т.е. время от появления дуги или изменения её конца до исчезновения дуги или изменения её конца, должно быть достаточно велико, чтобы по дуге успевало пройти хотя бы одно сообщение. Такими «долгоживущими» дугами могут быть не все дуги: достаточно, чтобы в каждый момент времени «долгоживущие» дуги порождали сильно связный суграф (подграф, содержащий все вершины графа).

Для ограниченности времени T необходимо, чтобы x и y были ограничены сверху: $x \le X$ и $y \le Y$, а z – снизу: $z \ge Z$. Выразим эту нижнюю границу Z через X и Y. Сообщение можно посылать по дуге сразу после того, как она появилась или освободилась (сигналы П, О). Однако в этот момент времени сигнал П или О ещё только ставится во входную очередь символов автомата начала дуги, длина которой может достигать $2m$. Поэтому сообщение посылается по дуге не сразу, а через время, которое может достигать $2mY$. Следовательно, сообщение дойдёт до конца дуги и будет поставлено во входную очередь символов автомата конца дуги через время, которое может достигать $2mY+X$, если в течение этого времени дуга не менялась. Поэтому будем предполагать, что нижняя граница времени существования «долгоживущих» дуг $Z \ge 2mY+X$.

Для оценки времени работы алгоритма мы будем для простоты предполагать, что временем срабатывания автомата можно пренебречь, то есть $Y=0$, время пересылки по дуге не превосходит 1 такта, то есть $X=1$, а время существования «долгоживущей» дуги может быть минимально, то есть $Z=2mY+X=1$.

Также как в [2] предполагается, что в начальный момент времени в каждой вершине имеется автомат в начальном состоянии, и для каждой дуги, имеющейся в графе в начальный момент времени, выработан сигнал П. Третье отличие от [2] – понятие начальной дуги и предположение о начальных дугах. Начальной дугой будем называть такую дугу $a \to b$, которая существует в начальный момент времени и которая не меняется до тех пор, пока по ней не пройдёт первое сообщение от a до b. Будем предполагать, что каждая вершина

достижима из корня по начальным дугам. Начальные дуги нужны для того, чтобы в алгоритме разметки определять момент завершения сбора информации о всех вершинах графа. Подробнее об этом будет рассказано ниже при описании алгоритма разметки.

4. Идея алгоритма

4.1. Распространение информации по динамическому графу

Предположение о долгоживущих дугах гарантирует возможность доставки информации из любой вершины в любую вершину, но только при условии, что автоматы в вершинах соблюдают определённую дисциплину передачи сообщений.

1) Каждая вершина, получив сообщение с данной информацией, должна сохранить её в своей памяти и далее помещать во все сообщения, которые постоянно нужно посылать по всем выходящим дугам, когда появляется такая возможность (по сигналам О, П).

Казалось бы, информацию можно не посылать по дуге повторно, однако пример на рис. 1 показывает, что в этом случае нет гарантии доставки информации. Здесь затемнёнными кружками показаны вершины, в которые поступила новая информация, двойной линией показаны дуги, по которым информация уже была отправлена и дошла до конца дуги. При переходе от п.4 к п.5 дуги 1 и 2 меняют свои концы, но по ним информация повторно не посылается. Из-за этого информация не достигает вершины *c*.

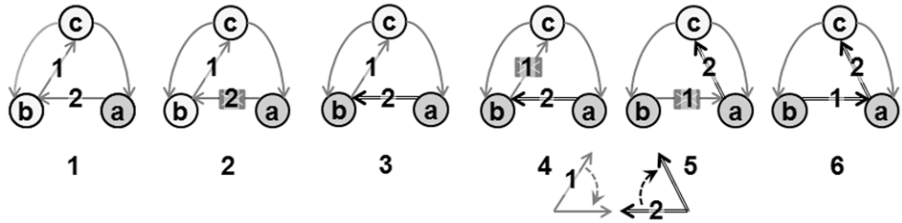

Рис. 1. Пример нераспространения информации по графу.

2. Из 1-го требования непосредственно следует, что если из вершины нужно передать в какие-то другие вершины несколько порций информации, то они должны посылаться не последовательными сообщениями, а помещаться в одно сообщение.

Эти требования достаточны для гарантированной доставки информации из каждой вершины в каждую вершину, что строго показано в [2] в несколько изменённой форме (Лемма 5). Здесь мы приводим доказательство аналогичного утверждения в нужной нам форме.

Лемма 1. Пусть в некоторый момент времени в некоторой вершине имеется новая информация. Тогда не более чем через $3(n-1)$ тактов эта информация будет доставлена в каждую вершину.

Доказательство. Пусть в некоторый момент времени $A \neq \varnothing$ – это множество вершин, в которых уже есть нужная информация. В каждый момент времени сураф, порождённый «долгоживущими» дугами, сильно связен, время существования «долгоживущей» дуги не меньше 1 такта, время пересылки по дуге не больше 1 такта, а временем срабатывания автомата мы пренебрегаем. Кроме того, в каждой вершине при появлении или освобождении дуги по ней сразу посылается сообщение.

Докажем *утверждение о расширении* множества A: с момента времени t_1, когда образуется множество $A \neq \varnothing$ и $A \neq V$, не более чем через 3 такта множество A будет расширено.

Очевидно, что множество A может только расширяться (не может сужаться). В любой момент времени, в том числе в момент времени t_1+2, должна существовать «долгоживущая» дуга p, ведущая из вершины $a \in A$ «наружу», т.е. в вершину $b \notin A$. В момент времени t_1+2 по дуге p передаётся сообщение S. Это сообщение отправлено из a не ранее момента времени $(t_1+2)-1=t_1+1$, поэтому S содержит нужную информацию. Возможны два случая.

1) Сообщение S дойдёт до вершины b. Тогда это произойдёт не позднее момента времени $(t_1+2)+1=t_1+3$. Следовательно, в этом случае не более чем через 3 такта после момента времени t_1 множество A будет расширено.
2) Сообщение S не дойдёт до вершины b. Это означает, что дуга p изменится до того, как по ней до вершины b дойдёт сообщение S. А тогда, поскольку p – «долгоживущая» дуга, по ней должно дойти до вершины b предыдущее сообщение S`. И это должно произойти до момента времени $t_1+2$. Поскольку сообщение $S$ отправлено из $a$ не ранее момента времени $t_1+1$, сообщение $S$` принято в b не ранее этого же момента времени. Следовательно, сообщение S` отправлено из $a$ не ранее момента времени $(t_1+1)-1=t_1$, поэтому $S$` содержит нужную информацию. Следовательно, в этом случае не более чем через 2 такта после момента времени t_1 множество A будет расширено.

При каждом расширении множества A в него добавляется хотя бы одна вершина. Поскольку с самого начала множество A содержит хотя бы одну вершину, а общее число вершин равно n, получается, что число расширений множества A не более $n-1$. Таким образом, требуется не более чем $3(n-1)$ тактов, чтобы в каждой вершине оказалась нужная информация.

Лемма доказана.

4.2. Оптимальная разметка графа

В [1] разметка графа состояла из двух остовных (содержащих все вершины графа) деревьев: *прямое дерево*, ориентированное от корня, и *обратное дерево*, ориентированное к корню. В каждой вершине (кроме листовых вершин прямого дерева) отмечены выходящие прямые дуги, т.е. дуги прямого дерева. Также в каждой вершине, кроме корня, отмечена одна выходящая обратная дуга, т.е. дуга обратного дерева. Кроме того, в каждой вершине устанавливался счётчик числа входящих обратных дуг. В алгоритме пульсации сообщение-вопрос, содержащее описание функции, распространялось от корня по прямому дереву. Обратное дерево задавало вложенное разбиение мультимножества $q(V)$, т.е. было предназначено для сбора ответов.

Листовая вершина обратного дерева, получив вопрос, сразу же вычисляет свой ответ и посылает его по выходящей обратной дуге. Внутренняя (не листовая) вершина обратного дерева сначала собирает сообщения-ответы по всем входящим в неё обратным дугам, после чего вычисляет свой ответ и посылает его по выходящей обратной дуге. Корень посылает ответ вовне графа.

Такой алгоритм позволял распространять по графу вопрос за время $\boldsymbol{O}(h_0)$, где h_0 – высота прямого дерева, после чего вычислять функцию, собирая все ответы, за время $\boldsymbol{O}(h)$, где h – высота обратного дерева. Однако если граф динамически меняется, то возникает проблема: изменение графа может потребовать изменения прямого или обратного дерева, если меняется (исчезает или меняет свой конец) дуга дерева. В общем случае такое изменение может потребовать полной перестройки деревьев, т.е. фактически понадобится строить эти деревья заново. Кроме того, не гарантируется доставка сообщений по дугам построенного дерева: эти дуги могут исчезать или менять свой конец до того, как по ним будет передано нужное сообщение.

В то же время предположение о долгоживущих дугах гарантирует возможность доставки информации из любой вершины в любую вершину. Для этого, как показано в подразделе 4.1 и в [2], каждая вершина, получив сообщение с этой информацией, должна сохранить её в своей памяти и далее помещать во все сообщения, которые постоянно нужно посылать по всем выходящим дугам, когда появляется такая возможность (по сигналам О, П).

Тогда вместо дуги $a \to b$ дерева можно использовать просто пару вершин (a,b), которую назовём виртуальной дугой. Информация, которую нужно переправить по виртуальной дуге, помещается в сообщение, которое отправляется из вершины a и распространяется «веером» по всем дугам, а когда сообщение с этой информацией попадает в вершину b, вершина получает требуемую информацию. Правда, время перемещения сообщения по дуге равно $\boldsymbol{O}(1)$, а по виртуальной дуге – $\boldsymbol{O}(n)$. Соответственно, вместо дерева

используется виртуальное дерево как набор виртуальных дуг. Вместо оценок $\boldsymbol{O}(h_0)$ и $\boldsymbol{O}(h)$ получатся оценки $\boldsymbol{O}(nh_0)$ и $\boldsymbol{O}(nh)$.

Сообщение-вопрос содержит одну и ту же информацию, предназначенную всем вершинам: функции *G* и *E* (функция *H* нужна только в корне для окончательного вычисления значения функции *F*). Такое сообщение может распространиться из корня по всему графу без использования прямого дерева за время $\boldsymbol{O}(n)$, что лучше, чем $\boldsymbol{O}(nh_0)$. Поэтому виртуальное прямое дерево не требуется.

Однако ответы, вычисляемые разными вершинами, разные. Кроме того, как показано в подразделе 4.1, все ответы, которые должны быть отправлены из вершины (ответ, вычисленный в ней, или проходящий «транзитом») должны отправляться в одном сообщении, и в этом же сообщении следует отправлять вопрос. Только в этом случае гарантируется доставка всей требуемой информации адресатам.

В то же время использование виртуального обратного дерева позволяет передавать в одном сообщении не все ответы. Дело в том, что если вершина *a* уже получила ответ от вершины *b* (или вычислила его сама при *b=a*) и получает сообщение с ответом от вершины *c*, расположенной на той же ветви обратного дерева, что вершина *b*, но *ниже* (ближе к корню) вершины *b*, то ответ от вершины *b* (вычисленное ею значение функции *G*) уже не нужен, поскольку он уже использован при вычислении ответа от *c*. Аналогично, если вершина *a* получает сообщение с ответом от вершины *c*, расположенной на той же ветви обратного дерева, что вершина *b*, но *выше* (дальше от корня) вершины *b*, то ответ от вершины *c* (вычисленное ею значение функции *G*) уже не нужен, поскольку он уже использован при вычислении ответа от *b*. Это означает, что в одном сообщении достаточно иметь не более одного ответа от вершин на одной ветви обратного дерева. Если есть ответ хотя бы от одной вершины ветви дерева, то достаточно ответа от самой нижней (ближней к корню) вершины.

Таким образом, наличие виртуального обратного дерева позволяет уменьшить размер сообщения. Без обратного дерева можно было бы собирать в корне просто мультимножество значений функции *q*, получая от каждой вершины v_i, где $i=1..n$, значение $q_i=q(v_i)$, и все вычисления производить в корне $F=H(E_n(G(q_1),...,G(q_n)))$. Однако значение $q(v_i)$ может быть слишком большим, например, это может быть база данных большого размера. Можно было бы также собирать в корне мультимножество значений композиции функций *Gq*, получая от каждой вершины v_i, где $i=1..n$, значение $G_i=G(q_i)$, и аналогично все вычисления производить в корне $F=H(E_n(G_1,...,G_n))$. Однако в этом случае в одном сообщении могут оказаться значения G_i для всех вершин v_i, кроме корня, т.е. до *n-1* значения функции *G*.

Дерево определяется двумя параметрами: высотой *h* и шириной (числом листовых вершин = число ветвей от листа до корня, причём корень листом не

считается) *w*. Высота *h* влияет на время работы алгоритма пульсации, а ширина *w* – на размер сообщения. Возникает задача: для данного числа вершин *n* определить минимальную высоту дерева *h* при заданной ширине $w \le n-1$ и описать деревья с такой минимальной высотой. Очевидно, что $n \le hw+1$. Отсюда $h \ge (n-1)/w$. Следовательно, минимальная высота $h =](n-1)/w[$ (ближайшее сверху целое число). Это достигается для деревьев вида, изображённого на рис. 2, который можно назвать «сбалансированный веник».

Заметим, что если разрешить задавать ширину $w>n-1$, то реальная ширина дерева всё равно будет не больше $n-1$. В частности, если $n=1$, т.е. граф содержит одну вершину, то дерево будет иметь фиксированную ширину $w=0$ и высоту $h=0$.

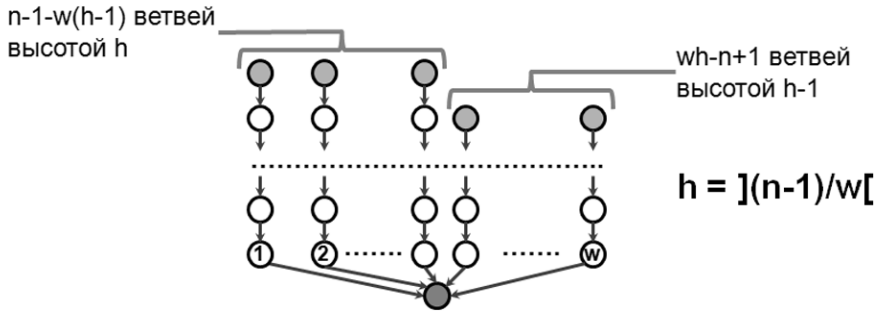

Рис. 2. «Сбалансированный веник»

Для «сбалансированного веника» внутренняя вершина (не лист и не корень) имеет только одну входящую дугу, поэтому для неё не требуется счётчика входящих дуг. Такой счётчик нужен только для корня, его значение равно *w*. Некорневой вершине достаточно «знать», является она листовой (счётчик равен *0*) или внутренней (счётчик равен *1*) вершиной дерева.

В [1] разметка графа при $k=1$ занимала время $\boldsymbol{O(n)}$. При этом можно было использовать относительно короткие сообщения, поскольку необходимую информацию можно было доставлять в вершины не одним, а несколькими сообщениями. Правда, при этом строились не оптимальные прямое и обратное деревья, а те, которые «получались» при данном алгоритме разметки и данном графе. Как показано в подразделе 4.1, в динамическом графе мы вынуждены передавать всю необходимую информацию в одном сообщении, размер которого, тем самым, существенно больше. За счёт этого мы можем за то же по порядку время $\boldsymbol{O(n)}$ строить оптимальное обратное виртуальное дерево. Как показано выше, такое дерево имеет вид «сбалансированного веника», а прямое виртуальное дерево излишне.

Обратное виртуальное дерево вида «сбалансированного веника» (далее просто дерево) можно описать с помощью двухуровневых индексов вершин. Индекс

состоит из номера ветви дерева от *1* до *w* и из номера вершины на ветви от *1* до *h* или *h-1*, считая, что корень имеет номер *0* на любой ветви. Таким образом, корень имеет *w* индексов вида *(i,0)*, а другая вершина имеет только один индекс. Дерево описывается как множество описаний некорневых вершин. Описание некорневой вершины содержит тип вершины «лист» или «внутренняя», идентификатор вершины и её индекс. Виртуальная дуга $a \to b$ существует тогда и только тогда, когда *a* имеет индекс *(i,j)*, $j>0$ и *b* имеет индекс *(i,j-1)*.

Алгоритм разметки выполняется в два этапа. На первом этапе в корне графа собирается информация о всех вершинах. После этого корень строит в своей памяти описание дерева, которое на втором этапе рассылается во все вершины.

4.3. Разметка графа: сбор информации о вершинах

Разметка графа начинается с получения извне сообщения «Старт» с параметром *w*, которое должно приходить в корень по предположению из раздела 3. На этапе сбора информации о вершинах графа по графу циркулируют сообщения типа *1*. Вершина графа и каждое сообщение содержат множество описаний дуг.

Описание дуги создаётся в её начале при появлении дуги (по сигналу П), но только для тех дуг, которые появляются до прихода в вершину первого сообщения. Предположение о начальных дугах в разделе 3 гарантирует, что будут созданы описания всех начальных дуг. При получении сообщения вершина переписывает к себе отсутствующие у неё описания дуг из сообщения, или корректирует имеющиеся у неё описания дуг. Посылая сообщение, вершина помещает в него всё накопленное в ней множество описаний дуг. Тем самым, сообщение аккумулирует в себе описания дуг из всех вершин, через которые оно проходит.

Описание дуги содержит: идентификатор дуги (идентификатор начала дуги и номер дуги в начале дуги) и статус дуги. Статус дуги принимает три значения: 1,2,3. Схема изменения статуса дуги $a \to b$ на рис. 3 иллюстрирует следующие правила:

Правило 1. *Нет описания дуги* $\to 1$ (сплошная линия на рис.3). Описания дуги $a \to b$ со статусом *1* создаётся в вершине *a* по сигналу П ещё до того, как в вершину *a* поступит первое сообщение. Предположение о существовании начальных дуг в начальный момент времени (раздел 3) гарантирует, что в вершине *a* будут созданы описания всех начальных дуг, выходящих из *a* (хотя не обязательно только начальных). Поэтому первое и, следовательно, каждое сообщение, посылаемое из вершины *a*, содержит описания этих дуг.

Правило 2. Изменение статуса дуги $1 \to 2$ (сплошная линия на рис.3) происходит в вершине a, когда вершина a по сигналу О узнаёт о том, что первое сообщение, посланное по дуге $a \to b$, дошло до её конца, вершины b.

Правило 3. Изменение статуса дуги $1 \to 3$ (сплошная линия на рис.3) происходит в вершине a, когда вершина a по сигналу И или П узнаёт о том, что первое сообщение, посланное по дуге $a \to b$, не дошло, или, по повторному сигналу П, узнаёт о том, что дуга повторно появилась (и, следовательно, исчезала) до посылки по ней первого сообщения. В обоих случаях дуга не является начальной.

Правило 4. Такое же изменение статуса дуги $1 \to 3$ (сплошная линия на рис.3) происходит в вершине b, когда вершина b получает по дуге $a \to b$ сообщение, в котором статус дуги $a \to b$ равен 1 (это будет первое сообщение, посылаемое по дуге $a \to b$).

Правило 5. Изменение статуса дуги происходит также в любой вершине при получении сообщения, в котором эта дуга имеет статус больше, чем в вершине или в вершине отсутствует описание дуги (сплошная или пунктирная стрелка на рис.3).

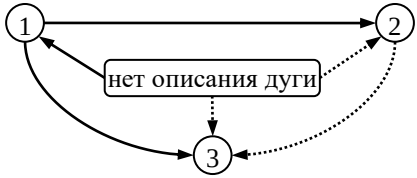

Рис. 3. Схема изменения статуса дуги

Корень получает информацию о вершине a, когда получает сообщение, содержащее описание некоторой дуги $a \to b$, в котором качестве идентификатора начала дуги указан идентификатор вершины a. Если статус дуги в корне равен 3, то это означает одно из двух. 1) Статус 3 установлен в конце дуги, вершине b, и сообщение из b дошло до корня. А тогда корень имеет информацию хотя бы об одной дуге, выходящей из b, тем самым корень знает о вершине b. 2) Статус 3 установлен в начале дуги, вершине a, следовательно, эта дуга не начальная.

Как сказано выше, предположение о существовании начальных дуг в начальный момент времени гарантирует, что корень, узнавая о вершине (т.е. хотя бы об одной выходящей из неё дуге), узнаёт о всех выходящих из неё дугах, которые были в начальный момент времени. Предположение о том, что любая вершина достижима из корня по начальным дугам, гарантирует, что до каждой вершины от корня по начальным дугам дойдёт сообщение, и это сообщение будет первым сообщением, передаваемым по этим дугам. Предположение о долгоживущих дугах гарантирует, что до корня дойдут все сообщения, посылаемые из вершин графа. Поэтому, когда в корне статус всех

дуг станет равным *3*, это будет означать, что корень узнал о всех вершинах графа. На этом первый этап заканчивается и начинается второй этап рассылки описаний виртуального обратного дерева.

4.4. Разметка графа: рассылка описания дерева

Когда информация о всех вершинах собрана в корне, корень строит виртуальное обратное дерево как множество описаний некорневых вершин (подраздел 4.2). Цель второго этапа разметки графа – доставить в каждую некорневую вершину её двухуровневый индекс и её тип «лист» или «внутренняя».

Для этого по графу распространяются сообщения типа 2, содержащие множества описаний вершин, из которых каждая вершина запоминает своё описание и удаляет его из множества.

Когда вершина *a* (не корень) первый раз получает сообщение типа *2*, она переписывает в свою память всё множество описаний вершин из сообщения. В этом множестве должно быть и описание вершины *a*, которое определяется по идентификатору вершины *a*. Из этого описания запоминается индекс и тип вершины *a*, а само описание удаляется из множества. После этого вершина игнорирует все принимаемые сообщения типа *1*, которые ещё могут оставаться на графе.

В дальнейшем, когда вершина *a* получает сообщение типа *2*, она строит пересечение множеств описаний вершин из сообщения и из своей памяти. Это пересечение сохраняется в памяти вершины *a*.

Каждый раз, когда по дуге, выходящей из *a*, можно послать сообщение (по сигналу О или П), вершина *a* помещает в это сообщение то подмножество описаний вершин, которое хранится в её памяти.

Конец рассылки описания дерева определяет корень, когда хранящееся в нём множество описаний вершин становится пустым.

Поскольку каждая вершина удаляет из множества описаний вершин своё описание, а при получении сообщения строится пересечение множеств, рано или поздно такое множество в некоторой вершине становится пустым. В этот момент времени все вершины получили предназначенную им информацию. Далее это пустое множество, двигаясь по графу, «опустошает» множества, хранящиеся в вершинах. Когда «опустошится» множество в корне, этап закончен, корень посылает вовне сообщение «*Готов к работе*».

4.5. Вычисление функции

Вычисление функции *F* начинается с получения корнем извне сообщения «Вопрос» с параметрами *H,G,E*. Функция *H* требуется только корню, а остальным вершинам вопрос доставляется как пара функций *(G,E)*.

Корень нумерует вопросы для того, чтобы при вычислении ответа на вопрос игнорировать все сообщения, относящиеся к вычислению ответов на предыдущие вопросы, поскольку они могут циркулировать по графу даже после окончания вычисления.

Виртуальное обратное дерево задаёт вложенное разбиение мультимножества $q(V)$, как описано в разделе 2 и подразделе 4.2.

Листовая вершина v с индексом (i,j), получив вопрос, сразу же вычисляет ответ $G(q(v))$ и должна передать его по виртуальной обратной дуге, т.е. в вершину с индексом $(i,j-1)$.

Внутренней вершине v дерева типа «сбалансированный веник» соответствует элемент $b=b_1 \cup b_2$ вложенного разбиения, где $b_1=q(D_v)$, где D_v – множество вершин, расположенных на дереве выше v, а $b_2=q(v)$. Вершина v сначала получает ответ $G(b_1)$ по входящей в неё виртуальной обратной дуге, то есть от вершины с индексом $(i,j+1)$. После этого вершина v вычисляет свой ответ $G(b) = E(G(b_1),G(b_2))$ и должна передать его по виртуальной выходящей обратной дуге, то есть вершине с индексом $(i,j-1)$.

Корню v_0 соответствует элемент $b=b_1 \cup ... \cup b_{w+1}$ вложенного разбиения, где w равно числу входящих в корень виртуальных обратных дуг, а $b_{w+1}=q(v_0)$. Корень сначала собирает ответы $G(b_1),...,G(b_w)$ по всем входящим в него виртуальным обратным дугам, то есть ответы от вершин с индексами $(1,1),...,(w,1)$. После этого корень вычисляет $G(b) = E_{w+1}(G(b_1),...,G(b_w),G(b_{w+1}))$. При этом сама функция E_{w+1} вычисляется итеративно с помощью функции E: для $j>1$ имеем $E_{j+1}(G_1,...,G_{j+1}) = E(E_j(G_1,...,G_j), G_{j+1})$. Дополнительно корень применяет функцию H, т.е. вычисляет $H(G(b))$, и посылает полученное значение вовне графа в сообщении «Ответ».

Как показано в подразделах 4.1 и 4.2, при вычислении используется сообщение, которое должно содержать всю распространяемую по графу информацию: вопрос (G,E), номер вопроса и все нужные ответы. Такое сообщение имеет тип 3. При этом достаточно, чтобы сообщение содержало не более одного ответа от вершин одной ветви дерева. Если вершина получила два ответа от одной ветви i, т.е. от вершин с индексами (i,j_1) и (i,j_2), то выбирается ответ от вершины ближайшей к корню. Это вершина с минимальным номером на ветви, т.е. вершина с индексом $(i,min\{j_1,j_2\})$. Сообщение типа 3 посылается по дугам графа при первой возможности (по сигналу О или П). В сообщении ответ (значение функции G) содержится вместе с индексом отправителя, т.е. вершины, вычислившей этот ответ. Эту пару (ответ, индекс отправителя) будем называть *индексированным ответом*.

5. Описание алгоритма

5.1. Память автомата и сообщения

1. Определения:
   - идентификатор дуги: (идентификатор начала дуги, номер дуги);
   - описание дуги: (идентификатор дуги, статус дуги);
   - индекс вершины: (номер ветви дерева, номер вершины на ветви);
   - описание вершины: (тип вершины: «лист» или «внутренняя», идентификатор вершины, индекс вершины);
   - индексированный ответ: (ответ как значение функции *G*, индекс вершины-отправителя).
2. Управляющие состояния автомата: *начальное, 0, 1, 2, 3*.
3. Память автомата (дополнительно к управляющему состоянию).
   3.1. Память автомата в начальном состоянии.
   Несущественно.
   3.2. Память автомата в состоянии *0*.
   - идентификатор вершины,
   - множество описаний дуг.

   3.3. Память автомата в состоянии *1*.
   - идентификатор вершины,
   - тип вершины: корень или не корень,
   - *w* (только для корня),
   - множество описаний дуг.

   3.4. Память автомата в состоянии *2*.
   - идентификатор вершины,
   - тип вершины: корень, лист, внутренняя,
   - *w* (только для корня),
   - индекс вершины (не для корня),
   - множество описаний вершин.

   3.5. Память автомата в состоянии *3*.
   - тип вершины: корень, лист, внутренняя,
   - *w* (только для корня),
   - *H* (только для корня),
   - индекс вершины (не для корня),
   - номер вопроса,
   - *(G,E)*,
   - множество индексированных ответов.

4. Формат «внешних» сообщений (дополнительно к типу сообщения).
   4.1. Извне сообщение «Старт»: *w*.
   4.2. Вовне сообщение «Готов к работе»: без параметров.
   4.3. Извне сообщение «Вопрос»: описание функций *H,G,E*.
   4.4. Вовне сообщение «Ответ»: значение *H(G(q(V)))*.
   4.5. Вовне сообщение «Ошибка протокола»: без параметров.
5. Формат «внутренних» сообщений (дополнительно к типу сообщения).
   5.1. Формат сообщения типа *1*.
      - идентификатор дуги, по которой сообщение передаётся,
      - множество описаний дуг.
   5.2. Формат сообщения типа *2*.
      - множество описаний вершин.
   5.3. Формат сообщения типа *3*.
      - номер вопроса,
      - *(G,E)*,
      - множество индексированных ответов.

5.2. Работа автомата

1. Начальное состояние.

   Автомат вершины начинает работать в начальный момент времени с *начального состояния*. Автомат переходит в состояние *0*, определяет идентификатор вершины, в которой он находится, с помощью примитива «Дай идентификатор» и запоминает его в своей памяти. Далее автомат формирует в своей памяти пустое множество описаний дуг.
2. Состояние *0*.
   2.1. Сигнал П, параметры: номер дуги.

      Если описания дуги с этим номером ещё нет, то создаётся описание дуги (по правилу 1 из 4.3): идентификатор вершины, номер дуги, статус = *1*. В противном случае (по правилу 3 из 4.3) статус дуги меняется на *3*.
   2.2. Сигналы О и И невозможны в состоянии *0*.
   2.3. Сообщение «Старт».

      По предположению, это сообщение приходит извне в корень. В этот момент в корне могут быть описания только выходящих дуг со статусом *1* или *3*. Автомат переходит в состояние *1*, устанавливает тип вершины = корень, запоминает *w* и посылает по каждой (выходящей) дуге со статусом *1* сообщение типа *1* с множеством описаний дуг из корня.

Альтернативное поведение в вырожденном случае: если в корне до получения сообщения «Старт» не появилось ни одной выходящей дуги. Отсюда следует, по предположению о начальных дугах в разделе 3, что нет начальных дуг и, следовательно, нет других вершин. Корень переходит в состояние 2, устанавливает тип вершины = корень, устанавливает *w=0*, формирует пустое множество описаний вершин и посылает вовне ответное сообщение «Готов к работе».

2.4. Сообщение «Вопрос».

По предположению, это сообщение извне может придти только в корень. При нормальной работе оно должно приходить только после того, как корень ответит на сообщение «Старт». Поэтому в состоянии *0* корень посылает ответное сообщение «Ошибка протокола».

2.5. Сообщение типа *1*.

Вершина в состоянии *0* может получить это сообщение только, если это не корневая вершина. Это сообщение приходит по входящей дуге из другой вершины. В этот момент времени в вершине могут быть описания только выходящих дуг со статусом *1* или *3*, а в принятом сообщении не могут быть описания этих дуг, поскольку автомат ещё не посылал ни одного сообщения по выходящим дугам. Автомат переходит в состояние *1*, устанавливает тип вершины = не корень и (по правилу 5 из 4.3) добавляет все описания дуг из сообщения во множество описаний дуг в вершине. Если статус дуги, по которой пришло сообщение, равен *1*, то (по правилу 4 из 4.3) он меняется на *3*. После этого автомат посылает по каждой выходящей дуге (дуге, идентификатор начала которой равен идентификатору текущей вершины) сообщение типа *1* с множеством описаний дуг из вершины.

2.6. Сообщения типа *2* и *3* невозможны в состоянии *0*.

3. Состояние *1*.

3.1. Сигнал О, параметры: номер дуги.

Если описание дуги есть в вершине и статус дуги равен *1*, то он меняется на *2* по правилу 2 из 4.3. В любом случае по дуге посылается сообщение типа *1* с множеством описаний дуг из вершины.

3.2. Сигнал П, параметры: номер дуги.

Если описание дуги есть в вершине и статус дуги равен *1*, то он меняется на *3* по правилу 3 из 4.3. В любом случае по дуге посылается сообщение типа *1* с множеством описаний дуг из вершины.

3.3. Сигнал И, параметры: номер дуги.

Если описание дуги есть в вершине и статус дуги равен *1*, то он меняется на *3* по правилу 3 из 4.3.

3.4. Сообщение «Старт» или «Вопрос».

По предположению, такое сообщение извне может придти только в корень. При нормальной работе сообщение «Старт» не должно приходить повторно, а сообщение «Вопрос» должно приходить только после того, как корень ответит на сообщение «Старт». Поэтому в состоянии *1* корень посылает ответное сообщение «Ошибка протокола».

3.5. Сообщение типа *1*.

Автомат просматривает множество описаний дуг в сообщении. По правилу 5 из 4.3, если описания дуги не было в вершине, оно добавляется в вершину, а в противном случае статус дуги в вершине корректируется. Если статус дуги, по которой пришло сообщение, равен *1*, то он меняется на *3* (по правилу 4 из 4.3).

Автомат корня дополнительно проверяет статусы всех дуг, описания которых хранятся в его памяти. Если статусы всех дуг равны *3*, то автомат переходит в состояние 2, корректирует ширину дерева $w:=min\{w,n-1\}$ и строит виртуальное дерево, как описано в подразделе 4.2, т.е. создаёт множество описаний некорневых вершин. Если это множество пусто (для случая $n=1$ при наличии петель, появившихся до получения сообщения «Старт»), корень посылает вовне сообщение «Готов к работе».

3.6. Сообщение типа *2*.

В состоянии *1* автомат вершины может получить сообщение типа *2* только в том случае, если эта вершина – не корень. Автомат переходит в состояние *2*, переписывает в свою память всё множество описаний вершин из сообщения. В этом множестве должно быть и описание вершины *a*, которое определяется по идентификатору вершины *a*. Из этого описания запоминается индекс и тип вершины *a*, а само описание удаляется из множества.

3.7. Сообщение типа *3* невозможно в состоянии *1*.

4. Состояние *2*.

4.1. Сигнал О или П, параметры: номер дуги.

По дуге посылается сообщение типа *2* с множеством описаний вершин из вершины.

4.2. Сигнал И, параметры: номер дуги.

Ничего не делается.

4.3. Сообщение «Старт».

Аналогично 3.4. корень посылает ответное сообщение «Ошибка протокола».

4.4. Сообщение «Вопрос».

По предположению, такое сообщение извне может придти только в корень. При нормальной работе в состоянии *2* сообщение «Вопрос» должно приходить только после того, как корень ответит на предыдущее сообщение извне «Старт». Корень проверяет, пусто ли множество описаний вершин в его памяти. Если не пусто, то это означает, что ответ на сообщение «Старт» ещё не отправлен вовне, поэтому корень посылает ответное сообщение «Ошибка протокола». Если пусто, корень переходит в состояние *3*, запоминает из сообщения функции *H*, *G* и *E*, устанавливает номер вопроса = *1*, формирует пустое множество индексированных ответов. Если *w=0*, т.е. корень v_0 – единственная вершина, корень вычисляет значение $H(G(q(v_0)))$ и посылает вовне сообщение «Ответ» с этим значением.

4.5. Сообщение типа *1*.

В состоянии *2* это сообщение игнорируется: автомат ничего не делает.

4.6. Сообщение типа *2*.

Автомат строит пересечение множеств описаний вершин из сообщения и из своей памяти. Это пересечение сохраняется в памяти автомата. Если это пересечение пусто, а текущая вершина – корень, то корень посылает вовне сообщение «Готов к работе».

4.7. Сообщение типа *3*.

В состоянии 2 такое сообщение может получить только некорневая вершина *v* с индексом *(i,j)*. Автомат переходит в состояние 3, переписывает в свою память из сообщения вопрос *(G,E)*, номер вопроса (который должен быть равен *1*) и множество индексированных ответов. Если это листовая вершина, то автомат вычисляет значение *G(q(v))* и записывает его вместе со своим индексом *(i,j)* как индексом отправителя во множество индексированных ответов. Заметим, что это будет первый индексированный ответ от вершин ветви *i*. Если это внутренняя вершина, то автомат проверяет, нет ли во множестве индексированных ответов в сообщении ответа $G(b_1)$ от вершины с индексом *(i,j+1)*. Если есть, то автомат вычисляет значение $E(G(b_1), G(q(v)))$ и записывает его вместе с индексом *(i,j)* как индексом отправителя во множество индексированных ответов вместо ответа $G(b_1)$ от вершины с индексом *(i,j+1)*.

5. Состояние *3*.

5.1. Сигнал О или П, параметры: номер дуги.

По дуге посылается сообщение типа *3* с множеством индексированных ответов из вершины.

5.2. Сигнал И, параметры: номер дуги.

Ничего не делается.

5.3. Сообщение «Старт».

По предположению, такое сообщение извне может придти только в корень. При нормальной работе сообщение «Старт» не должно приходить повторно. Поэтому в состоянии *3* корень посылает ответное сообщение «Ошибка протокола».

5.4. Сообщение «Вопрос».

По предположению, такое сообщение извне может придти только в корень. При нормальной работе сообщение «Вопрос» должно приходить только после того, как корень ответит на предыдущее сообщение извне «Вопрос». Корень проверяет, что во множестве индексированных ответов индексы отправителей пробегают всё множество индексов ближайших к корню вершин: $\{(i,1)|i\in[1..w]\}$.

Если это не так, то это означает, что корень ещё не ответил вовне на предыдущий «Вопрос». Поэтому корень посылает ответное сообщение «Ошибка протокола».

В противном случае корень начинает работу над поступившим вопросом. Запоминает из сообщения функции *H*, *G* и *E*, увеличивает номер вопроса на *1*, формирует пустое множество индексированных ответов. Если $w=0$, т.е. корень v_0 – единственная вершина, корень вычисляет значение $H(G(q(v_0)))$ и посылает вовне сообщение «Ответ» с этим значением.

5.5. Сообщение типа *1*, типа *2*.

В состоянии *3* такое сообщение игнорируется: автомат ничего не делает.

5.6. Сообщение типа *3*.

Пусть текущая вершина v имеет индекс (i,j). Автомат сравнивает номер вопроса a в сообщении и номер вопроса b в своей памяти.

5.6.1. Если $a<b$, то это означает, что ответ на вопрос с номером a уже отправлен корнем вовне, а принятое сообщение – «остаточное» от вычисления ответа на более ранний вопрос сообщение игнорируется: автомат ничего не делает.

5.6.2. Если $a>b$, то это означает, что ответ на вопрос с номером b уже отправлен корнем вовне, а сообщение содержит новый вопрос. Заметим, что в этом случае $a=b+1$, т.е. сообщение содержит следующий вопрос, а текущая вершина – не корень. Автомат приступает к работе над новым вопросом: переписывает в свою

память из сообщения вопрос *(G,E)*, номер вопроса и множество индексированных ответов.

Если это листовая вершина, то автомат вычисляет значение *G(q(v))* и записывает его вместе со своим индексом *(i,j)* как индексом отправителя во множество индексированных ответов. Заметим, что это будет первый индексированный ответ (на вопрос с номером *a*) от вершин ветви *i*.

Если это внутренняя вершина, то автомат проверяет, нет ли во множестве индексированных ответов в сообщении ответа $G(b_1)$ от вершины с индексом *(i,j+1)*. Если есть, то автомат вычисляет значение $E(G(b_1), G(q(v)))$ и записывает его вместе с индексом *(i,j)* как индексом отправителя во множество индексированных ответов вместо ответа $G(b_1)$ от вершины с индексом *(i,j+1)*.

5.6.3. Если $a=b$, то это означает, что продолжается работа над текущим вопросом с номером $a=b$. Автомат сравнивает индексированные ответы из сообщения и из своей памяти. Если в сообщении есть индексированный ответ $(G_1,(i_1,j_1))$, а в вершине нет ответа от вершины на той же ветви i_1, то индексированный ответ $(G_1,(i_1,j_1))$ добавляется в множество индексированных ответов в вершине. Если в вершине есть индексированный ответ $(G_2,(i_1,j_2))$, то автомат сравнивает расположение на ветви отправителей ответов из сообщения и из вершины. Если $j_1 \geq j_2$, то ответ из сообщения игнорируется. Если $j_1 < j_2$, то ответ $(G_1,(i_1,j_1))$ из сообщения замещает собой ответ $(G_2,(i_1,j_2))$ в вершине.

Если текущая вершина листовая, то автомат больше ничего не делает.

Если текущая вершина внутренняя, то автомат проверяет, нет ли во множестве индексированных ответов в вершине ответа $G(b_1)$ от вершины с индексом *(i,j+1)*. Если есть, то автомат вычисляет значение $E(G(b_1), G(q(v)))$ и записывает его вместе с индексом *(i,j)* как индексом отправителя в множество индексированных ответов вместо ответа $G(b_1)$ от вершины с индексом *(i,j+1)*.

Если текущая вершина корень, то автомат проверяет, что во множестве индексированных ответов индексы отправителей пробегают всё множество индексов ближайших к корню вершин: $\{(i,1)|i \in [1..w]\}$. Если это не так, то автомат больше ничего не делает. В противном случае автомат вычисляет результирующее значение функции $H(E_{w+1}(G_1,...,G_w,G_{w+1}))$, где G_i ответ от вершины с индексом *(i,1)* для $i=1..w$, $G_{w+1}=G(q(v_0))$. При этом функция E_{w+1} вычисляется итеративно с помощью функции E: для $j>1$ имеем $E_{j+1}(G_1,...,G_{j+1}) = E(E_j(G_1,...,G_j), G_{j+1})$.

Вычисленное значение функции автомат отправляет вовне в сообщении «Ответ».

6. Утверждения и оценки

6.1. Размеры памяти автомата и сообщения

Будем считать, что идентификатор вершины занимает x бит памяти, номер дуги ограничен сверху числом $s \le m$, описание каждой из функций H,G или E занимает не более y бит памяти, значение функции занимает не более z бит памяти, номер вопроса не превышает N. Тогда следующие величины имеют ограничения по размеру: идентификатор дуги – $x+logs$, описание дуги – $x+logs+2$, номер ветви дерева – $logw$, номер вершины на ветви – $logh=log(](n-1)/w[)$, индекс – $logw+logh=logn$, описание вершины – $1+x+logn$, индексированный ответ – $z+logn$, управляющее состояние автомата – 3, множество описаний дуг – $m(x+logs+2)$, тип вершины – 2, множество описаний вершин – $n(1+x+logn)$, множество индексированных ответов – $w(z+logn)$, тип сообщения – 3.

Отсюда следуют следующие ограничения на размер памяти автомата в различных состояниях (учитывая, что $max\{logw,logn\}=logn$):

начальное состояние – $3 = \boldsymbol{O}(1)$,

состояние 0 – $3+x+m(x+logs+2) = \boldsymbol{O}(mx+mlogs)$,

состояние 1 – $3+x+3+logw+m(x+logs+2) = \boldsymbol{O}(logw+mx+mlogs)$,

состояние 2 – $3+x+3+logn+n(1+x+logn) = \boldsymbol{O}(nx+nlogn)$,

состояние 3 – $3+3+logn+3y+logN+w(z+logn) = \boldsymbol{O}(y+logN+wz+wlogn)$.

Также получаются следующие ограничения на размер сообщения (учитывая, что $max\{logw,logn\}=logn$):

Старт – $3+logw = \boldsymbol{O}(logw)$,

Готов к работе – $3 = \boldsymbol{O}(1)$,

Вопрос – $3+3y = \boldsymbol{O}(y)$,

Ответ – $3+z = \boldsymbol{O}(z)$,

Ошибка протокола – $3 = \boldsymbol{O}(1)$,

Тип 1 – $3+x+logs+m(x+logs+2) = \boldsymbol{O}(mx+mlogs)$,

Тип 2 – $3+n(1+x+logn) = \boldsymbol{O}(nx+nlogn)$,

Тип 3 – $3+logN+2y+w(z+logn) = \boldsymbol{O}(logN+y+wz+wlogn)$.

6.2. Время работы алгоритма

Лемма 2. После получения корнем извне сообщения «Старт» за время $\boldsymbol{O}(n)$ каждая вершина получит сообщение.

Доказательство. Для корня утверждение очевидно. Пусть $n>1$. По предположению о начальных дугах в разделе 3, начальная дуга существует в

начальный момент времени и не меняется до тех пор, пока по ней не пройдёт первое сообщение, и каждая вершина достижима из корня по начальным дугам. Поскольку временем срабатывания автомата мы пренебрегаем, автомат корня сформирует сообщение типа *1* через *0* тактов после получения сообщения «Старт». Поскольку длина любого пути в графе, в том числе пути по начальным дугам, не превосходит *n-1*, от корня до каждой некорневой вершины дойдёт сообщение типа *1* за время не более *n-1*.

Лемма доказана.

Регистрируемой дугой назовём такую дугу $a \to b$, которая появляется первый раз до получения вершиной *a* первого сообщения.

Лемма 3. В любой момент времени, когда вершина находится в состоянии *0* или *1*, множество описаний дуг в этой вершине содержит описания только регистрируемых дуг. Если в этом множестве есть описание дуги $a \to b$, то в нём есть описание всех регистрируемых дуг, выходящих из *a*.

Доказательство. Описание дуги $a \to b$ создаётся только до получения вершиной *a* первого сообщения, поэтому создаются описания только регистрируемых дуг, и только такие описания могут оказаться в сообщениях и в вершинах. Поскольку все регистрируемые дуги, выходящие из вершины *a*, появляются до получения вершиной *a* первого сообщения, описания всех этих дуг совместно оказываются в каком-либо сообщении или в какой-нибудь вершине.

Лемма доказана.

Лемма 4. Через время $\boldsymbol{O(n)}$ после получения вершиной *a* первого сообщения все регистрируемые дуги, выходящие из этой вершины, будут в корне иметь описание со статусом *3*.

Доказательство. Рассмотрим регистрируемую дугу $a \to b$. При получении вершиной *a* первого сообщения по дуге $a \to b$ посылается сообщение, содержащее описание этой дуги со статусом *1*. Поскольку время передачи сообщения по дуге не превосходит 1 такт, не позже чем через 1 такт это сообщение либо будет доставлено в вершину *b*, либо пропадёт. В первом случае в вершине *b* дуга $a \to b$ получит статус *3*, после чего по лемме 1 через время не более *3(n-1)* в корне эта дуга тоже будет иметь статус *3*. Во втором случае в вершине *a* дуга $a \to b$ получит статус *3*, после чего по лемме 1 через время не более *3(n-1)* в корне эта дуга тоже будет иметь статус *3*. Таким образом, после получения вершиной *a* первого сообщения через время не более чем $1+3(n-1)=\boldsymbol{O(n)}$ в корне все регистрируемые дуги, выходящие из этой вершины, будут иметь статус *3*.

Лемма доказана.

Лемма 5. Если начальная дуга $a \to b$ имеет в корне статус *3*, то корень «знает» о конце дуги, вершине *b*, т.е. в корне есть множество описаний всех регистрируемых дуг, выходящих из *b*, и это множество не пусто.

Доказательство. Поскольку начальная дуга $a \to b$ не меняется до тех пор, пока по ней не пройдёт первое сообщение, она получает статус *3* в вершине *b*, когда это сообщение дойдёт до вершины *b*. Одновременно по лемме 3 вершина *b* добавит во множество описаний все регистрируемые дуги, выходящие из неё. В этот же момент времени в вершине *a* дуга получает статус *2*, следовательно, в вершине *a* эта дуга не получит статус *3* по сигналам И или П. Следовательно, любое множество описаний дуг (в вершине или в сообщении), содержащее описание этой дуги со статусом *3*, содержит и описание всех регистрируемых дуг, выходящих из вершины *b*. Покажем, что из вершины *b* выходит хотя бы одна регистрируемая дуга. Действительно, по предположению о долгоживущих дугах (в каждый момент времени они образуют сильно связный суграф), хотя бы одна долгоживущая дуга должна существовать в момент получения вершиной *b* первого сообщения. А тогда эта долгоживущая дуга регистрируемая.

Лемма доказана.

Теорема 1. После получения корнем извне сообщения «Старт» за время $O(n)$ корень перейдёт в состояние *2*, причём во множестве описаний вершин в корне будут все вершины графа.

Доказательство. В вырожденном случае, когда из корня не выходит ни одна регистрируемая дуга, корень переходит в состояние *2* в тот же момент времени, когда он получает сообщение «Старт». В этом случае по предположению о начальных дугах $n=0$.

Пусть $n>1$. По лемме 2 за время $O(n)$ все некорневые вершины получат первое сообщение. По описанию алгоритма это может быть только сообщение типа *1*. По лемме 4 после этого через время $O(n)$ в корне будет описание всех регистрируемых дуг со статусом *3*. А в этом случае автомат корня переходит в состояние 2. Теперь покажем, что во множестве описаний вершин в корне будут все вершины графа. Для этого достаточно, чтобы при переходе из состояния 1 в состояние 2 в корне каждая вершина встречалась как начало описания некоторой дуги. По предположению о начальных дугах в каждую вершину *b* существует путь от корня по начальным дугам. Поскольку все начальные дуги регистрируемые, в корне будет описание некоторой начальной дуги $a \to b$. А тогда по лемме 5 корень «знает» о вершине *b*. Итак, за время $O(n)$ корень перейдёт в состояние *2*, причём во множестве описаний вершин в корне будут все вершины графа.

Теорема доказана.

Теорема 2. Через время $O(n)$ после перехода корня в состояние *2* корень отправит вовне сообщение «Готов к работе».

Доказательство. Когда корень переходит в состояние 2, по теореме 1 корень «знает» о всех вершинах. Он формирует описание виртуального обратного дерева как множество описаний вершин и инициирует рассылку сообщения типа *2*. По лемме 1 все вершины получат это сообщение первый раз через

время $O(n)$. Каждая вершина, получая сообщение типа *2* первый раз, удаляет из него своё описание. Кроме того, при получении вершиной каждого следующего сообщения типа 2 строится пересечение множеств описаний вершин в сообщении и в вершине. Поэтому по лемме 1 через время $O(n)$ в корне получится пустое множество описаний вершин. Тогда корень отправит вовне сообщение «Готов к работе». Общее время $O(n)$.

Теорема доказана.

Теорема 3. Через время $O(n^2/w)$ после получения корнем сообщения «Вопрос» (без нарушения протокола) корень посылает вовне сообщение «Ответ», в котором находится правильно вычисленное значение указанной в вопросе функции.

Доказательство. Правильность вычисления функции непосредственно следует из её представления через минимальное агрегатное расширение (раздел 2), вложенное разбиение мультимножества $q(V)$ на виртуальном обратном дереве (подраздел 4.2) и реализацию этого дерева в алгоритме.

По лемме 1 за время $O(n)$ каждая вершина, в том числе каждая листовая вершина, получит сообщение типа 3 с текущим вопросом. Листовая вершина сразу вычисляет требуемый ответ. Далее каждая вершина с индексом (i,j) вычисляет требуемый ответ после получения по виртуальной дуге сообщения, в котором содержится ответ от предыдущей на ветви вершины с индексом $(i,j+1)$. По лемме 1 время передачи сообщения по виртуальной дуге равно $O(n)$. Следовательно, корень получит ответы от всех вершин с индексами $(1,1)..(w,1)$ за время $O(hn)$. Корень вычисляет свой ответ и посылает его вовне в сообщении «Ответ». Таким образом, общее время вычисления равно $O(hn)=O(n^2/w)$, поскольку $h=](n-1)/w[$.

Теорема доказана.

Список литературы

[1]. И. Бурдонов, А. Косачев, В. Кулямин. Параллельные вычисления на графе. Программирование, 2015, №1, с. 3-20.

[2]. И. Бурдонов, А. Косачев. Мониторинг динамически меняющегося графа. Труды Института системного программирования РАН Том 27. Выпуск 1. 2015 г. Стр. 69-96. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). DOI: 10.15514/ISPRAS-2015-27(1)-5

[3]. Steven S. Skiena. The Algorithm Design Manual. Springer-Verlag, New York, 1997.

[4]. И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов. Детерминированный случай. Программирование, 2003 г., №5, с. 59-69.

[5]. И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированный случай. Программирование, 2004 г., №1, с. 2-17.

[6]. M.O. Rabin. Maze Threading Automata. An unpublished lecture presented at MIT and UC. Berkeley, 1967.

[7]. И.Б. Бурдонов. Обход неизвестного ориентированного графа конечным роботом. Программирование, 2004 г., № 4, с. 11-34.
[8]. И.Б. Бурдонов. Проблема отката по дереву при обходе неизвестного ориентированного графа конечным роботом. Программирование, 2004 г., № 6, с. 6-29.
[9]. Бурдонов И.Б., Косачев А.С. Обход неизвестного графа коллективом автоматов. Труды ИСП РАН. Том 26-2, 2014 г., стр. 43-86.
[10]. Бурдонов И.Б., Косачев А.С., Кулямин В.В. Исследование графа набором автоматов. Программирование, 2015, №6, (в печати).
[11]. Кушнеренко А.Г., Лебедев Г.В. "Программирование для математиков", Наука, Главная редакция физико-математической литературы, Москва, 1988.

Parallel Calculations on Dynamic Graph

Igor Burdonov <igor@ispras.ru>
Alexander Kossatchev <kos@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. The problem of parallel computation of the value of a function of multiset of values recorded at the vertices of a directed strongly connected graph is considered. Computation is performed by automata that are located at the graph vertices. The automaton has local information on the graph: it "knows" only about arcs outgoing from the vertex it resides in, but it "does not know" where (to which vertices) those arcs go. The automata exchange messages with each other that are transmitted along the graph arcs that play role of message transfer channels. Computation is initiated by a message coming from outside to the automaton located at the initial vertex of the graph. At the end of work, this automaton sends outside the calculated function value. Two algorithms are proposed to solve this problem. The first algorithm carries out an analysis of the graph. Its purpose is to mark the graph by a change of the states of the automata at the vertices. Such marking is used by the second algorithm, which calculates the function value. This calculation is based on a pulsation algorithm: first, request messages are distributed from the automaton of the initial vertex over the graph, which should reach each vertex, and then response messages are sent from each vertex back to the initial vertex. In fact, the pulsation algorithm calculates aggregate functions for which the value of a function of a union of multisets is calculated by the values of the function of these multisets. However, it is shown that any function F(x) has an aggregate extension; that is, an aggregate function can be calculated as H(G(x)), where G is an aggregate function. Note that the marking of a graph does not depend on a function that will be calculated. This means that the marking of a graph is carried out once; after that, it can be reused for calculating various functions. Since the automata located in different vertices of the graph work in parallel, both graph marking and function calculation are performed in parallel. It is the first feature of this work. The second feature is that calculations are performed on a dynamically changing graph: its arcs can disappear, reappear or change their

target vertices. The constraints put on the graph changes are as minimal as it allows solving this problem in limited time. Estimate of working time is given for both algorithms.

Keywords: directed graphs; graph exploration, communicating automata, parallel processing, aggregate functions, dynamic graphs.

DOI: 10.15514/ISPRAS-2015-27(2)-12

For citation: Burdonov Igor, Kossatchev Alexander. Parallel Calculations on Dynamic Graph. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 189-220 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-12.

References

[1]. I. B. Burdonov, A. S. Kossatchev, V. V. Kuliamin. Parallel computations on graphs. Programming and computer Software, 41(1): 1-13, 2015.

[2]. I. B. Burdonov, A. S. Kossatchev. Monitoring of dynamically changed graph. Proceedings of the Institute for System Programming Volume 27 (Issue 1). 2015. pp. 69-96. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). DOI: 10.15514/ISPRAS-2015-27(1)-5 (in Russian).

[3]. Steven S. Skiena. The Algorithm Design Manual. Springer-Verlag, New York, 1997.

[4]. I. B. Burdonov, A. S. Kossatchev, V. V. Kuliamin. Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case. Programming and Computer Software, 29(5):245-258, 2003.

[5]. I. B. Burdonov, A. S. Kossatchev, V. V. Kuliamin. Irredundant Algorithms for Traversing Directed Graphs: The Nondeterministic Case. Programming and Computer Software, 30(1):2-17, 2004.

[6]. M.O. Rabin. Maze Threading Automata. An unpublished lecture presented at MIT and UC. Berkeley, 1967.

[7]. I. B. Burdonov. Traversal of an unknown directed graph by a finite automaton. Programming and Computer Software, 30(4): 11-34, 2004.

[8]. I. B. Burdonov. Backtracking on a tree in traversal of an unknown directed graph by a finite automaton. Programming and Computer Software, 30(6): 6-29, 2004.

[9]. I. B. Burdonov, A. S. Kossatchev. Obkhod neizvestnogo grafa kollektivom avtomatov [Unknown graph traversing by learning by automata group] Trudy ISP RAN [The proceeding of ISP RAS], Vol. 26-2, 2014, pp. 43-86. (in Russian)

[10]. I. B. Burdonov, A. S. Kossatchev, V. V. Kuliamin. Graph Learning by automata group. Programming and computer Software, 41(6), 2015 On printing.

[11]. Kushnerenko, A.G., and Lebedev, G.V., Programmirovanie dlya matematikov (Programming for Mathematicians), Moscow: Nauka, 1988

Моделирование и анализ поведения последовательных реагирующих программ

В.А.Захаров <zakh@cs.msu.su>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25;
НИУ Высшая школа экономики, Россия, Москва, 101000, ул. Мясницкая, д. 20.

Аннотация. Автоматы-преобразователи с конечным числом состояний над полугруппами могут служить простой моделью последовательных реагирующих программ. Эти программы работают во взаимодействии с окружающей средой, получая на входе поток управляющих сигналов и выполняя последовательности действий. Как только программа достигает определенного состояния управления, она выдает на выходе текущий результат вычисления. Элементарные действия реагирующей программы можно рассматривать как порождающие элементы некоторой полугруппы, а результат последовательного выполнения этих действий расценивается как элемент полугруппы, представляющий собой композицию этих действий. В данной статье предложен общий подход к решению двух задач анализа вычислений преобразователей такого вида – задачи проверки k-значности конечных преобразователей и задачи проверки эквивалентности k-значных преобразователей. Показано, что обе указанные задач можно свести к задаче поиска опровергающих вершин в ограниченных фрагментах размеченных системах переходов. При помощи предложенного подхода показано, что задача проверки эквивалентности детерминированных конечных преобразователей над полугруппами, которые могут быть вложены в конечно порожденные разрешимые полугруппы, и задача проверки k-значности таких преобразователей разрешимы за полиномиальное время. Кроме того, установлено, что задача проверки эквивалентности k-значных преобразователей разрешима за время, экспоненциальное относительно их размеров.

Ключевые слова: реагирующая программа; автомат-преобразователь; полугруппа; размеченная система переходов; эквивалентность; свойство k-значности; разрешающий алгоритм; сложность.

DOI: 10.15514/ISPRAS-2015-27(2)-13

Для цитирования: Захаров В.А. Моделирование и анализ поведения последовательных реагирующих программ. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 221-250. DOI: 10.15514/ISPRAS-2015-27(2)-13.

1. Введение

Автоматы-преобразователи (transducers) с конечным числом состояний естественным образом расширяют широко известную модель вычислений конечных автоматов Рабина-Скотта. В отличие от конечных автоматов, способных распознавать регулярные языки $L, L \subseteq \Sigma^*$, конечные преобразователи позволяют распознавать регулярные (рациональные) отношения $R, R \subseteq \Sigma^* \times \Delta^*$, на множествах конечных слов. Поэтому их область применения гораздо обширнее. Конечные преобразователи используются в системном программировании для построения простейших компиляторов [2], драйверов, осуществляющих фильтрацию и преобразования строк, изображений, потоков данных [3,21], в системах автоматизированного проектирования управляющих систем для разработки контроллеров [16], в компьютерной лингвистике для создания программ распознавания речи [14] и др. Для практического использования этой модели вычислений в столь разнообразных областях, нужны эффективные алгоритмы построения композиций, проверки эквивалентности и минимизации преобразователей.

Преобразователи могут служить простой моделью некоторого класса специальных программ. Эти программы работают во взаимодействии с окружающей средой, получая на входе поток управляющих сигналов, запросов или показаний датчиков. После приема очередной порции входных данных такая программа выполняет некоторую конечную последовательность действий и переходит в новое состояние управления. Как только программа достигает определенного состояния управления, она выдает на выходе текущий результат вычисления. Программы такого рода называются реагирующими программами. Их функция состоит в том, чтобы вырабатывать правильные отклики в ответ на внешние воздействия. К числу программ такого рода относятся операционные системы, сетевые протоколы, драйверы, контроллеры. При выборе подходящей математической модели важным обстоятельством являются два факта. Во-первых, разные последовательности действий, выполняемые такой программой, могут приводить к одному и тому же результату. Поэтому элементарные (базовые) действия реагирующей программы можно рассматривать как порождающие элементы некоторой полугруппы, а результат последовательного выполнения этих действий расценивается как элемент полугруппы, представляющий собой композицию этих действий. Во-вторых, одна и та же последовательность входных данных (запросов, управляющих сигналов) может приводить к разным результатам, поскольку на поведение программы помимо входных данных могут также влиять внешние факторы, скрытые от внешнего наблюдателя. Для моделирования такой реагирующей программы можно воспользоваться конечным автоматом-преобразователем (в общем случае недетерминированным), который преобразует слова входного алфавита Σ (алфавита управляющих сигналов) в полугрупповые выражения

Представим себе в качестве примера, что радиоуправляемый робот движется по поверхности планеты. Он может делать шаги в любом из четырех направлений N, E, S, W. Если этот робот получает некоторый управляющий сигнал syg, пребывая при этом в состоянии q, то он может выбрать и совершить некоторую серию шагов (скажем, N, N, W, S), а также перейти в следующее состояние q'. В особых состояниях управления q_{fin} робот оповещает о своем текущем местоположении. Существенными являются две особенности этого робота: 1) робот может достичь одного и того же места, совершая разные последовательности шагов (например N, N, W, W, S, E и W, N), и 2) выбор допустимой последовательности шагов может зависеть от случайных, не поддающихся контролю обстоятельств (например, от освещенности местности, погодных условий, состояния грунта и пр.). Наиболее простой моделью вычислений, пригодной для проектирования и анализа программы управления движением такого робота может служить недетерминированный конечный преобразователь, работающий над абелевой группой ранга 2.

Так мы приходим к концепции автомата-преобразователя, который принимает на входе слова некоторого алфавита Σ и выдает на выходе элементы некоторой конечно порожденной полугруппы S. Для преобразователей такого вида нужно уметь решать тот же самый набор алгоритмических задач, что и для конечных автоматов Рабина-Скотта и традиционных автоматов-преобразователей, а именно

- задачу синтеза преобразователя по заданному алгебраическому или логическому описанию рационального отношения $R, R \subseteq \Sigma^* \times S$;
- задачу верификации заданного преобразователя относительно заданных алгебраических или логических спецификаций его поведения;
- задачу проверки эквивалентности двух заданных преобразователей;
- задачи детерминизации и минимизации преобразователей.

В данной статье приводятся некоторые результаты исследования проблемы эквивалентности для конечных преобразователей над полугруппами. Изучение этой проблемы и ряда смежных задач для классических автоматов-преобразователей началось в 60-х годах XX века. Вначале было установлено, что проблема эквивалентности для недетерминированных автоматов-преобразователей (generalized finite state machines) неразрешима [8,9] и при этом даже для однобуквенного входного алфавита [11]. Однако неразрешимость возникает только для таких недетерминированных преобразователей, у которых входные слова могут иметь неограниченно много выходных образов. На следующем этапе проблема эквивалентности изучалась для класса ограниченно недетерминированных преобразователей, у которых для любого входного слова количество его различных выходных образов не превосходит некоторого фиксированного числа k. Было

установлено, что свойство ограниченной недетерминированности преобразователей можно проверять за полиномиальное время [22]. В статье [10] было показано, что столь же эффективно можно проверять и свойство -значности недетерминированных преобразователей для любого заданного числа k. Разрешимость проблемы эквивалентности была установлена для детерминированных автоматов-преобразователей [5], функциональных (однозначных) преобразователей [4,19] и k-значных недетерминированных преобразователей [7,23]. В серии работ [6,17,18,20] авторы предложили метод декомпозиции произвольного -значного недетерминированного преобразователя в сумму функциональных и недвусмысленных (unambiguous) преобразователей. Этот метод был использован для построения алгоритмов проверки ограниченной недетерминированности, -значности и эквивалентности классических преобразователей, работающих над словами.

В данной статье предложен новый универсальный метод решения задач, связанных с проверкой эквивалентности конечных автоматов-преобразователей, работающих над полугруппами. Для проверки эквивалентности преобразователей π_1 и π_2 предлагается исследовать свойства размеченной системы переходов (Labeled Transition System, LTS) Γ_{π_1,π_2}, ассоциированной с этими преобразователями. Маршруты в этой LTS представляют все возможные пары вычислений преобразователей π_1 и π_2 на одном и том же входном слове. Каждая вершина u LTS Γ_{π_1,π_2}содержит информацию о том, в каких состояниях оказываются преобразователи π_1 и π_2 после прочтения некоторого входного слова, и том, каково расхождение выходных полугрупповых элементов, вычисленных преобразователями к этому моменту. Если оба преобразователя достигают своих заключительных состояний и дефицит вычисленных ими выходных результатов оказывается ненулевым, то это служит признаком того, что преобразователи π_1 и π_2 на некотором входном слове вычисляют разные результаты, т.е. не являются эквивалентными. Вершины LTS Γ_{π_1,π_2}, в которых проявляется указанный эффект, называются *опровергающими* вершинами. Таким образом, проверка эквивалентности преобразователей π_1 и π_2 сводится к проверке достижимости опровергающих вершин в LTS Γ_{π_1,π_2}. Оказывается, что для проверки достижимости опровергающих вершин достаточно исследовать лишь ограниченный фрагмент LTS Γ_{π_1,π_2}. Если анализируемые преобразователи π_1 и π_2 являются детерминированными, то размер этого фрагмента ограничен полиномом, зависящим от размера π_1 и π_2, и поэтому проблема эквивалентности для преобразователей такого рода может быть решена за полиномиальное время. Если же анализируемые преобразователи являются ограниченно недетерминированными, то размер этого фрагмента ограничен экспонентой, зависящей от размера π_1 и π_2 . Поскольку проблема эквивалентности недетерминированных конечных автоматов Рабина-Скотта является PSPACE-полной, эта оценка вряд ли может быть существенно улучшена. Тот же самый подход можно использовать и для проверки свойства

-значности недетерминированности конечных преобразователей, работающих над полугруппами.

Используемый в данной статье подход был впервые предложен в работе [24] для разработки полиномиальных по времени алгоритмов проверки эквивалентности программ. Сходные идеи были развиты также в статьях [6,17,18] и применены для анализа вычислений традиционных автоматов-преобразователей. Основные преимущества нашего подхода (помимо того очевидного факта, что он применим к автоматам-преобразователям более общего вида) таковы. Во-первых, в отличие от метода проверки свойств вычислений автоматов-преобразователей, впервые предложенного в статье [23] и затем развитого в статье [17], наш подход не требует предварительной декомпозиции анализируемых преобразователей и может быть применен к любым преобразователям без предварительной их подготовки. Во-вторых, предложенные нами процедуры верификации преобразователей, в отличие от методов, описанных в работах [18,20], не опираются ни на какие особенности устройства отношения переходов анализируемых преобразователей. Фактически, все рассматриваемые далее задачи, связанные с проблемой эквивалентности, решаются одной и той же процедурой обхода размеченной системы переходов в глубину с возвратом с целью обнаружения опровергающих вершин. Специфика решаемой задачи отражается лишь в устройстве самой системы переходов. Вследствие этого сложность разработанных нами алгоритмов проверки эквивалентности и ограниченной недетерминированности автоматов преобразователей существенно меньше сложности алгоритмов, описанных в статьях [4-7,17,18,20]. Разработанный нами подход применим к автоматам-преобразователям, проводящих вычисления в произвольной конечно порожденной полугруппе, которая может быть вложена в группу с разрешимой проблемой равенства слов.

Статья устроена следующим образом. В разделе 2 введены основные известные понятия, относящиеся к теории конечных последовательных автоматов-преобразователей. В разделе 3 описан алгоритм проверки эквивалентности детерминированных автоматов-преобразователей, работающих над полугруппами. В разделе 4 описаны сходные по устройству алгоритмы решения двух задач – проверки свойства функциональности преобразователей и проверки эквивалентности функциональных преобразователей. В разделе 5 предложен метод проверки свойства -значности недетерминированных преобразователей. Для большей наглядности в этом разделе подробно описан и обоснован алгоритм проверки свойства 2-значности преобразователей. Из описания этого алгоритма легко выводится и общий способ построения алгоритмов проверки свойства -значности недетерминированности преобразователей для любого значения параметра k. В разделе 6 описан алгоритм проверки эквивалентности 2-значных недетерминированных автоматов преобразователей. В разделе 7 приведены оценки сложности предложенных разрешающих алгоритмов для конечных

преобразователей над свободной полугруппой и свободной коммутативной полугруппой. В заключительном разделе проведен сравнительный анализ полученных результатов, а также обозначены перспективы решения других задач синтеза и анализа автоматов-преобразователей над полугруппами.

2. Основные понятия

Пусть задан конечный алфавит Σ. Записью Σ^* обозначим множество всех конечных слов над алфавитом Σ. Буквы алфавита Σ могут быть истолкованы как элементарные события, происходящие во внешней среде и оказывающие влияния на информационную систему, взаимодействующую с этой средой. В этом случае слова можно истолковывать как возможный сценарий поведения внешней среды по отношению к информационной системе.

Конечным автоматом над алфавитом Σ назовем систему $M = \langle \Sigma, Q, init, F, \varphi \rangle$, в которой

- Q – конечное множество *состояний*;
- $init$ – *начальное состояние*, $init \in Q$;
- F – подмножество *финальных состояний*, $F \subseteq Q$;
- $\varphi \subseteq Q \times \Sigma \times Q$ – *отношение переходов*.

Автомат M допускает слово $w = a_1 a_2 \dots a_n$, если существует такая последовательность состояний $q_0, q_1, \dots, q_n$, в которой $q_0 = init$, $q_n \in F$, и $(q_{i-1}, a_i, q_i) \in \varphi$ выполняется для каждого $i, 1 \leq i \leq n$. Язык $L(M)$ автомата M – это множество всех слов, допускаемых автоматом M. Запись $M[q]$ будет использоваться для обозначения автомата $\langle \Sigma, Q, q, F, \varphi \rangle$, в котором начальным состоянием служит состояние q.

Рассмотрим полугруппу $S = (B, \cdot, e)$, порожденную множеством элементов B и имеющую единичный (нейтральный) элемент e. Порождающие элементы полугруппы можно рассматривать как элементарные операторы (действия), которые выполняет информационная система в ответ на внешние воздействия. Бинарная операция полугруппы в этом случае соответствует последовательной композиции этих операторов. Если два полугрупповых выражения s_1 и s_2 имеют одинаковое значение, то это означает, что соответствующие этим выражениям последовательности операторов вычисляют одинаковый результат.

Конечным преобразователем (далее просто преобразователем) над полугруппой S называется система $\pi = \langle \Sigma, S, Q, q_0, F, T \rangle$, в которой

- Q – конечное множество состояний;
- q_0 – начальное состояние, $q_0 \in Q$;
- F – подмножество финальных состояний, $F \subseteq Q$;

- $T \subseteq Q \times \Sigma \times S \times Q$ – конечное отношение переходов.

Четверки (q, a, s, q') из множества T называются *переходами*; для их обозначения используем запись $q \xrightarrow{a/s} q'$. Для каждого состояния $q, q \in Q$, запись $\pi[q]$ будет обозначать преобразователь $\langle \Sigma, S, Q, q, F, T \rangle$, в котором состояние q играет роль начального состояния. Мы будем использовать запись M_π для обозначения *автомата-подложки* $\langle \Sigma, Q, q_0, F, \varphi_\pi \rangle$, в котором $\varphi_\pi = \{(q, a, q') : q \xrightarrow{a/s} q' \text{ для некоторого } s \text{ из } S\}$. Для конечного автомата $M = \langle \Sigma, Q', init, F', \varphi' \rangle$ и преобразователя $\pi = \langle \Sigma, S, Q, q_0, F, T \rangle$ *проекцией преобразователя* π *на автомат* M называется преобразователь $\pi|_M = \langle \Sigma, S, Q \times Q', (q_0, init), F \times F', \hat{T} \rangle$, в котором отношение переходов $\hat{T}$ определяется требованием: $(q, p) \xrightarrow{a/s} (q', p') \Leftrightarrow q \xrightarrow{a/s} q' \wedge (p, a, p') \in \varphi$.

Вычислением преобразователя π на входном слове $w = a_1 a_2 \dots a_n$ называется всякая последовательность переходов

$$run = q_i \xrightarrow{a_1/s_1} q_{i+1} \xrightarrow{a_2/s_2} \cdots \xrightarrow{a_{n-1}/s_{n-1}} q_{i+n-1} \xrightarrow{a_n/s_n} q_{i+n} .$$

Конечный преобразователь служит моделью последовательной реагирующей программы. Каждый переход $q \xrightarrow{a/s} q'$ означает, что при получении внешнего воздействия a в состоянии управления q эта программа совершает переход в состояние управления q' и выполняет последовательность операторов s. Последовательность таких переходов образует вычисление реагирующей программы.

Элемент $s = s_1 \cdot s_2 \cdots s_n$ полугруппы S называется *образом* входного слова w, а пара (w, s) называется *меткой* вычисления run. Запись $q_i \xrightarrow{w/s} q_{i+n}$,будет использоваться для сокращенного обозначения вычисления преобразователя на входном слове w. Если $q_i = q_0$, то вычисление run называется *начальным вычислением*. Если $q_{i+n} \in F$, то вычисление run называется *финальным вычислением*. *Полным вычислением* называется всякое вычисление преобразователя, которое является как начальным, так и финальным. Записью $Lab(\pi)$ обозначим множество меток (w, s) всех полных вычислений преобразователя π; это множество является отношением, которое реализует преобразователь π. Преобразователи π' и π'' считаются *эквивалентными* (и обозначаются записью $\pi' \sim \pi''$), если $Lab(\pi') = Lab(\pi'')$.

Образ входного слова представляет собой результат вычисления реагирующей программы в ответ на последовательность воздействий среды. Внешний наблюдатель регистрирует результаты лишь тех вычислений, которые достигают финальных состояний управления. Поэтому наблюдаемое поведение реагирующей программы π полностью определяется множеством меток $Lab(\pi)$. Таким образом, эквивалентность преобразователей означает, что соответствующие реагирующие программы имеют одинаковое наблюдаемое поведение.

Состояние q преобразователя π считается *полезным*, если хотя бы одно полное вычисление проходит через это состояние. Далее будем полагать, что все состояния преобразователя полезны, поскольку от бесполезных состояний можно избавиться, сохранив при этом реализуемое преобразователем отношение. Преобразователи такого рода будем называть *правильными*. Преобразователь считается *детерминированным*, если для любой буквы a и для любого состояния q множество T содержит не более одного перехода вида $q \xrightarrow{a/s} q'$. Преобразователь π называется -*значным*, где k – положительное целое число, если для любого входного слова w отношение $Lab(\pi)$ содержит не более k меток вида (w, s). Однозначный преобразователь также называется *функциональным*.

Далее в статье будет описан общий метод построения разрешающих процедур для задач проверки эквивалентности и -значности преобразователей на полугруппами S , вложимыми в конечно порожденные разрешимые группы. Полугруппа S называется *вложимой* в группу G, если эта группа содержит подполугруппу S' , изоморфную полугруппе S. Необходимые и достаточные условия вложимости полугруппы в группу были установлены А.П. Мальцевым в статье [12]. В последующей статье [13] было доказано, что никакой критерий вложимости полугруппы в группу не может представлен конечным множеством условий. Однако для полугрупп специального вида условия их вложимости в группу могут быть сформулированы сравнительно просто. Например, свободная полугруппа всегда вложима в свободную группу. Для вложимости в группу коммутативной полугруппы необходимо и достаточно, чтобы она обладала свойствами левого и правого сокращения. Группа называется разрешимой, если в ней разрешима *проблема равенства слов*, т.е. существует алгоритм, который для любой пары выражений, составленных из порождающих элементов группы, может определить, представляют ли эти выражения один и тот же элемент группы. В этой статье мы будем полагать, что G является конечно порожденной разрешимой группой, содержащей полугруппу S в качестве подполугруппы. Поэтому для всякого элемента s полугруппы S запись s^- будет обозначать элемент группы G, обратный групповому элементу s.

3. Проверка эквивалентности детерминированных преобразователей

Пусть заданы пара детерминированных преобразователей $\pi' = \langle \Sigma, S, Q', q_0', F', T' \rangle$ и $\pi'' = \langle \Sigma, S, Q'', q_0'', F'', T'' \rangle$ над полугруппой S, которая вложена в конечно порожденную разрешимую группу G. Для проверки эквивалентности этих преобразователей рассмотрим размеченную систему переходов (LTS) $\Gamma^0_{\pi',\pi''} = \langle Q' \times Q'' \times G, \Rightarrow \rangle$. Ее вершинами являются тройки вида (q', q'', g), где $q' \in Q', q'' \in Q''$, и $g \in G$. Отношение переходов $\Rightarrow$ определяется следующим образом: для каждой пары вершин $v_1 = (q_1', q_1'', g_1)$

и $v_2 = (q_2', q_2'', g_2)$, и произвольной буквы a отношение $v_1 \overset{a}{\Rightarrow} v_2$ имеет место тогда и только тогда, когда в преобразователях π' и π'' есть переходы $q_1' \xrightarrow{a/s'} q_2'$ и $q_1'' \xrightarrow{a/s''} q_2''$ соответственно, и при этом $g_2 = (s')^- g_1 s''$.

Пусть задано слово $w = a_1 a_2 \dots a_n$ и пара вершин $v = (q_1', q_1'', g_1)$ и $u = (q_2', q_2'', g_2)$. Запись $v \overset{w}{\Rightarrow} u$ будет обозначать последовательность переходов $v \overset{a_1}{\Rightarrow} v_1 \overset{a_2}{\Rightarrow} v_2 \overset{a_3}{\Rightarrow} \cdots \overset{a_{n-1}}{\Longrightarrow} v_{n-1} \overset{a_n}{\Rightarrow} u$, которую будем называть маршрутом в LTS $\Gamma^0_{\pi',\pi''}$. В этом случае будем говорить, что вершина u достижима из вершины v. Нетрудно видеть, что в LTS $\Gamma^0_{\pi',\pi''}$ существует маршрут $v \overset{w}{\Rightarrow} u$ в том и только том случае, если ${q'}_1 \xrightarrow{w/s'} {q'}_2$, ${q''}_1 \xrightarrow{w/s''} {q''}_2$, и $(s')^- g_1 s'' = g_2$.

Вершину $v_{src} = (q_0', q_0'', e)$, где e – единичный элемент группы G, назовем *стартовой вершиной* LTS $\Gamma^0_{\pi',\pi''}$. Используем запись $\mathrm{V}^0_{\pi',\pi''}$ для обозначения множества вершин LTS $\Gamma^0_{\pi',\pi''}$, достижимых из стартовой вершины. Всякую вершину (q_1, q_2, g) будем называть *опровергающей*, если она удовлетворяет одному из следующих требований:

- q_1 и q_2 – финальные состояния преобразователей π' и π'', и при этом $g \neq e$;
- одно из состояний q_σ, $\sigma \in \{1,2\}$, является финальном, тогда как другое состояние $q_{3-\sigma}$ не является финальным;
- для некоторой буквы a одно из состояний q_σ, $\sigma \in \{1,2\}$, имеет a-переход $q_\sigma \xrightarrow{a/s} q'_\sigma$, тогда как другое состояние $q_{3-\sigma}$ не имеет a-переходов.

Множество всех опровергающих вершин LTS $\Gamma^0_{\pi',\pi''}$ обозначим записью $R^0_{\pi',\pi''}$. Значение множеств вершин $\mathrm{V}^0_{\pi',\pi''}$ и $R^0_{\pi',\pi''}$ для проверки эквивалентности детерминированных преобразователей проясняют следующие две леммы.

Лемма 1. Детерминированные преобразователи π' и π'' эквивалентны тогда и только тогда, когда $\mathrm{V}^0_{\pi',\pi''} \cap R^0_{\pi',\pi''} = \emptyset$.

Доказательство. Следует непосредственно из определений LTS $\Gamma^0_{\pi',\pi''}$, и множества вершин $\mathrm{V}^0_{\pi',\pi''}$, $R^0_{\pi',\pi''}$. Детерминированные правильные преобразователи π' и π'' неэквивалентны тогда и только тогда, когда для некоторого слова w выполняется хотя бы одно из двух условий:

1) $(w, s') \in Lab(\pi')$, $(w, s'') \in Lab(\pi'')$, и при этом $s' \neq s''$;
2) Для одного из преобразователей, множество его меток содержит пару (w, s), тогда как другой преобразователь не имеет никаких образов для слова w.

Первое из указанных условий выполняется в том и только том случае, если из стартовой вершины LTS $\Gamma^0_{\pi',\pi''}$, по маршруту с пометкой w достижима вершина (q_1, q_2, g), где q_1 и q_2 – финальные состояния преобразователей π' и π'', и при этом $g = (s')^- s'' \neq e$. Второе из указанных условий выполняется в том и только том случае, если либо из стартовой вершины LTS $\Gamma^0_{\pi',\pi''}$ по маршруту с пометкой w достижима некоторая вершина (q_1, q_2, g), в которой только одно из двух состояний q_1, q_2 является финальным, либо $w = w'aw''$, и при этом из стартовой вершины LTS $\Gamma^0_{\pi',\pi''}$ по маршруту с пометкой w' достижима некоторая вершина (q_1, q_2, g), в которой только одно из двух состояний q_1, q_2 имеет -переход.

QED

Таким образом, проверка эквивалентности детерминированных преобразователей сводится к проверке достижимости опровергающих вершин из стартовой вершины LTS $\Gamma^0_{\pi',\pi''}$. Следующая лемма показывает, что для проверки достижимости опровергающих вершин достаточно исследовать лишь ограниченный фрагмент LTS.

Лемма 2. Если множество $V^0_{\pi',\pi''}$ содержит пару вершин $v_1 = (q_1', q_1'', g_1)$ и $v_2 = (q_1', q_1'', g_2)$, удовлетворяющую соотношению $g_1 \neq g_2$, то $\mathrm{V}^0_{\pi',\pi''} \cap R^0_{\pi',\pi''} \neq \emptyset$.

Доказательство. Предположим, что, вопреки утверждению леммы, $\mathrm{V}^0_{\pi',\pi''} \cap R^0_{\pi',\pi''} = \emptyset$, и, следовательно, $\pi' \sim \pi''$. По определению LTS $\Gamma^0_{\pi',\pi''}$ достижимость вершины v_1 означает, что существует такое слово w_0, для которого имеются начальные вычисления $q_0' \xrightarrow{w_0/s_0'} q_1'$ и $q_0'' \xrightarrow{w_0/s_0''} q_1''$, и при этом $g_1 = (s_0')^- s_0''$. Поскольку состояние q_1' является полезным, существует такое слово w, для которого вычисление $q_0' \xrightarrow{w_0/s_0'} q_1' \xrightarrow{w/s'} q_3'$ является полным вычислением преобразователя π'. Предполагая, что $\pi' \sim \pi''$, и принимая во внимание детерминированность преобразователя π'', приходим к заключению о том, что преобразователь π'' имеет такое полное вычисление $q_0'' \xrightarrow{w_0/s_0''} q_1'' \xrightarrow{w/s''} q_3''$, для которого $s_0' s' = s_0'' s''$. Значит, $(s_0')^- s_0'' = s'(s'')^-$, и поэтому $g_1 = s'(s'')^-$. По определению LTS $\Gamma^0_{\pi',\pi''}$ существование вычислений $q_1' \xrightarrow{w/s'} q_3'$ и $q_1'' \xrightarrow{w/s''} q_3''$ означает, что в LTS имеется маршрут $v_2 \overset{w}{\Rightarrow} (q_3', q_3'', g)$, причем $g = (s')^- g_2\, s''$. Учитывая, что (q_3', q_3'', g) принадлежит множеству $\mathrm{V}^0_{\pi',\pi''}$, а также то, что оба состояния q_3' и q_3'' являются финальными, и предполагая, что $\mathrm{V}^0_{\pi',\pi''} \cap R^0_{\pi',\pi''} = \emptyset$, мы приходим к равенству $g = e$. Следовательно, $g_2 = s'(s'')^- = g_1$, а это противоречит условию данной леммы. QED

Из лемм 1 и 2 вытекает, что для проверки эквивалентности правильных детерминированных преобразователей π' и π'' достаточно проанализировать не более $|Q'||Q''|+1$ вершин, достижимых из стартовой вершины LTS $\Gamma^0_{\pi',\pi''}$. Итак, мы приходим к следующей теореме.

Теорема 1. Если полугруппа S вложима в конечно порожденную разрешимую группу G, то проблема эквивалентности детерминированных преобразователей над S разрешима. Кроме того, если проблема равенства слов в группе G разрешима за полиномиальное время, то и проблема эквивалентности детерминированных преобразователей над S разрешима за полиномиальное время.

4. Проверка эквивалентности функциональных преобразователей

Для проверки того, является ли преобразователь $\pi = \langle A, S, Q, q_0, F, T\rangle$ функциональным, мы воспользуемся размеченными системами переходов подобно тому, как это было сделано для проверки эквивалентности детерминированных преобразователей. Но для недетерминированных преобразователей соответствующие LTS нуждаются в некоторых изменениях.

Пусть заданы преобразователи $\pi' = \langle A, S, Q', q'_0, F', T'\rangle$ и $\pi'' = \langle A, S, Q'', q''_0, F'', T''\rangle$ над полугруппой S, которая вложена в конечно порожденную разрешимую группу G. Отношение переходов в LTS $\Gamma^1_{\pi',\pi''} = \langle\, Q' \times Q'' \times G, \Rightarrow \rangle$ определим так: для каждой пары вершин $v_1 = (q'_1, q''_1, g_1)$ и $v_2 = (q'_2, q''_2, g_2)$ и буквы a отношение $v_1 \overset{a}{\Rightarrow} v_2$ имеет место тогда и только тогда, когда в преобразователях π' и π'' есть переходы $q'_1 \xrightarrow{a/s'} q'_2$ и $q''_1 \xrightarrow{a/s''} q''_2$ соответственно, и при этом $g_2 = (s')^- g_1 s''$ и $L(M_{\pi'}[q'_2]) \cap L(M_{\pi''}[q''_2]) \neq \emptyset$. Множество всех вершин LTS $\Gamma^1_{\pi',\pi''}$, достижимых из стартовой вершины (q'_0, q''_0, e) обозначим записью $V^1_{\pi',\pi''}$. Будем говорить, что вершина (q_1, q_2, g) является опровергающей, если q_1 и q_2 – финальные состояния и при этом $g \neq e$. Множество всех опровергающих вершин LTS $\Gamma^1_{\pi',\pi''}$ обозначим записью $R^1_{\pi',\pi''}$. Для проверки свойства функциональности преобразователя π рассмотрим LTS $\Gamma^1_{\pi,\pi}$ и множества вершин $V^1_{\pi,\pi}$ и $R^1_{\pi,\pi}$.

Лемма 3. Правильный преобразователь π является однозначным тогда и только тогда, когда $V^1_{\pi,\pi} \cap R^1_{\pi,\pi} = \emptyset$.

Доказательство. Проводится также, как и доказательство леммы 1 на основании определений LTS $\Gamma^1_{\pi,\pi}$ и множеств $V^1_{\pi,\pi}$ и $R^1_{\pi,\pi}$ QED

Лемма 4. Если множество $V^1_{\pi,\pi}$ содержит такую пару вершин $v_1 = (q'_1, q''_1, g_1)$ и $v_2 = (q'_1, q''_1, g_2)$, для которой имеет место соотношение $g_1 \neq g_2$, то $V^1_{\pi,\pi} \cap R^1_{\pi,\pi} \neq \emptyset$.

Доказательство. Проводится также, как и доказательство леммы 2. QED

Из лемм 3 и 4 вытекает, что для проверки свойства функциональности преобразователя π достаточно рассмотреть не более $|Q|^2 + 1$ вершин, достижимых из стартовой вершины LTS $\Gamma^1_{\pi,\pi}$.

Теорема 2. Если полугруппа S вложима в конечно порожденную разрешимую группу G, то свойство функциональности преобразователя над полугруппой S разрешимо. Кроме того, если проблема равенства слов в группе G разрешима за полиномиальное время, то и проверку свойства функциональности преобразователей над S можно провести за полиномиальное время.

Эквивалентность функциональных преобразователей π' и π'' можно проверить точно так же, как и свойство функциональности, при помощи LTS $\Gamma^1_{\pi',\pi''}$. Но для этого нам вначале нужно убедиться, что $L(M_{\pi'}[q'_0]) = L(M_{\pi''}[q''\_0])$. Проверку этого соотношения необходимо проводить потому, что в случае функциональных преобразователей, в отличие от детерминированных преобразователей, достижимость вершин (q_1, q_2, g), в которой одно из состояний q_σ, $\sigma \in \{1,2\}$, является финальном, а другое нет, не является признаком неэквивалентности. Это, в частности, означает, что задача проверки эквивалентности для функциональных преобразователей является PSPACE-трудной независимо от того, над какой полугруппой работают эти преобразователи.

Лемма 5. Если $L(M_{\pi'}[q'_0]) = L(M_{\pi''}[q''_0])$, то функциональные преобразователи π' и π'' эквивалентны тогда и только тогда, когда $V^1_{\pi',\pi''} \cap R^1_{\pi',\pi''} = \emptyset$.

Лемма 6. Если множество $V^1_{\pi',\pi''}$ содержит такую пару вершин $v_1 = (q'_1, q''_1, g_1)$ и $v_2 = (q'_2, q''_2, g_2)$, для которой $g_1 \neq g_2$, то $V^1_{\pi',\pi''} \cap R^1_{\pi',\pi''} \neq \emptyset$.

Из лемм 5 и 6 вытекает, что для проверки эквивалентности функциональных преобразователей π' и π'' достаточно рассмотреть не более $|Q'||Q''| + 1$ вершин, достижимых из стартовой вершины LTS $\Gamma^1_{\pi',\pi''}$. Таким образом, верна

Теорема 3. Если полугруппа S вложима в конечно порожденную разрешимую группу G, то проблема эквивалентности для функциональных преобразователей над полугруппой S разрешима. Кроме того, если проблема равенства слов в группе G разрешима за полиномиальное время, то проблема эквивалентности функциональных преобразователей над S является PSPACE-полной.

5. Проверка свойства двухзначности моделей реагирующих программ

Метод анализа поведения преобразователей, основанный на размеченных системах переходов, предложенный в предыдущих разделах статьи для проверки эквивалентности детерминированных и функциональных преобразователей, можно развить далее для проверки свойств

конечнозначных преобразователей. Для простоты изложения мы ограничимся рассмотрением 2-значных преобразователей.

Вначале обратимся к задаче проверки свойства 2-значности конечных преобразователей, работающих над полугруппами, которые вложены в группы. Для заданного преобразователя $\pi = \langle A, S, Q, q_0, F, T\rangle$ определим LTS $\Gamma_\pi^2 = \langle Q \times (Q \times G)^2, \Rightarrow \rangle$ следующим образом: для каждой пары вершин $v_1 = (q_1, (q_2, g_{12}), (q_3, g_{13}))$ и $v_2 = (q_1', (q_2', g_{12}'), (q_3', g_{13}'))$, и произвольной буквы a отношение $v_1 \underset{a}{\Rightarrow} v_2$ имеет место тогда и только тогда, когда в преобразователе π существуют такие переходы $q_1 \xrightarrow{a/s_1} q_1'$, $q_2 \xrightarrow{a/s_2} q_2'$, и $q_3 \xrightarrow{a/s_3} q_3'$, для которых выполняются равенства $g_{12}' = (s_1)^{-} g_{12} s_2$ и $g_{13}' = (s_1)^{-} g_{13} s_3$, и при этом $L(M_\pi[q_1']) \cap L(M_\pi[q_2']) \cap L(M_\pi[q_3']) \neq \emptyset$.

Тройку состояний (q_1, q_2, q_3) будем называть *типом* вершины $(q_1, (q_2, g_{12}), (q_3, g_{13}))$. Как и в случае $k = 1$, обозначим записью V_π^2 множество всех вершин LTS Γ_π^2, достижимых из стартовой вершины $(q_0, (q_0, e), (q_0, e))$. Как следует из определений LTS Γ_π^2 и множества V_π^2, вершина $v = (q_1, (q_2, g_{12}), (q_3, g_{13}))$ содержится в множестве V_π^2 тогда и только тогда, когда для некоторого слова w преобразователь π имеет вычисления $q_0 \xrightarrow{w/s_1} q_1$, $q_0 \xrightarrow{w/s_2} q_2$, $q_0 \xrightarrow{w/s_3} q_3$, удовлетворяющие равенствам $g_{12} = (s_1)^{-} s_2, g_{13} = (s_1)^{-} s_3$. Поэтому, если некоторая вершина $v = (q_1, (q_2, g_{12}), (q_3, g_{13}))$ содержится в множестве V_π^2, то в этом же множестве содержатся так же и вершины $(q_1, (q_1, e), (q_3, g_{13}))$, $(q_1, (q_2, g_{12}), (q_1, e))$, $(q_1, (q_2, g_{12}), (q_2, g_{12}))$, и т. д..

Множество опровергающих вершин R_π^2 состоит из всех тех вершин $(q_1, (q_2, g), (q_3, h))$, для которых состояния q_1, q_2, q_3 являются финальными, и при этом $g \neq e, h \neq e, g \neq h$.

Лемма 7. Преобразователь π является 2-значным тогда и только тогда, когда $V_\pi^2 \cap R_\pi^2 = \emptyset$.

Доказательство. Следует непосредственно из определений множеств V_π^2, R_π^2, а также определения свойства 2-значности конечных преобразователей. QED

В общем случае множество достижимых вершин V_π^2 бесконечно, и поэтому, чтобы воспользоваться леммой 7 для построения эффективной процедуры проверки свойства 2-значности преобразователя, нужно каким-то образом выделить в этом множестве ограниченный фрагмент, анализ которого позволил бы решить вопрос о выполнимости равенства $V_\pi^2 \cap R_\pi^2 = \emptyset$. Для этой цели воспользуемся следующими тремя леммами, доказательство которых по существу опирается на принцип Дирихле и простейшие тождества теории групп.

Лемма 8. Предположим, что множество V_π^2 содержит четыре вершины $v_i = \big(q', (q'', h_i), (q''', g_i)\big)$, $1 \le i \le 4$, одного и того же типа, удовлетворяющие соотношениям $h_i \neq h_j, g_i \neq g_j$ и $h_i g_i^- \neq h_j g_j^-$ для каждой пары индексов $i, j, 1 \le i \le 4$. Тогда $V_\pi^2 \cap R_\pi^2 \neq \emptyset$.

Доказательство. Поскольку все вершины $v_i, 1 \le i \le 4$, принадлежат множеству V_π^2, верно соотношение $L(M_\pi[q']) \cap L(M_\pi[q'']) \cap L(M_\pi[q''']) \neq \emptyset$. Значит, есть такое слово w, для которого преобразователь π имеет финальные вычисления $q' \xrightarrow{w/s'} p'$, $q'' \xrightarrow{w/s''} p''$ и $q''' \xrightarrow{w/s'''} p'''$. Тогда, по определению LTS Γ_π^2, множество V_π^2 содержит четыре вершины $u_i = (p', (p'', (s')^- h_i\, s''), (p''', (s')^- g_i s'''))$, $1 \le i \le 4$. Если вершина u_1 не является опровергающей, то верно одно из трех равенств: $(s')^-\, h_1\, s'' = e, (s')^- g_1 s''' = e$, or $(s')^- h_1 s'' = (s')^- g_1 s'''$. Не ограничивая общности, рассмотрим лишь случай $(s')^- h_1 s'' = e$ (в двух других случаях рассуждения проводятся аналогичным образом). Так как $h_1 \neq h_2$, справедливо соотношение $(s')^- h_2 s'' \neq e$. Значит, если u_2 не является опровергающей вершиной, то верно одно из двух равенств $(s')^- g_2 s''' = e$ или $(s')^- h_2 s'' = (s')^- g_2 s'''$. Рассмотрим случай $(s')^- g_2 s''' = e$ (те же самые рассуждения можно провести и в случае другого равенства). Так как $h_1 \neq h_3$ и $g_2 \neq g_3$, из равенств $(s')^- h_1 s'' = e$ и $(s')^- g_2 s''' = e$ следует соотношение $(s')^- h_3 s'' \neq e$ и $(s')^- g_3 s''' \neq e$. Поэтому, если u_3 не является опровергающей вершиной, то $(s')^- h_3 s'' = (s')^- g_3 s'''$. Ввиду того, что $h_1 \neq h_4, g_2 \neq g_4$ и $h_3 g_3^- \neq h_4 g_4^-$, мы приходим к заключению, что следствием равенств $(s')^- h_1 s'' = e, (s')^- g_2 s''' = e$ и $(s')^- h_3 s'' = (s')^- g_3 s'''$ являются соотношения $(s')^- h_4 s'' \neq e, (s')^- g_4 s''' \neq e$ и $(s')^- h_4 s'' \neq (s')^- g_4 s'''$. Согласно определению опровергающей вершины, это означает, что $v_4 \in R_\pi^2$. QED

Лемма 9. Предположим, что в LTS Γ_π^2 есть четыре различные вершины $v_i = (q', (q'', g'')), (q'', g_i''')\big)$, $1 \le i \le 4$, удовлетворяющие одному из следующих трех условий:

a) равенство $g_i'' = g_j''$ выполняется для любой пары индексов $i, j,\ 1 \le i < j \le 4$;

b) равенство $g_i''' = g_j'''$ выполняется для любой пары индексов $i, j,\ 1 \le i < j \le 4$;

c) равенство $(g_i'')^- g_i''' = \big(g_j''\big)^- g_j'''$ выполняется для любой пары индексов $i, j,\ 1 \le i < j \le 4$.

Тогда, если опровергающая вершина достижима из вершины v_4, то некоторая вершина также достижима из одной из вершин v_1, v_2, v_3.

Доказательство. Ограничимся рассмотрением случая, когда все вершины удовлетворяют первому из перечисленных условий; аналогичные рассуждения применимы и в случае выполнимости любого из двух других условий.

Предположим, что в LTS Γ_π^2 существует путь, помеченный словом w и позволяющий достичь опровергающую вершину $u_4 = (p', (p'', h''), (p''', h_4'''))$ из вершины v_4. Тогда преобразователь π имеет три таких финальных вычисления $q' \xrightarrow{w/s'} p'$, $q'' \xrightarrow{w/s''} p''$и $q''' \xrightarrow{w/s'''} p'''$, для которых выполняются равенства $h'' = (s')^- g'' s''$ и $h_4''' = (s')^- g_4''' s'''$. Поскольку вершина u_4 является опровергающей, должно быть соблюдено соотношение $h'' \neq e$.

По определению LTS Γ_π^2 для каждого значения индекса $i, 1 \leq i \leq 3$, в этой системе переходов есть путь из вершины v_i в такую вершину $u_i = (p', (p'', h''), (p''', h_i'''))$, для которой имеет место равенство $h_i''' = (s')^- g_i''' s'''$. Если $u_1 \notin R_\pi^2$, то либо $h_1''' = e$, либо $(h'')^- h''' = e$. Проанализируем случай $h_1''' = e$ (в другом случае мы также можем воспользоваться аналогичными рассуждениями). Ввиду того, что $g_2''' \neq g_1'''$ и $g_1''' \neq g_3'''$, верны соотношения $h_2''' \neq e$ и $h_3''' \neq e$. Значит, если $u_2 \notin R_\pi^2$, то $(h'')^- h_2''' = e$. Однако, коль скоро $g_2''' \neq g_3'''$, справедливо соотношение $(h'')^- h_3''' \neq e$, и в результате мы приходим к заключению о том, вершина u_3 является опровергающей. QED

При помощи лемм 8 и 9 можно доказать следующую теорему.

Теорема 4. Если полугруппа S вложим в конечно порожденную разрешимую группу G, то свойство 2-значности конечных преобразователей над полугруппой S разрешимо. Кроме того, если проблема равенства слов в группе G разрешима за полиномиальное время, то и свойство 2-значности конечных преобразователей над S разрешима за полиномиальное время.

Доказательство. Как следует из леммы 7, для проверки 2-значности преобразователя π достаточно провести проверку достижимости опровергающих вершин из стартовой вершины в LTS Γ_π^2. Поиск опровергающих вершин можно проводить методом обхода вершин LTS по стратегии поиска в глубину с возвратом. Обход начинается из стартовой вершины $(q_0, (q_0, e), (q_0, e))$; при этом продвижение в глубину проводится только из значимых вершин, которые определяются на основании леммы 9. Указанный обход вершин LTS Γ_π^2 может быть прерван досрочно, если будет обнаружено, что выполнены условия леммы 8. В этом случае факт достижимости опровергающей вершины устанавливается без явного предъявления пути, ведущего в нее из стартовой вершины.

Предположим, что на некотором этапе предложенного обхода LTS Γ_π^2 была достигнута ранее не встречавшаяся вершина $v = (q', (q'', g''), (q''', g'''))$. Возможны следующие четыре альтернативы.

1) Если вершина v является опровергающей, то обход прекращается, и выносится вердикт о том, что преобразователь π не является 2-значным.

2) В противном случае, если ранее были пройдены 3 значимых вершины $v_i = \left(q', (q'', g_i''), (q''', g_i''')\right)$, $1 \leq i \leq 3$, одного и того же типа, для которых выполняется одно из трех следующих условий:

а) $g'' = g_i''$ для каждого значения индекса i, $1 \leq i \leq 3$;

b) $g''' = g_i'''$ для каждого значения индекса $i,\ 1 \le i \le 3$;

c) $(g'')^- g''' = (g_i'')^- g_i'''$ для каждого значения индекса $i,\ 1 \le i \le 3$,

то вершина v объявляется незначимой, и из нее совершается откат к ее предшественнику в этом обходе.

3) В противном случае, если ранее были пройдены 27 значимых вершин $v_i = (q', (q'', g_i''), (q''', g_i'')), i,\ 1 \le i \le 27$, того же самого типа, что и вершина v, то обход прекращается, и выносится вердикт о том, что преобразователь π не является 2-значным.

4) В противном случае вершина v объявляется значимой, и процедура обхода LTS Γ_π^2 продолжается из этой вершины.

Если процедура обхода завершается в стартовой вершине, преобразователь π признается 2-значным.

Как видно из приведенного описания процедуры обхода, она всегда завершается после посещения не более $27|Q|^3$ значимых вершин LTS Γ_π^2. Лемма 9 обеспечивает гарантию того, что игнорирование незначимых вершин не приводит к потере возможности достичь опровергающую вершину. Это обстоятельство гарантирует полноту нашей процедуры поиска. Чтобы убедиться в корректности этой процедуры нужно показать, что третья альтернатива приводит к правильному решению. Действительно, как следует из простых комбинаторных соображений, если имеются 28 вершин (v и v_i, $1 \le i \le 27$), и из которых никакие четыре вершины не удовлетворяют ни одному из условий леммы 9 (а это объясняется тем, что все пройденные в нашем обходе вершины являются значимыми), то в этом множестве вершин найдутся такие четыре вершины, которые удовлетворяют предпосылкам леммы 8. QED

Обход LTS Γ_π^2 может быть еще более сокращен, если воспользоваться следующей леммой.

Лемма 10. Пусть $q \in \{q', q'', q'''\}$. Предположим, что существует такое входное слово w, принадлежащее множеству слов $L(M_\pi[q']) \cap L(M_\pi[q'']) \cap L(M_\pi[q'''])$, для которого преобразователь π имеет два финальных вычисления $q \xrightarrow{w/s_1} p_1$ и $q \xrightarrow{w/s_2} p_2$, удовлетворяющих соотношению $s_1 \ne s_2$. Тогда если в множестве V_π^2 есть пять вершин $v_i = (q', (q'', g_i), (q''', h_i)), 1 \le i \le 5$, то $V_\pi^2 \cap R_\pi^2 \ne \emptyset$.

Доказательство. Для определенности будем считать, что $q = q'$. Так как все вершины $v_i,\ 1 \le i \le 5$, принадлежат множеству V_π^2, это множество содержит также 10 вершин $u_{1i} = (q', (q', e), (q'', g_i)),\ u_{2i} = (q', (q', e), (q''', h_i)), 1 \le i \le 5$. Коль скоро слово w принадлежит множеству $M_\pi[q']) \cap L(M_\pi[q'']) \cap L(M_\pi[q'''])$, преобразователь π имеет финальные вычисления $q'' \xrightarrow{w/s''} p''$, $q''' \xrightarrow{w/s'''} p'''$. Таким образом, из стартовой вершины LTS Γ_π^2 достижимы следующие 10 вершин: $u_i'' = (p_1, (p_2, s_1^- s_2), (p'', s_1^- g_i s''))$ и $u_i''' = (p_1, (p_2, s_1^- s_2), (p''', s_1^- h_i))$, где $1 \le i \le 5$.

Предположим, что ни одна из вершин $u_i'', u_i''', 1 \le i \le 4$, не является опровергающей. Следует заметить, что $s_1 \neq s_2$, и, следовательно, $s_1^- s_2 \neq e$. Поэтому, по определению опровергающей вершины, для каждого i, $1 \le i \le 4$, элемент g_i группы G либо равен $s_1(s'')^-$, либо равен $s_2(s'')^-$, а элемент h_i либо равен $s_1(s''')^-$, либо равен $s_2(s''')^-$. Принимая во внимание, что вершины $v_i, 1 \le i \le 5$, попарно различны, мы приходим к выводу о том, что ни одно из двух равенств $g_5 = s_1(s'')^-$ и $g_5 = s_2(s'')^-$ не может быть выполнено. Таким образом, $u_5'' = (p_1', (p_2', s_1^- s_2), (p'', s_1^- g_5 s''))$ является опровергающей вершиной, достижимой из стартовой вершины LTS Γ_π^2. QED

Чтобы проверить основное требование предпосылки леммы 10 – существование двух финальных вычислений $q \xrightarrow{w/s_1} p_1$ и $q \xrightarrow{w/s_2} p_2$ с разными образами, – можно воспользоваться алгоритмом проверки свойств функциональности конечных преобразователей. Условия леммы 10 выполнены для состояния q тогда и только тогда, когда проекция преобразователя $\pi[q]$ на конечный автомат, распознающий язык $M_\pi[q']) \cap L(M_\pi[q'']) \cap L(M_\pi[q''']),$ обладает свойством функциональности. Таким образом, алгоритм обхода LTS Γ_π^2, описанный в доказательстве теоремы 4 можно модифицировать, введя следующую дополнительную альтернативу:

3') В противном случае, если ранее были пройдены 4 значимых вершины $v_i = \left(q', (q'', g_i''), (q''', g_i''')\right), 1 \le i \le 4$, одного и того же типа, для которых выполняется условие леммы 10, то обход прекращается, и выносится вердикт о том, что преобразователь π не является 2-значным.

Предложенный метод проверки 2-значности недетерминированности преобразователей – леммы 9, 10 и процедуру поиска опровергающих вершин, описанную в доказательстве теоремы 4, – можно использовать и для проверки -значности конечных преобразователей при любом $k > 1$. В этом случае вершинами LTS служат наборы вида $(q_0, (q_1, h_1), \dots, (q_k, h_k))$, и для проверки достижимости опровергающей вершины в LTS Γ_π^2 достаточно совершить обход не более $\binom{k+1}{2}^{\binom{k+1}{2}} |Q|^{k+1} + 1$ значимых вершин.

6. Проверка эквивалентности двухзначных моделей реагирующих программ

Вместо решения проблемы эквивалентности для конечных преобразователей мы исследуем более общую задачу проверки включения: для заданной пары преобразователей π и π' проверить выполнимость включения $Lab(\pi') \subseteq Lab(\pi)$. Решение этой задачи также проведем на основе размеченных систем переходов.

Пусть задана пара 2-значных преобразователей $\pi = \langle A, S, Q, q_0, F, T\rangle$ и $\pi' = \langle A, S, Q', q_0', F', T'\rangle$. Ясно, что $Lab(\pi') \subseteq Lab(\pi)$ влечет $L(M_{\pi'}) \subseteq L(M_\pi)$. Поэтому проверку включения преобразователей начнем с проверки

включения соответствующих автоматов-подложек, и далее в этом разделе будем считать, не оговаривая этого особо всякий раз, что для анализируемых преобразователей π и π' имеет место включение $L(M_{\pi'}) \subseteq L(M_\pi)$.

Чтобы дать определение LTS $\Gamma^3_{\pi,\pi'}$, соответствующей проблеме включения для преобразователей π и π', нам понадобится ввести вспомогательное понятие блока состояний. Рассмотрим некоторое мультимножество состояний $\hat{Q}$ преобразователя π. *Блоком состояний* в мультимножестве $\hat{Q}$ называется всякое максимальное по включению (т.е. нерасширяемое) подмножество B множества $\hat{Q}$, удовлетворяющее условию $\bigcap_{q\in B} L(M_\pi[q] \neq \emptyset)$, которое подразумевает существование хотя бы одного слова, которое допускается каждым автоматом $M_\pi[q], q \in B$.

LTS $\Gamma^3_{\pi,\pi'} = \langle V, \Rightarrow \rangle$ определяется следующим образом. Множество вершин V образуют всевозможные пары $u = (q', X)$, где $q' \in Q'$, $X = \{(q_1, g_1), \dots, (q_m, g_m)\} \subseteq Q \times G$, и при этом выполнено условие $L(M_{\pi'}\{q'\]) \cap \bigcap_{i=1}^{m} L(M_\pi[q_i]) \neq \emptyset$. Пару $(q', \{q_1, \dots, q_m\})$ будем называть типом вершины u. Для каждой буквы a и пары вершин $u = (q', X)$ и $v = (p', Y)$, имеющих типы (q', B_u) и (p', B_v) соответственно, переход $u \overset{a}{\Rightarrow} v$ возможен тогда и только тогда, когда выполнены следующие требования:

1) в преобразователе π' есть переход $q' \xrightarrow{a/s'} p'$,
2) B_v – это блок состояний в мультимножестве $\hat{Q} = \{\hat{q} : \exists\, q\ (q \in B_u \wedge q \xrightarrow{a/s} \hat{q})\}$,
3) пара (p, h) принадлежит множеству Y в том и только том случае, если $p \in B_v$ и при этом для некоторой пары (q, g) из множества X в преобразователе π есть переход $q \xrightarrow{a/s} p$, для которого верно равенство $h = (s')^- g s$.

Как обычно, для любого заданного слова w мы будем обозначать записью $u \overset{w}{\Rightarrow} v$ композицию соответствующих однобуквенных переходов в LTS. Вершина $v_{src} = (q'_0, \{(q_0, e)\})$ объявляется стартовой вершиной LTS $\Gamma^3_{\pi,\pi'}$. Запись $\mathrm{V}^3_{\pi,\pi'}$ служит обозначением множества всех вершин, достижимых из стартовой вершины v_{src}. Опровергающей вершиной назовем всякую вершину (q', X), в которой $q' \in F'$, и для каждой пары (q, g) из X либо $q \notin F$, либо $g \neq e$. Множество всех опровергающих вершин LTS $\Gamma^3_{\pi,\pi'}$ обозначим записью $\mathrm{R}^3_{\pi,\pi'}$.

Содержательный смысл LTS $\Gamma^3_{\pi,\pi'}$ применительно к задаче проверки включения преобразователей π и π' проясняют следующие утверждения.

Утверждение 1. Пусть w_0 и w_1 – произвольные слова, и $q_0' \xrightarrow{w_0/s_0'} q_1' \xrightarrow{w_1/s_1'} q_2'$ – полное вычисление преобразователя π'. Тогда существует такая вершина $v = (q_1', X)$, что $v_{src} \overset{w_0}{\Rightarrow} v$, и для каждого полного вычисления $q_0 \xrightarrow{w_0/s_0} q_1 \xrightarrow{w_1/s_1} q_2$ преобразователя π мультимножество X содержит пару $(q_1, (s_0')^{-}s_0)$.

Утверждение 2. Предположим, что $v_{src} \overset{w_0}{\Rightarrow} (q_1', X)$. Тогда существует такое слово w_1 и полное вычисление $q_0' \xrightarrow{w_0/s_0'} q_1' \xrightarrow{w_1/s_1'} q_2'$ преобразователя π', что для любого полного вычисления $q_0 \xrightarrow{w_0/s_0} q_1 \xrightarrow{w_1/s_1} q_2$ преобразователя π мультимножество X содержит пару $(q_1, (s_0')^{-}s_0)$.

Оба эти утверждения нетрудно доказать, воспользовавшись индукцией по длине слова w_0 и опираясь лишь на определение отношения переходов $\Rightarrow$ в LTS $\Gamma^3_{\pi,\pi'}$. Здесь нужно особо отметить, что справедливость приведенных утверждений обусловлена тем фактом, что тип каждой вершины связан именно с блоком состояний как максимальным множеством пар, удовлетворяющих определенному условию.

Лемма 11. $Lab(\pi') \subseteq Lab(\pi) \Leftrightarrow \mathrm{V}^3_{\pi,\pi'} \cap \mathrm{R}^3_{\pi,\pi'} = \emptyset$.

Доказательство. Следует из утверждений 1 и 2 на основании определения опровергающей вершины LTS $\Gamma^3_{\pi,\pi'}$. QED

Мы покажем, что даже в том случае, когда множество $\mathrm{V}^3_{\pi,\pi'}$ оказывается бесконечным, нужно исследовать только конечный его фрагмент, чтобы убедиться в (не)достижимости опровергающих вершин.

Рассмотрим произвольную достижимую вершину v типа (q', B). Поскольку π – это 2-значный преобразователь, для каждого его состояния q в мультимножестве B может содержаться не более двух копий состояний q. Поэтому, $|B| \leq 2|Q|$, и общее число типов вершин LTS $\Gamma^3_{\pi,\pi'}$, достижимых из стартовой вершины, не превосходит величины $|Q'|3^{|Q|}$.

Рассмотрим язык $L = L(M_{\pi'}[q']) \cap \bigcap_{q \in B} L(M_{\pi}[q])$; будем называть его языком типа (q', B). По определению LTS $\Gamma^3_{\pi,\pi'}$, этот язык непуст. Семейство типов всех достижимых вершин можно разделить на три класса в зависимости от свойств языка L. Тип (q', B) будет называться A-типом в том случае, когда язык L содержит такое слово w, которое имеет два разных образа s_1' и s_2' в двух финальных вычислениях $q' \xrightarrow{w/s_1'} p_1'$ и $q' \xrightarrow{w/s_2'} p_2'$ преобразователя π'. Тип (q', B) будет называться B-типом в том случае, когда он не относится к классу A, и при этом в мультимножестве B существует такое состояние q, а в языке L найдется такое слово w, которое имеет два разных образа s_1 и s_2 в двух финальных вычислениях $q \xrightarrow{w/s_1} p_1$ и $q \xrightarrow{w/s_2} p_2$ преобразователя π. Все остальные типы будем называть C-типами. Леммы, которые приводятся далее,

раскрывают некоторых характерные свойства этих классов, важные для решения проблемы включения.

Лемма 12. Предположим $Lab(\pi') \subseteq Lab(\pi)$, и пара (q', B) является A-типом. Тогда из стартовой вершины LTS $\Gamma^3_{\pi,\pi'}$ достижимо не более $2^{|B|}$ вершин этого типа.

Доказательство. Рассмотрим язык L типа (q', B), произвольную вершину $v = (q', X)$ типа (q', B), для которой имеется путь $v_{src} \overset{w_0}{\Rightarrow} v$. Пусть (q, g) – произвольная пара из множества X. Так как тип (q', B) является A-типом, в языке L найдется такое слово w, которое имеет два разных образа s'_1 и s'_2 в двух финальных вычислениях $q' \xrightarrow{w/s'_1} p'_1$ и $q' \xrightarrow{w/s'_2} p'_2$ преобразователя π'. По определению языка L, преобразователь π имеет финальное вычисление $q \xrightarrow{w/s} q_1$. Заметим, что элементы s'_1, s'_2 и s группы G зависят только от типа (q', B) и состояния q. Согласно утверждению 2, преобразователи π и π' имеют такие начальные вычисления $q_0 \xrightarrow{w_0/s_0} q$ и $q'_0 \xrightarrow{w_0/s'_0} q'$, для которых верно равенство $g = (s'_0)^- s_0$. Но тогда преобразователь π' имеет два полных вычисления $q'_0 \xrightarrow{w_0/s'_0} q' \xrightarrow{w/s'_1} p'_1$ и $q'_0 \xrightarrow{w_0/s'_0} q' \xrightarrow{w/s'_2} p'_2$, а преобразователь π имеет полное вычисление $q_0 \xrightarrow{w_0/s_0} q \xrightarrow{w/s} q_1$. Принимая во внимание, что преобразователь π является 2-значным, то, что $s'_0 s'_1 \neq s'_0 s'_2$, а также то, что $Lab(\pi') \subseteq Lab(\pi)$, мы можем быть уверены, что справедливо хотя бы одно из равенств $s_0 s = s'_0 s'_1$ или $s_0 s = s'_0 s'_2$. Следовательно, либо $g = s'_1 s^-$, либо $g = s'_2 s^-$. Таким образом, утверждение леммы следует из того факта, что оба возможных значений g зависят только от типа (q', B) и рассматриваемого состояния q. QED

Лемма 13. Предположим, что $Lab(\pi') \subseteq Lab(\pi)$ и пара (q', B) является B-типом. Тогда из стартовой вершины LTS $\Gamma^3_{\pi,\pi'}$ достижимо не более $3^{|B|}$ вершин типа (q', B).

Доказательство. Рассмотрим язык L типа (q', B) и произвольную вершину $v = (q', X)$ типа (q', B), для которой имеется путь $v_{src} \overset{w_0}{\Rightarrow} v$. Пусть (q, g) – произвольная пара из множества X, для которой есть такое слово w из L, что финальные вычисления $q \xrightarrow{w/s_1} p_1$ и $q \xrightarrow{w/s_2} p_2$ преобразователя π дают разные образы слова w. Выберем произвольную пару (p, h) из множества X. Так как $w \in L$, есть хотя бы одно финальное вычисление $p \xrightarrow{w/s} p_3$ и $q' \xrightarrow{w/s'} p'$ у каждого из преобразователей π и π'. Но тогда в силу утверждения 2 мы приходим к следующим выводам. Ввиду включения $Lab(\pi') \subseteq Lab(\pi)$, справедливо в точности одно из равенств $s' = g s_1$ или $s' = g s_2$. Поскольку преобразователь π является 2-значным, справедливо в точности одно из равенств $g s_1 = h s$ или $g s_2 = h s$. Значит, возможен лишь один из трех вариантов равенства: $h = s' s^-$, $h = s' (s_1)^- s_2 s^-$, или $h = s' (s_1)^- s_2 s^-$.

Тогда утверждение леммы следует из того факта, что во всех трех случаях значение h зависит только от типа (q', B) и выбранных состояний q и p. QED

Предположим, что пара (q', B) относится к С-типу, и $B = \{q_1, \dots, q_m\}$, а L – язык этого типа. Будем ассоциировать с типом (q', B) произвольное слово w_0 из множества L. Рассмотрим финальное вычисление $q' \xrightarrow{w_0/s'} p'$ преобразователя π', а также финальные вычисления $q_i \xrightarrow{w_0/s_i} p_i$ преобразователя π для каждого i, $1 \le i \le m$. Набор $(s', s_1, \dots, s_m)$ элементов полугруппы S будем называть w_0-*характеристикой* типа (q', B). Она будет использована для сокращения поиска опровергающих вершин. Допустим, что вершина $u = (q', \{(q_1, g_1), \dots, (q_m, g_m)\})$ является достижимой вершиной С-типа (q', B). Если соотношение $s' \neq g_i s_i$ соблюдается для каждого $i, 1 \le i \le m$, то, по определению LTS $\Gamma^3_{\pi,\pi'}$, опровергающая вершина достижима из вершины u. Тогда мы будем говорить, что вершина u является предопровергающей вершиной типа (q', B). В противном случае множество X можно разбить на два подмножества $X_0 = \{(q_i, g_i) : s' = g_i s_i, 1 \le i \le m\}$ и $X_1 = \{(q_j, g_j) : s' \neq g_j s_j, 1 \le j \le m\}$ и использовать обозначение $(q', X_0 \oplus X_1)$ для такой вершины u. Заметим, что в силу 2-значности преобразователя π, для любых двух пар (q_i, g_i), (q_j, g_j) из X_1 верно равенство $g_i s_i = g_j s_j$.

Лемма 14. Пусть пара (q', B) относится к С-типу, $B = \{q_1, \dots, q_m\}$, и $k = 2m$. Предположим, что из стартовой вершины LTS $\Gamma^3_{\pi,\pi'}$ достижимы $k+1$ вершин $u_1 = (q', X_0 \oplus X_{11}), \dots, u_{k+1} = (q', X_0 \oplus X_{1k+1})$ типа (q', B). Тогда опровергающая вершина достижима из одной из вершин списка $u_1, \dots, u_{k+1}$ в том и только том случае, если опровергающая вершина достижима из какой-либо вершины списка $u_1, \dots, u_k$.

Доказательство. Пусть $(s', s_1, \dots, s_m)$ – характеристика типа (q', B). Предположим, что $X_0 = \{(q_1, g_1), \dots, (q_\ell, g_\ell)\}$ и $X_{1j} = \{(q_{\ell+1}, g_{\ell+1j}), \dots, (q_m, g_{mj})\}$ для каждого j, $1 \le j \le k+1$.

Допустим, что путь $u_{k+1} \overset{w}{\Rightarrow} v$ позволяет достичь опровергающей вершины v для некоторого слова w. Тогда, по определению LTS $\Gamma^3_{\pi,\pi'}$, преобразователь π' имеет финальное вычисление $q' \xrightarrow{w/s'} p'$, и для каждого $i, 1 \le i \le m$, преобразователь π либо не имеет финальных вычислений на слове w из состояния q_i, либо в каждом финальном вычислении $q_i \xrightarrow{w/t_i} p_i$ образ t_i слова w удовлетворяет соотношению $s' \neq g_{ik+1} t_i$ (на самом деле, одно и то же слово может иметь не более двух разных образов t_{i1} и t_{i2} в силу 2-значности преобразователя π). Не умаляя общности мы можем предполагать, что вторая альтернатива осуществляется для каждого состояния $q_i, 1 \le i \le m$. Таким образом, мы располагаем не более чем $2(m-1)$ элементами $t_{i\sigma}$, $\sigma \in \{1,2\}$, полугруппы S, которые являются образами одного и того же слова w для всех возможных финальных вычислений преобразователя π из состояний $q_{\ell+1}, \dots, q_m$.

Если опровергающая вершина недостижима из вершины u_1, то для некоторой пары (q_i, g_{i1}) из X_{11} и для некоторого образа t слова w верно равенство $s' = g_{i1}t$, т.е. $g_{i1} = s't^-$. Заметим, что для всякой другой пары (q_j, g_{j1}) имеет место равенство $g_{i1}s_i = g_{j1}s_j$, т.е. $g_{j1} = s't^-s_is_j^-$. Это означает, что образ t полностью определяет все элементы g_{j1}, $\ell + 1 \le j \le m$, из X_{11}. Ясно, что разные образы слова w определяют разные элементы в различных множествах X_{1i}. Так как число разных образов слова w не превосходит величины $2(m-1) < k$, существует такая вершина u_i, $1 \le i \le k$, для которой соотношение $s' \ne g_{ji}t_{j\sigma}$ соблюдается для каждой компоненты (q_j, g_{ji}) множества X_{1i} и образов $t_{j\sigma}$ слова w. Последнее означает, что опровергающая вершина достижима из этой вершины u_i. QED

Теорема 5. Если полугруппа S может быть вложена в конечно порожденную разрешимую группу G, то проблема включения $Lab(\pi') \subseteq Lab(\pi)$ для 2-значных преобразователей над полугруппой S разрешима.

Доказательство. Поиск опровергающей вершины в LTS $\Gamma^3_{\pi,\pi'}$ представляет собой обход этой системы переходов, начинающийся в стартовой вершине v_{src}. Предположим, что на некотором шаге этот обход достиг ранее не пройденной вершины $u = (q', X)$ типа (q', B). Тогда возможны шесть следующих альтернатив продолжения этого обхода.

1) Если $u \in \mathrm{R}^3_{\pi,\pi'}$, то обход прекращается, и объявляется о том, что преобразователь π не включает преобразователь π' (отношение $Lab(\pi') \subseteq Lab(\pi)$ не выполняется).

2) В противном случае, если тип (q', B) относится к A-типу и ранее были пройдены $2^{|B|}$ вершин того же типа, то обход прекращается и объявляется о том, что преобразователь π не включает преобразователь π'.

3) В противном случае, если тип (q', B) относится к B-типу и ранее были пройдены $3^{|B|}$ вершин того же типа, то обход прекращается и объявляется о том, что преобразователь π не включает преобразователь π'.

4) В противном случае, если тип (q', B) относится к C-типу и вершина u является пред-опровергающей вершиной этого типа, то обход прекращается и объявляется о том, что преобразователь π не включает преобразователь π'.

5) В противном случае, если тип (q', B) относится к C-типу, $u = (q', X_0 \oplus X_1)$, и $2|B|$ вершин вида $u_i = (q', X_0 \oplus X_{1i})$ уже были пройдены ранее, то из вершины u совершается откат к ее предшественнику в этом обходе.

6) В противном случае обход в глубину LTS Γ^2_π продолжается регулярным образом.

Если при обходе происходит окончательный откат в стартовую вершину, то объявляется о том, что справедливо включение $Lab(\pi') \subseteq Lab(\pi)$.

Завершаемость, корректность и полнота описанной процедуры проверки отношения включения для преобразователей вытекает из лемм 11-14. Из описания процедуры обхода видно, что для проверки включения $Lab(\pi') \subseteq$

$Lab(\pi)$ достаточно проанализировать не более $|Q'|8^{|Q|}$ вершин LTS $\Gamma^3_{\pi,\pi'}$. QED

Следствие.

Если полугруппа S может быть вложена в конечно порожденную разрешимую группу G, то проблема эквивалентности для 2-значных преобразователей над полугруппой S разрешима.

7. Сложность разрешающих алгоритмов

Сложность разрешающих алгоритмов существенно зависит от свойств группы G, и, в частности, от сложности разрешения проблемы равенства слов в группе G. Мы ограничимся рассмотрением двух случаев, когда S является свободной полугруппой и свободной коммутативной полугруппой. Сложность предложенных алгоритмов оценивается относительно следующих параметров, характеризующих размер анализируемых преобразователей π' и π'': n – максимальное число состояний преобразователей π' и π'', m – максимальное число переходов в преобразователях π' и π'', ℓ – максимальная длина полугруппового выражения s в переходах $q \xrightarrow{a/s} q'$ преобразователей π' и π''.

Всякая конечно порожденная свободная полугруппа S может быть вложена в свободную группу G с тем же множеством образующих. Каждое выражение g составленное из порождающих элементов свободной группы может быть задано двоичной строкой длины $O(|g|)$.

Всякая конечно порожденная свободная коммутативная полугруппа S может быть вложена в абелеву группу G с тем же множеством образующих. Каждое выражение g составленное из порождающих элементов свободной группы может быть задано двоичной строкой длины $O(\log|g|)$.

Верхние оценки сложности разрешающих алгоритмов будем вычислять, руководствуясь следующей схемой.

1) Оцениваем максимальное число N вершин, которые могут быть пройдены при обходе размеченной системы переходов для проверки анализируемого свойства преобразователей. Это число явно указано в соответствующих разделах 2-5 данной статьи. Вершинами каждой такой LTS являются различные структуры данных, которые включают в себя состояния q преобразователей и элементы g группы G. Очевидно, что все групповые элементы g, встречающиеся в пройденных вершинах LTS, могут быть заданы двоичными строками размера $N_1 = O(\ell N)$ (в том случае, если S – свободная полугруппа) или $N_1 = O(\log \ell N)$ (в том случае, если S – свободная коммутативная полугруппа).
2) Оцениваем максимальное число N_2 переходов из тех вершин LTS, которые должны быть пройдены при ее обходе. Это число зависит от

структур данных, используемых для представления этих вершин. Если a) такая структура данных включает не более r_1 состояний преобразователей, и b) при обходе LTS может быть пройдено не более r_2 вершин с одним и тем же ансамблем состояний $(q_1, \dots, q_{r_1})$, то $N_2 = r_2\, m^{r_1}$.

3) Оцениваем сложность $\varphi(N_1)$ построения отдельного перехода $u \overset{a}{\Rightarrow} v$. Она зависит от a) структур данных, используемых для представления вершин LTS u и v, и b) размера N_1 двоичного представления групповых элементов g, содержащихся в этих структурах данных.

Проведя вычисление указанных параметров, мы можем получить верхнюю оценку $N_2\varphi(N_1)$ сложности по времени верификации свойств ограниченной недетерминированности и эквивалентности конечных преобразователей над полугруппами.

Утверждение 3. Пусть π' и π'' – пара преобразователей над свободной полугруппой S. Предположим, что из стартовой вершины LTS $\Gamma^0_{\pi',\pi''}$ достижима вершина $v = (q', q'', s)$, где $s \notin S$ и $s^- \notin S$. Тогда $\mathrm{V}^0_{\pi',\pi''} \cap R^0_{\pi',\pi''} \neq \emptyset$.

Доказательство. Нетрудно заметить, что если G – свободная группа и $s \notin S$ и $s^- \notin S$, то для любой пары слов h, g из множества A^* верно $h^- sg \neq e$. Поэтому любая вершина $(p', p'', \hat{s})$, достижимая из вершины v, является опровергающей, если хотя бы одно из состояний p', p'' является финальным.

QED

Утверждение 4. Пусть π' и π'' – пара преобразователей над свободной полугруппой S. Предположим, что из стартовой вершины LTS $\Gamma^0_{\pi',\pi''}$ достижима вершина (q', q'', s), где $s \in S$ или $s^- \in S$, и длина выражения s превосходит величину ℓn. Тогда $\mathrm{V}^0_{\pi',\pi''} \cap R^0_{\pi',\pi''} \neq \emptyset$.

Доказательство. Рассмотрим лишь случай $s \in S$. Пусть w – кратчайшее слово, для которого преобразователь π' имеет финальное вычисление $q' \xrightarrow{w/s'} p'$. Очевидно, что длина выражения s не превосходит величины ℓn. Если преобразователь π'' также имеет финальное вычисление $q'' \xrightarrow{w/s''} p''$, то в LTS $\Gamma^0_{\pi',\pi''}$ из вершины (q', q'', s) достижима вершина $u = (q', q'', (s')^- s s'')$. Нетрудно видеть, что $ss'' \in S$, и при этом длина выражения ss'' превосходит величины ℓn. Тогда, как следует из свойств свободной группы G, справедливо соотношение $(s')^- s s'' \neq$ e. Значит, $u \in \mathrm{V}^0_{\pi',\pi''} \cap R^0_{\pi',\pi''}$.

QED

Из этих утверждений и теоремы 1 вытекает

Теорема 6. Проблема эквивалентности детерминированных преобразователей над свободной полугруппой разрешима за время $O(\ell n^3)$.

Доказательство. Из утверждения 3 и 4 следует, что $N_1 = O(\ell n)$. Для детерминированных преобразователей верно $m = O(n)$. Потому $N_2 = O(n^2)$. Каждый переход в LTS $\Gamma^0_{\pi,\pi'}$ можно построить за время $\varphi(N_1) = 2\log n + N_1 = O(n)$. Таким образом, описанная выше схема вычислений дает указанную в теореме оценку сложности. QED

Аналогичным образом может быть доказана

Теорема 7. Проблема эквивалентности детерминированных преобразователей над свободной коммутативной полугруппой разрешима за время $O(n^2\ log\ \ell n)$.

Теорема 8. Свойство функциональности конечных преобразователей над свободной полугруппой можно проверить за время $O(\ell m^2 n^2)$.

Доказательство. Как следует из лемм 3 и 4, для проверки свойства функциональности (однозначности) недетерминированного преобразователя нужно рассмотреть не более $n^2 + 1$ вершин $v = (q_1, q_1', g)$ в LTS $\Gamma^1_{\pi,\pi}$. Поэтому $N = n^2 + 1$ и $N_1 = O(\ell n^2)$. Каждый переход $(q_1, q_1', g_1) \overset{a}{\Rightarrow} (q_1, q_2', g_2)$ в LTS $\Gamma^1_{\pi,\pi}$ можно построить за время $\varphi(N_1) = \log n + N_1 + \ell = O(n^2)$. Нужно отметить, что при построении этих переходов нет необходимости всякий раз проверять условие $L(A_\pi[q_2]) \cap L(A_{\pi'}[q_2']) \neq \emptyset$: верификация проводится «налету» по ходу поиска опровергающей вершины. Таким образом, получается указанная в теореме верхняя оценка сложности $O(\ell\ n^2 m^2)$. QED

Теорема 9. Свойство функциональности конечных преобразователей над свободной коммутативной полугруппой можно проверить за время $O(m^2 \log \ell n)$.

Теорема 10. Свойство k-значности конечных преобразователей над свободной полугруппой можно проверить за время $O((k+1)^{2(k+1)^2}\ell\ m^{k+1} n^{k+1})$.

Доказательство. Хотя в теореме 4 был описан только метод проверки 2-значности конечных преобразователей, его можно применить и для проверки свойства k-значности преобразователей при любом k. Вершинами LTS Γ^2_π являются наборы $v = (q_0, (q_1, h_1), \dots, (q_k, h_k))$, и задача проверки k-значности преобразователей сводится к проверке достижимости опровергающих вершин в LTS Γ^2_π. Для этого достаточно проверить не более $\binom{k+1}{2}^{\binom{k+1}{2}} |Q|^{k+1} + 1$ значимых вершин. Значит, $\mathrm{N}_1 = \ell \binom{k+1}{2}^{\binom{k+1}{2}} n^{k+1}$, $\mathrm{N}_2 = \binom{k+1}{2}^{\binom{k+1}{2}} \mathrm{m}^{\mathrm{k+1}}$, $\varphi(N_1) = O(kN_1)$. Таким образом, мы приходим к указанной и теореме оценке сложности. QED

Оценка сложности в теореме 10 показывает, что предложенный нами алгоритм существенно эффективнее процедуры проверки свойства k-значности конечных преобразователей, описанной в статье [18], которая решает эту задачу за время $O(2^{(k+1)^4}\ell\ m^{k+1} n^{k+1})$.

Теорема 11. Проверку эквивалентности функциональных преобразователей над свободной полугруппой можно провести за время $\ell 2^{O(n)}$.

Доказательство. Отличительная особенность проверки эквивалентности функциональных преобразователей π и π' состоит в том, что вначале нужно проверить равенство $L(A_\pi[q_0]) = L(A_{\pi'}[q_0'])$. Этот предварительный анализ можно провести за время $2^{O(n)}$. Все остальные операции в алгоритме проверки эквивалентности, описанном в теореме 3, можно выполнить за время $O(\ell\, m^2\, n^2)$. QED

Теорема 12. Проблема эквивалентности для 2-значных конечных преобразователей над свободными полугруппами может быть решена за время $\ell\, 2^{O(n \log n)}$.

Доказательство. 1. По теореме 5 для проверки эквивалентности $\pi \sim \pi'$ нужно проверить не более $N = 2^{O(n)}$ вершин в LTS $\Gamma^3_{\pi,\pi'}$. Поэтому $N_1 = \ell\, 2^{O(n)}$.

2. Рассмотрим произвольную тройку вершин v_1, v_2, v_3, являющихся последователями вершины u в LTS $\Gamma^3_{\pi,\pi'}$. Поскольку π и π' являются 2-значными преобразователями, из определения $\Gamma^3_{\pi,\pi'}$ следует, что вершины v_1, v_2, v_3 не могут иметь один и тот же тип. Но поскольку в LTS $\Gamma^3_{\pi,\pi'}$ существует не более $n3^n$ различных типов вершин, приходим к выводу о том, что $N_2 = 2^{O(n)}$.

3. Выделить все возможные блоки состояний можно за время $2^n n^n = 2^{O(n \log n)}$. Как только это сделано, каждый переход в LTS $\Gamma^3_{\pi,\pi'}$ можно построить за время $\varphi(N_1) = O(mN_1)$.

Так мы приходим к оценке сложности, объявленной в теореме. QED

Полученная оценка сложности показывает, что предложенный нами алгоритм существенно эффективнее процедуры проверки эквивалентности 2-значных конечных преобразователей, описанной в статье [20], которая решает эту задачу за время $O(2^{n^6})$.

8. Заключение

В данной статье предложен универсальный подход к решению некоторых задач анализа поведения конечных преобразователей над полугруппами; преобразователи такого вида можно рассматривать как простую модель последовательных реагирующих программ. Предложенный метод основывается на сведении рассматриваемых проблем – задачи проверки k-значности и задачи проверки эквивалентности преобразователей – к хорошо известной задача проверки достижимости заданного класса вершин в конечном ориентированном графе. Фактически, сложность алгоритмов, которые могут быть построены при помощи нашего метода, определяется размером того графа (размеченной системы переходов), который сопоставляется анализируемым преобразователям. Построенные нами алгоритмы имеют более простое устройство и меньшую вычислительную сложность, чем ранее известные алгоритмы решения тех же самых задач.

Есть основания полагать, что полученные в данной статье результаты можно применить и для решения задач минимизации конечных преобразователей над полугруппами и обобщить тем самым ранее известные результаты исследования этой задачи, представленные в статье [15]. Также представляет интерес задача детерминизации конечных преобразователей над полугруппами. В этом случае использование алгебраических особенностей полугрупп, над которыми определяются вычисления преобразователей, может существенно расширить класс однозначных преобразователей, допускающих детерминизацию.

Наконец, как видно из приведенных оценок сложности решения задач проверки свойства -значности преобразователей, предложенный нами метод позволяет решать эту задачу за время, полиномиально зависящее относительно размера проверяемого преобразователя, но при этом экспоненциально зависящее от параметра k. Такой же эффект наблюдается и для разрешающих алгоритмов, описанных в работе [17]. Поэтому при больших значениях параметра k все известные алгоритмы практически непригодны для проверки свойства -значности преобразователей. Поэтому желательно выяснить, какова нижняя оценка того вклада, который привносит параметр k в сложность решения указанной задачи.

Список литературы

[1]. Aho A.V., Hopcroft J.E., Ullman J.D. The design and analysis of computer algorithms. Addison-Wesley, Reading, MA, 1974.

[2]. Aho A.V., Sethi R., Ullman J.D. Compilers: principles, techniques, and tools. Addison-Wesley, Reading, MA, 1986.

[3]. Alur R., Cerny P. Streaming transducers for algorithmic verification of single-pass list-processing programs. Proceedings of 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 2011, p. 599-610.

[4]. Blattner M, Head T. Single-valued a-transducers. Journal of Computer and System Sciences, 1977, v. 15, p. 310-327.

[5]. Blattner M, Head T. The decidability of equivalence for deterministic finite transducers. Journal of Computer and System Sciences, 1979, v. 19, p. 45-49.

[6]. Beal M.-P., Carton O., Prieur C., Sakarovitch J. Squaring transducers: an efficient procedure for deciding functionality and sequentiality. Theoretical Computer Science, 2003, v. 292.

[7]. Culik K., Karhumaki J. The equivalence of finite-valued transducers (on HDTOL languages) is decidable. Theoretical Computer Science, 1986, v. 47, p. 71-84.

[8]. Fischer P.C., Rosenberg A.L. Multi-tape one-way nonwriting automata. Journal of Computer and System Sciences, 1968, v. 2, p. 88-101.

[9]. Griffiths T. The unsolvability of the equivalence problem for ε-free nondeterministic generalized machines. Journal of the ACM, 1968, v. 15, p.409-413.

[10]. Gurari, E., Ibarra, O. A note on finite-valued and finitely ambiguous transducers. Mathematical Systems Theory, 1983, v. 16, p. 61–66.

[11]. Ibarra O. The unsolvability of the equivalence problem for Efree NGSM's with unary input (output) alphabet and applications. SIAM Journal on Computing, 1978, v. 4.

[12]. Malcev, A. I. Uber die Einbettung von assoziativen Systemen. Gruppen, Rec. Math. (Mat. Sbornik) N.S., 1939, v. 6, p. 331–336.
[13]. Malcev, A. I. Uber die Einbettung von assoziativen Systemen. Gruppen. II, Rec. Math. (Mat. Sbornik) N.S., 1940, v. 8, p. 251–264.
[14]. Mohri M. Finite state transducers in language and speech processing. Computer Linguistics, 1997, v. 23, N 2.
[15]. Mohri M. Minimization algorithms for sequential transducers. Theoretical Computer Science, 2000, v. 234, p. 177-201.
[16]. Nerode A., Kohn W. Models for hybrid systems: automata, topology, controllability, observability. Cornell University, Technical Report 93-28, 1993, MIT Press, Cambridge.
[17]. Sakarovitch J., de Souza R. On the decomposition of k-valued rational relations. Proceedings of 25th International Symposium on Theoretical Aspects of Computer Science, 2008, p.621-632.
[18]. Sakarovitch J., de Souza R. On the decidability of bounded valuedness for transducers. Proceedings of the 33rd International Symposium on Mathematical Foundations of Computer Science, 2008, p. 588-600.
[19]. Schutzenberger M. P. Sur les relations rationnelles. Proceedings of Conference on Automata Theory and Formal Languages, 1975, p. 209-213.
[20]. de Souza R. On the decidability of the equivalence for k-valued transducers. Proceedings of 12th International Conference on Developments in Language Theory, 2008, p. 252-263.
[21]. Veanes M., Hooimeijer P., Livshits B., et al. Symbolic finite state transducers: algorithms and applications. Proceedings of the 39th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 2012.
[22]. Weber A. On the valuedness of finite transducers. Acta Informatica, 1989, v. 27, p. 749–780.
[23]. Weber A. Decomposing finite-valued transducers and deciding their equivalence. SIAM Journal on Computing, 1993, v. 22, p. 175-202.
[24]. Zakharov V.A. An efficient and unified approach to the decidability of equivalence of propositional program schemes. Proceedings of the 25th International Colloquium "Automata, Languages and Programming", 1998, p. 247-258.

Modeling and Analysis of the Behavior of Successive Reactive Programs

V.A. Zakharov <zakh@cs.msu.su>
Institute for System Programming Russian Academy of Science,
109004, A. Solzhenitsina, 25, Moscow, Russia;
Higher School of Economics National Research University,
101000, Myasnitskaya, 20, Moscow, Russia

Annotation. Finite state transducers extend the finite state automata to model functions on strings or lists. They may be used also as simple models of sequential reactive programs. These programs operate in the interaction with the environment permanently receiving data (requests) from it. At receiving a piece of data such program performs a sequence of actions.

When certain control points are achieved a program outputs the current results of computation as a response. It is significant that different sequences of actions may yield the same result. Therefore, the basic actions of a program may be viewed as generating elements of some appropriate semigroup, and the result of computation may be regarded as the composition of actions performed by the program. This paper offers an alternative technique for the analysis of finite state transducers over semigroups. To check the equivalence of transducers $\boldsymbol{\pi_1}$ and $\boldsymbol{\pi_2}$ we associate with them a Labeled Transition Systems $\boldsymbol{\Gamma_{\pi_1,\pi_2}}$. Each path in this LTS represents all possible runs of $\boldsymbol{\pi_1}$ and $\boldsymbol{\pi_2}$ on the same input word. Every node of $\boldsymbol{\Gamma_{\pi_1,\pi_2}}$ keeps track of the states of $\boldsymbol{\pi_1}$ and $\boldsymbol{\pi_2}$ achieved at reading some input word and the deficiency of the output words computed so far. If both transducers reach their final states and the deficiency of their outputs is nonzero then this indicates that $\boldsymbol{\pi_1}$ and $\boldsymbol{\pi_2}$ produce different images for the same word, and, hence, they are not equivalent. The nodes of $\boldsymbol{\Gamma_{\pi_1,\pi_2}}$ that capture this effect are called rejecting nodes. Thus, the equivalence checking of $\boldsymbol{\pi_1}$ and $\boldsymbol{\pi_2}$ is reduced to checking the reachability of rejecting nodes in LTS $\boldsymbol{\Gamma_{\pi_1,\pi_2}}$. We show that one needs to analyze only a bounded fragment of $\boldsymbol{\Gamma_{\pi_1,\pi_2}}$ to certify (un)reachability of rejecting nodes. The size of this fragment is polynomial of the size of $\boldsymbol{\pi_1}$ and $\boldsymbol{\pi_2}$ if both transducers are deterministic, and single-exponential if they are k-bounded. The same approach is applicable for checking k-valuedness of transducers over semigroups.

Keywords: reactive program; finite state transducer; semigroup; Labelled Transition System; equivalence checking; k-valuedness; decision procedure; complexity.

DOI: 10.15514/ISPRAS-2015-27(2)-13

For citation: Zakharov V.A. Modeling and Analysis of the Behavior of Successive Reactive Programs. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 221-250 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-13.

References

[1]. Aho A.V., Hopcroft J.E., Ullman J.D. The design and analysis of computer algorithms. Addison-Wesley, Reading, MA, 1974.

[2]. Aho A.V., Sethi R., Ullman J.D. Compilers: principles, techniques, and tools. Addison-Wesley, Reading, MA, 1986.

[3]. Alur R., Cerny P. Streaming transducers for algorithmic verification of single-pass list-processing programs. Proceedings of 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 2011, p. 599-610.

[4]. Blattner M, Head T. Single-valued a-transducers. Journal of Computer and System Sciences, 1977, v. 15, p. 310-327.

[5]. Blattner M, Head T. The decidability of equivalence for deterministic finite transducers. Journal of Computer and System Sciences, 1979, v. 19, p. 45-49.

[6]. Beal M.-P., Carton O., Prieur C., Sakarovitch J. Squaring transducers: an efficient procedure for deciding functionality and sequentiality. Theoretical Computer Science, 2003, v. 292.

[7]. Culik K., Karhumaki J. The equivalence of finite-valued transducers (on HDTOL languages) is decidable. Theoretical Computer Science, 1986, v. 47, p. 71-84.

[8]. Fischer P.C., Rosenberg A.L. Multi-tape one-way nonwriting automata. Journal of Computer and System Sciences, 1968, v. 2, p. 88-101.

[9]. Griffiths T. The unsolvability of the equivalence problem for ε-free nondeterministic generalized machines. Journal of the ACM, 1968, v. 15, p.409-413.
[10]. Gurari, E., Ibarra, O. A note on finite-valued and finitely ambiguous transducers. Mathematical Systems Theory, 1983, v. 16, p. 61–66.
[11]. Ibarra O. The unsolvability of the equivalence problem for Efree NGSM's with unary input (output) alphabet and applications. SIAM Journal on Computing, 1978, v. 4.
[12]. Malcev, A. I. Uber die Einbettung von assoziativen Systemen. Gruppen, Rec. Math. (Mat. Sbornik) N.S., 1939, v. 6, p. 331–336.
[13]. Malcev, A. I. Uber die Einbettung von assoziativen Systemen. Gruppen. II, Rec. Math. (Mat. Sbornik) N.S., 1940, v. 8, p. 251–264.
[14]. Mohri M. Finite state transducers in language and speech processing. Computer Linguistics, 1997, v. 23, N 2.
[15]. Mohri M. Minimization algorithms for sequential transducers. Theoretical Computer Science, 2000, v. 234, p. 177-201.
[16]. Nerode A., Kohn W. Models for hybrid systems: automata, topology, controllability, observability. Cornell University, Technical Report 93-28, 1993, MIT Press, Cambridge.
[17]. Sakarovitch J., de Souza R. On the decomposition of k-valued rational relations. Proceedings of 25th International Symposium on Theoretical Aspects of Computer Science, 2008, p.621-632.
[18]. Sakarovitch J., de Souza R. On the decidability of bounded valuedness for transducers. Proceedings of the 33rd International Symposium on Mathematical Foundations of Computer Science, 2008, p. 588-600.
[19]. Schutzenberger M. P. Sur les relations rationnelles. Proceedings of Conference on Automata Theory and Formal Languages, 1975, p. 209-213.
[20]. de Souza R. On the decidability of the equivalence for k-valued transducers. Proceedings of 12th International Conference on Developments in Language Theory, 2008, p. 252-263.
[21]. Veanes M., Hooimeijer P., Livshits B., et al. Symbolic finite state transducers: algorithms and applications. Proceedings of the 39th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 2012.
[22]. Weber A. On the valuedness of finite transducers. Acta Informatica, 1989, v. 27, p. 749–780.
[23]. Weber A. Decomposing finite-valued transducers and deciding their equivalence. SIAM Journal on Computing, 1993, v. 22, p. 175-202.
[24]. Zakharov V.A. An efficient and unified approach to the decidability of equivalence of propositional program schemes. Proceedings of the 25th International Colloquium "Automata, Languages and Programming", 1998, p. 247-258.