УДК 681.518.5

А.С. Базин

АВТОМАТИЗИРОВАННОЕ ТЕСТИРОВАНИЕ ПРОГРАММНЫХ КОМПЛЕКСОВ

Нижегородский государственный технический университет им. Р.Е. Алексеева

Разработан автоматизированный алгоритм тестирования программных комплексов, позволяющий повысить эффективность тестирования за счет использования принципа декомпозиции.

Ключевые слова: программный комплекс, тестирование, декомпозиция, алгоритм, граф.

Введение

Повышение надежности функционирования и эффективности тестирования программных комплексов, сложность которых непрерывно возрастает, является актуальной задачей, решение которой начинается уже на этапе их проектирования.

Известные методы тестирования, методы обеспечения качества программных комплексов носят, как правило, частный характер, т.е. разрабатываются заново для каждого комплекса. Не решена в общем виде задача повышения надежности и контролепригодности программных комплексов.

При тестировании программных комплексов методами тестирования технических средств возникают принципиальные трудности, связанные с основными особенностями программного комплекса: более сложными интерфейсами и трудностями формализации описаний.

В первую очередь сложность комплекса определяется сложностью решаемой проблемы и зависит от сложности отдельных компонент комплекса и связей между ними. Для повышения надежности и контролепригодности программного комплекса стремятся снизить зависимость между компонентами комплекса, т.е. стремятся разбить комплекс на такие модули, между которыми должно остаться по возможности меньшее количество связей [1].

Использование иерархических структур позволяет стратифицировать связи между компонентами комплекса.

В качестве модели программного комплекса предлагается использовать управляющий граф, вершинами которого являются процедуры, которые реализуют функции комплекса. Данный граф разбивается на подграфы. Подмножества вершин, входящих в образованные подграфы, соответствуют программным модулям комплекса. Далее программные модули тестируются отдельно и параллельно на основании использования их внутренних структур. Такой подход позволяет проводить тестирование программного комплекса за меньший промежуток времени, и при этом тестовое покрытие комплекса увеличивается.

В основе алгоритма разбиения комплекса на модули лежит количественная характеристика программных процедур комплекса: «популярность» (активность) процедур в комплексе, т.е. частота вызова той или иной процедуры другими процедурами комплекса.

Процедуры, выполняющие подзадачи, характерные для определенной функциональности в комплексе, могут быть вызваны только строго определенной процедурой, в этом случае вызываемая и вызывающая процедуры должны быть выделены в один и тот же модуль комплекса. В отличие от этого, процедуры, использующиеся для хранения или доступа к значениям в структурах данных, таких как списки или хэш-таблицы, вероятно, будут вызваны несколькими различными процедурами и такие «сервисные» процедуры должны быть выделены в отдельные модули.

В работе предлагается следующий подход к решению проблемы тестирования программного комплекса:

- осуществляем статический анализ исходного кода комплекса для выявления его структуры;

- разбиваем комплекс на модули согласно алгоритму, описанному ранее;
- тестируем выделенные модули с применением автоматизированной технологии генерации тестов;
- тестируем связи между выделенными модулями с применением автоматизированной технологии генерации тестов;
- анализируем результаты тестирования.

С помощью автоматизированной технологии генерации тестов можно обнаружить стандартные ошибки, такие как неожиданное завершение программы (crash), утверждения о нарушениях логики работы программы (assertion violation) и незавершение программы. Во время тестирования осуществляется направленный поиск (разновидность динамического формирования тестов) [2-3]. Начиная со случайного входного вектора, во время исполнения текущего теста вычисляется очередной вектор для следующего исполнения. Этот вектор содержит значения, которые являются решением символических связей, данное значение состоит из предикатов операторов ветвления (операторов перехода), накопленных в течение предыдущего исполнения. Новый входной вектор указывает программе новый путь выполнения. Повторяя этот процесс, направленный поиск стремится охватить все возможные пути исполнения программы.

Для автоматизированной генерации тестов предлагается использовать утилиту CUTE [4]. Данная утилита реализует символьное выполнение программ.

Рассмотрим предлагаемый подход на примере.

Построение управляющего графа программного комплекса

1. Построение управляющего графа для конкретного программного комплекса проводится по простым правилам с требуемой степенью детальности (и, как правило, автоматически).

В работе предлагается использовать управляющий граф программного комплекса, детализированный до пользовательских процедур. При таком подходе в вершинах управляющего графа будут располагаться пользовательские процедуры, операторы и стандартные процедуры языка программирования не будут образовывать вершины в графе.

Для примера рассмотрим программу, содержащую следующие пользовательские процедуры:

1) *main*
2) *chooseBehaviour*
3) *waitForMessage*
4) *sendTraffic*
5) *handleReceivedMessage*
6) *handleMgmtMessage*
7) *handleDataMessage*
8) *handleRoutingInfoMessage*
9) *handleSystemReq*
10) *sendAck*
11) *send2Network*
12) *sendData*
13) *sendMgmtMessage*

Передача управления в данной программе между процедурами происходит следующим образом:

1) *"main" -> "chooseBehaviour";*
2) *"chooseBehaviour" -> "waitForMessage";*
3) *"chooseBehaviour" -> "sendTraffic";*
4) *"waitForMessage" -> "handleReceivedMessage";*
5) *"handleReceivedMessage" -> "handleMgmtMessage";*
6) *"handleReceivedMessage" -> "handleDataMessage";*

7) "handleMgmtMessage" -> "handleRoutingInfoMessage";
8) "handleMgmtMessage" -> "handleSystemReq";
9) "handleDataMessage" -> "sendAck";
10) "handleRoutingInfoMessage" -> "sendAck";
11) "handleSystemReq" -> "chooseBehaviour";
12) "sendAck" -> "send2Network";
13) "sendTraffic" -> "sendData";
14) "sendTraffic" -> "sendMgmtMessage";
15) "sendData" -> "send2Network";
16) "sendMgmtMessage" -> "send2Network";
17) "send2Network" -> "chooseBehaviour";

Управляющий граф для данной программы с учетом предложенного подхода выглядит следующим образом (рис. 1).

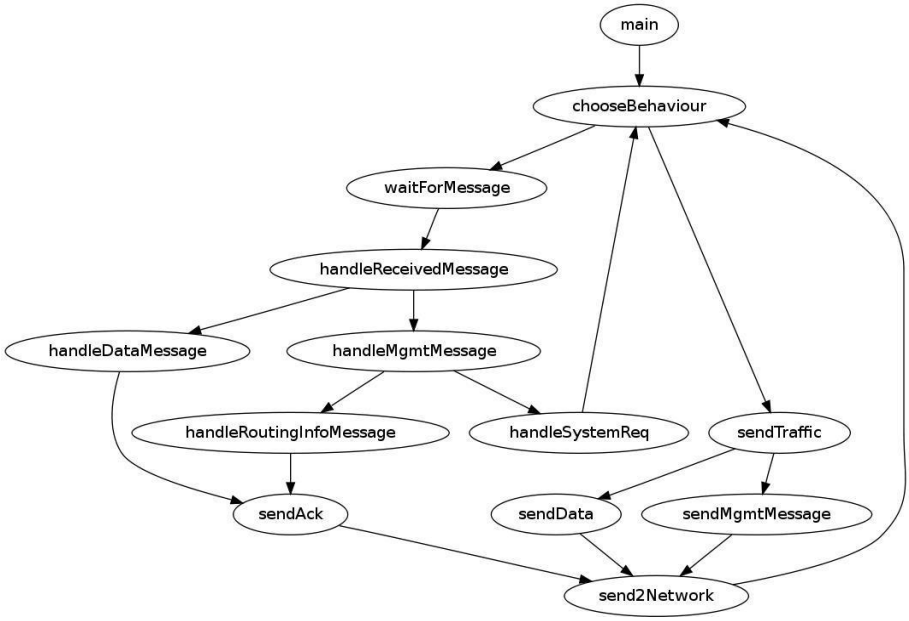

Рис. 1. Управляющий граф программы

Построение управляющего графа для конкретного программного комплекса может быть выполнено автоматически. Автор исследовал программные комплексы, реализованные на языке программирования С, опишем, как происходит автоматическое моделирование объекта для данного языка программирования.

Для выполнения данной задачи необходимо установить компиляторы GCC и Perl и выполнить следующие действия:

1) компилировать исходный код программного комплекса компилятором GCC с опцией “-dr”, результатом данной операции будут дамп-файлы с расширением RTL (Register Transfer Language);

2) запустить скрипт, написанный на языке программирования Perl, который извлекает всю информацию о вызовах процедур из файлов RTL.

Рассмотрим работу предложенного алгоритма на следующем примере:

1) программа состоит из трех файлов: main.c, client.c, server.c.

Компилировать исходный код программы компилятором GCC с опцией “-dr”:

 # gcc –dr main.c client.c server.c

В результате данной операции создаются следующие файлы:

main.c.01.rtl, client.c.01.rtl, server.c.01.rtl;

2) запускаем Perl скрипт:

 # rtl2callgraph main.c.01.rtl client.c.01.rtl server.c.01.rtl

Результатом этой операции будет диаграмма вызовов процедур в программе:

```
digraph callgraph {
"main" -> "chooseBehaviour" [style=solid];
"chooseBehaviour" -> "waitForMessage" [style=solid];
"chooseBehaviour" -> "sendTraffic" [style=solid];
"waitForMessage" -> "handleReceivedMessage" [style=solid];
"handleReceivedMessage" -> "handleMgmtMessage" [style=solid];
"handleReceivedMessage" -> "handleDataMessage" [style=solid];
"handleMgmtMessage" -> "handleRoutingInfoMessage" [style=solid];
"handleMgmtMessage" -> "handleSystemReq" [style=solid];
"handleDataMessage" -> "sendAck" [style=solid];
"handleRoutingInfoMessage" -> "sendAck" [style=solid];
"handleSystemReq" -> "chooseBehaviour" [style=solid];
"sendAck" -> "send2Network" [style=solid];
"sendTraffic" -> "sendData" [style=solid];
"sendTraffic" -> "sendMgmtMessage" [style=solid];
"sendData" -> "send2Network" [style=solid];
"sendMgmtMessage" -> "send2Network" [style=solid];
"send2Network" -> "chooseBehaviour" [style=solid];
}
```

Полученную информацию о структуре программы можно использовать для построения управляющего графа.

Алгоритм разбиения комплекса на модули

Предлагаемый алгоритм генерирует распределение модулей комплекса, в которых процедуры f и g с большей вероятностью будут отнесены к одному и тому же модулю, если f вызывает g и процедура g не очень популярна.

Алгоритм можно описать следующим образом:

1. Строим граф «популярности» процедур комплекса, взвешенный направленный граф $G = (V, E)$ с множеством вершин $V = S$, где S – множество процедур комплекса и направленные ребра множества E обозначают вызовы между процедурами в комплексе. Вес каждого ребра равен «популярности» вызываемой процедуры (т.е. «популярности» процедуры, соответствующей вершине назначения ребра).

2. Выбираем значение c, равное средней «популярности» процедур в комплексе. На первой итерации алгоритма значение c определяется пользователем на основе эмпирических соображений (результаты вычислительных экспериментов показывают, что при $c=3$ на первой итерации алгоритма получаем оптимальное разбиение комплекса на модули), а при следующих итерациях значение c выбирается равным максимальному весу ребра в графе G.

3. Временно удаляем вершины в графе «популярности» процедур, для которых «популярность» соответствующей процедуры выше выбранной средней «популярности» процедур c, образуя граф G'.

4. В полученном графе G' находим вершины, для которых полустепень захода равна нулю.

5. Вычисляем множество достижимых вершин в G' из найденных вершин с нулевой полустепенью захода, если достижима хотя бы одна вершина, полученный набор вершин определяется как единое целое, модуль. Вершины, которые выделяются в модули, удаляются из графа G и из подграфа G'. Если не достижима ни одна вершина из найденной вершины, найденная вершина удаляется из графа G'.

Шаги со 2-го по 5-й повторяются до тех пор, пока все процедуры не будут выделены в модули. В результате мы получаем множество модулей U.

Рассмотрим работу данного алгоритма на примере.

Выделим модули в программе, управляющий граф которой изображен на рис. 1.

1. Построим граф популярности процедур. Затем выбираем значение средней «популярности» процедур $c = 2$ и временно удаляем вершины в графе, для которых «популярность» соответствующей процедуры выше выбранной средней «популярности» процедур $c=2$. Удаляем вершины *chooseBehaviour* и *send2Network*, так как «популярность» соответствующих процедур равна трем. Получаем граф G' (рис. 2).

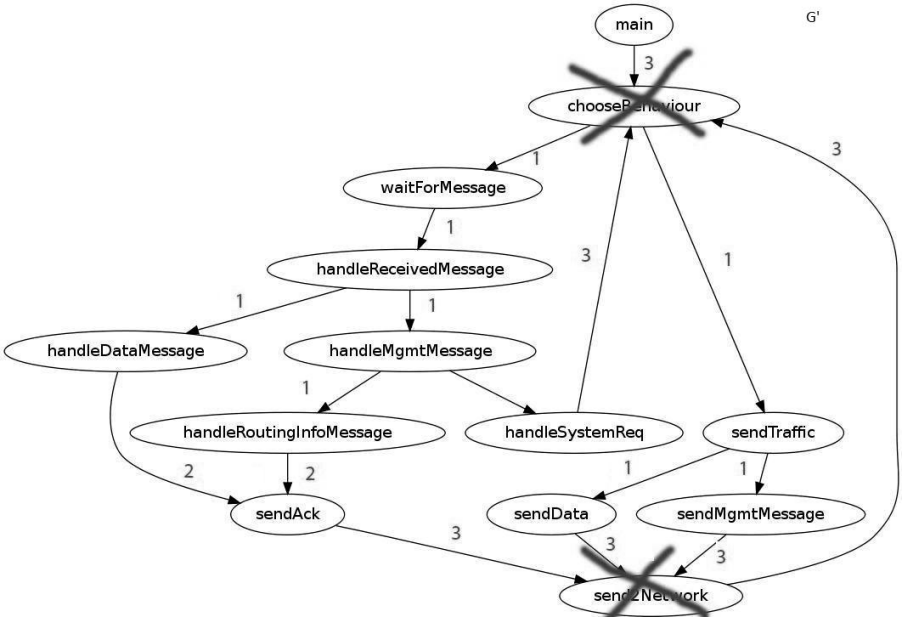

Рис. 2. Алгоритм разбиения комплекса на модули

2. В полученном графе G' находим вершины, для которых полустепень захода равна нулю, такими вершинами являются *main*, *waitForMessage* и *sendTraffic*. Вычисляем множество достижимых вершин в G' из найденных вершин. Из вершины *main* не достижима ни одна вершина графа G', удаляем эту вершину из графа G'. Из вершины *waitForMessage* достижимы вершины*: handleReceivedMessage, handleMgmtMessage, handleDataMessage, handleRoutingInfoMessage, handleSystemReq, sendAck* – в таком случае объединяем эти вершины в модуль и удаляем их из графов G и G'. Из вершины *sendTraffic* достижимы вершины *sendMgmtMessage* и *sendData* –объединяем эти вершины в новый модуль и удаляем их из графов G и G'. Граф G' оказался пустым (рис. 3).

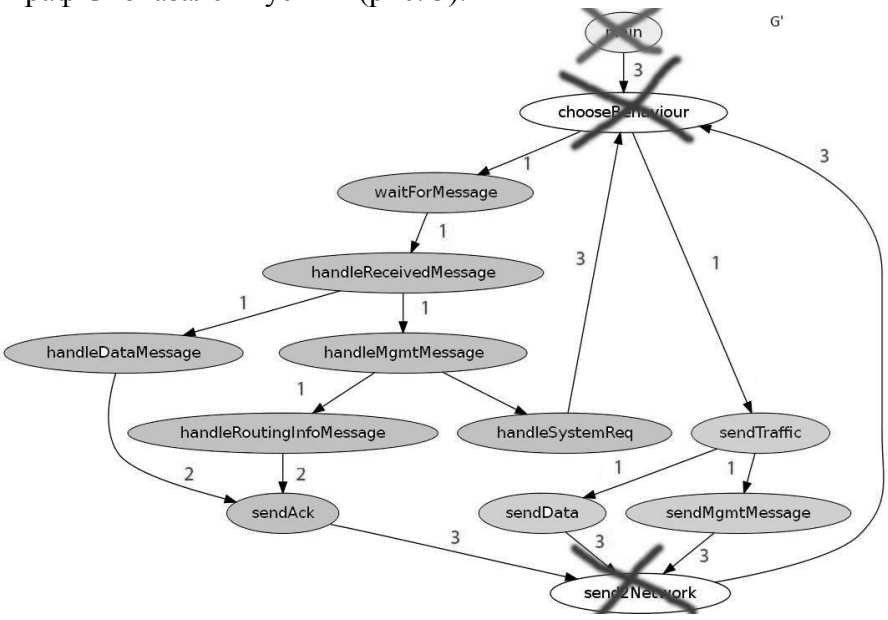

Рис. 3. Алгоритм разбиения комплекса на модули

3. В результате предыдущих шагов в графе *G* осталось всего три вершины: *main*, *chooseBehaviour*, *send2Network*. В этом графе находим вершины, для которых полустепень захода равна нулю, такими вершинами являются *main* и *send2Network*. Вычисляем множество достижимых вершин в графе *G* из найденных вершин. Из вершины *main* достижима вершина *chooseBehaviour*, объединяем эти вершины в модуль и удаляем из графа *G*. Осталась единственная нерассмотренная вершина *send2Network*, из нее не достижима ни одна вершина графа *G*, но, тем не менее, мы выделяем ее в отдельный модуль (рис. 4).

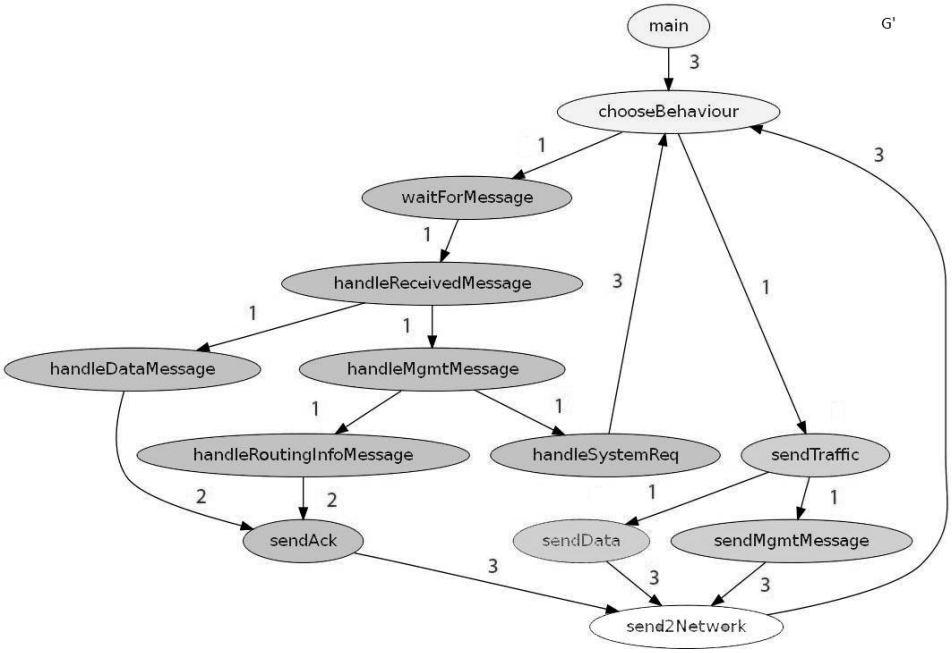

Рис. 4. Алгоритм разбиения комплекса на модули

Таким образом, программа была разбита на четыремодуля, содержащие следующие функции:

1) *waitForMessage, handleReceivedMessage, handleMgmtMessage, handleDataMessage, handleRoutingInfoMessage, handleSystemReq, sendAck;*
2) *sendTraffic, sendData, sendMgmtMessage;*
3) *main, chooseBehaviour;*
4) *send2Network.*

Алгоритм выделил клиентскую и серверную функции программы абсолютно верно (первый и второй модули).

Автоматизированная технология генерации тестов

Для автоматизированной генерации тестов в данной работе используется утилита CUTE (Concolic Unit Testing Engine for C and Java). Утилита CUTE является инструментом для систематического и автоматического тестирования программ, написанных на языках программирования C и Java. Данная утилита реализует символьное выполнение программ. Также CUTE позволяет обозревать все возможные пути исполнения программы и вести статистику тестового покрытия кода.

Команда запуска CUTE выглядит следующим образом:

#cute название_программы процедура_входа;

название_программы – название тестируемой программы;

процедура_входа – название входной процедуры, с которой начнется процесс тестирования.

Остановимся подробнее на методе написания тестовых драйверов программы.

Допустим, что мы хотим протестировать процедуру testme в небольшой программе (cell.c), написанной далее:

```
#include <stdio.h>:
#include <assert.h>

typedef struct cell {
   int v;
   struct cell *next;
} cell;

int g(int v) {
   return 2*v + 1;
}

int testme(cell *p, int x) {
   if (x > 0)
      if (p != NULL)
         if (g(x) == p->v)
               if (p->next == p)
                  assert(0);
   return 0;
}
```

Тест-драйвер объявляет две переменные: *p* и *x*. Затем указывается, что данные переменные будут браться из тестового окружения (CUTE_input) и процедура *testme* выполняется с переменными *p* и *x*.

```
f(){
   cell *p;
   int x;
   CUTE_input(p);
   CUTE_input(x);
   testme(p,x);
}
```

Теперь процедура *testme* может быть протестирована, но для начала компилируем данную программу с помощью утилиты *cutec:*

#cutec cell.c f

Данная команда показывает, что процедура *f* – процедура, с которой начнется процесс тестирования программы. Процесс компиляции генерирует два исполняемых файла: *cell.exe* и *cell.g.exe*. Программа *cell.exe* выполняется несколько раз, она генерирует новые входные данные для тестируемой программы. Сгенерированные входные данные сохраняется, чтобы помочь процессу отладки программы в случае ошибки.

После компиляции программы запускаем процесс тестирования:

#cute cell –i 1000

Данная команда указывает, что должно быть сгенерировано не менее 1000 различных входных данных для тестируемой программы.

Вычислительный эксперимент

Для демонстрации эффективности предложенного автором подхода был выбран программный комплекс Open SIP, разработка протокола установления сеансов (Session Initiation Protocol) с открытым исходным кодом. Протокол SIP описывает, каким образом клиентское приложение может запросить начало соединения у другого, возможно, физически удалённого клиента, находящегося в той же сети, используя его уникальное имя. Протокол определяет

способ согласования между клиентами об открытии каналов обмена на основе других протоколов, которые могут использоваться для непосредственной передачи информации. Допускается добавление или удаление таких каналов в течение установленного сеанса, а также подключение и отключение дополнительных клиентов (то есть допускается участие в обмене более двух сторон — конференц-связь). Протокол также определяет порядок завершения сеанса.

Реализация данного комплекса на языке программирования C доступна на сайте http://www.gnu.org/software/osip/osip.html. Для тестирования была использована версия продукта 2.2.1, исходный код комплекса состоит примерно из 30,000 строк программного кода.

Для тестирования был выделен программный код, отвечающий за кодирование и декодирование сообщений SIP протокола, данный фрагмент кода состоит из 70 процедур (~10,500 строк программного кода).

Построив управляющий граф для данного комплекса, можно увидеть, что 25 из 70 процедур мало популярны: вызываются лишь одной или двумя другими процедурами, 22 процедуры вызываются 5-ю другими процедурами, 15 процедур вызываются 10 другими процедурами, и 8 процедур очень популярны, вызываются 20 разными процедурами.

Таблица 1

Результаты эксперимента

Алгоритм разбиения	**Количество выделенных модулей**	**Тестовое покрытие исходного кода комплекса**
Случайное разбиение комплекса на 6 модулей (r6)	6	0,44
Случайное разбиение комплекса на 11 модулей (r11)	11	0,49
Случайное разбиение комплекса на 16 модулей (r16)	16	0,32
Разработанный автором алгоритм	**18**	**0,85**
Случайное разбиение комплекса на 21 модулей (r21)	21	0,5
Случайное разбиение комплекса на 26 модулей (r26)	26	0,49
Алгоритм Fiduccia-Mattheyses (FM)	30	0,75
Случайное разбиение комплекса на 31 модулей (r31)	31	0,52
Алгоритм Kernighan-Lin (KL)	35	0,65
Случайное разбиение комплекса на 36 модулей (r36)	36	0,58
Случайное разбиение комплекса на 41 модулей (r41)	41	0,52
Случайное разбиение комплекса на 46 модулей (r46)	46	0,56
Случайное разбиение комплекса на 51 модулей (r51)	51	0,68
Каждая процедура комплекса выделяется в отдельный модуль (sml)	55	0,63

Алгоритм, предложенный автором, разбил данный комплекс на 18 модулей. Далее тестирование полученных модулей проводилось утилитой CUTE, которая генерировала до

1000 различных входных данных для каждого модуля, в результате чего было обнаружено несколько ошибок в реализации протокола SIP, которые могли повлечь к неожиданному завершению работы комплекса. Покрытие исходного кода комплекса тестами составило 85%.

Для подтверждения эффективности алгоритма, предложено автором, был проведен следующий эксперимент. Описанный в программный комплекс был разделен на модули разными алгоритмами (случайное разделение на *N* частей, алгоритм Kernighan-Lin [5], алгоритм Fiduccia-Mattheyses [6]), и далее полученные модуля тестировались программой CUTE, генерировалось 1000 различных входных данных для каждого модуля. В процессе тестирования велась статистика тестового покрытия исходного кода.

Результаты эксперимента приведены в табл. 1.

Результат эксперимента можно представить в виде графика (рис. 6).

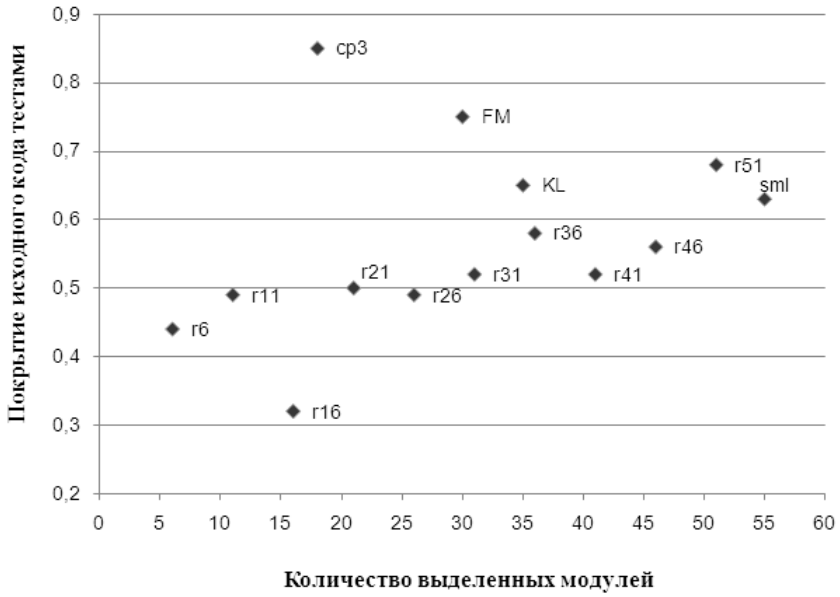

Рис. 5. Тестовое покрытие исходного кода комплекса

Вычислительный эксперимент показал, что наилучшее разбиение достигается при использовании алгоритма, предложенного автором, при использовании данного алгоритма достигается максимальное покрытие кода тестами. Это можно объяснить следующим образом: другие алгоритмы разбиения в большинстве не учитывают направление ребер в управляющем графе комплекса, и в результате этого при разбиении некоторые вершины в модулях становятся не достижимыми ни из одной другой вершины модуля, и, как результат, некоторые процедуры остаются непротестированными. Предложенный автором алгоритм объединяет вершины в модули, которые достижимы из некоторой начальной вершины модуля, тем самым гарантируя, что все вершины в модуле достижимы и все процедуры модуля могут быть протестированы. В связи с этим улучшается качество тестирования программного продукта.

Полученные результаты вычислительного эксперимента подтвердили целесообразность применения декомпозиционного подхода к решению задачи тестирования программных комплексов.

Библиографический список

1. **Липаев, В.В.** Программная инженерия. Методологические основы / В.В. Липаев. – М.: ТЕИС, 2006.
2. Korel B. A dynamic Approach of Test Data Generation // IEEE Conference on Software. IEEE Computer Society Press, 1990. November. Pp. 311-317.

3. Gupta N., Mathur A.P., and Soffa M.L. Generating test data for branch coverage. // Proceedings of the 15th IEEE International Conference on Automated Software Engineering. IEEE Computer Society Press, 2000. September. Pp. 219-227.
4. Sen K., Marinov D., and Agha G. CUTE: A concolic unit testing engine for C. // Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, 2005. Pp. 263-272.
5. Pothen A., Simon H., and Liou K.-P. Partitioning sparse matrices with eigenvectors of graphs. // SIAM Journal of Matrix Analysis and Applications, 1990. № 11. Pp. 430-452.
6. Mattheyses R.M. and Fiduccia C.M. A linear time heuristic for improving network partitions. // In Proceedings 19th IEEE Design Automation Conference, 1982. Pp. 175-181.

Дата поступления
в редакцию 05.10.2012

A.S. Bazin

AUTOMATED TESTING OF SOFTWARE SYSTEMS

Nizhny Novgorod state technical university n.a. R.E. Alexeev

Purpose: A key problem for effective testing of software systems is the difficulty of partitioning large software systems into appropriate units that can be tested in isolation.
Design/methodology/approach: We present an approach that identifies control and data inter-dependencies between software components using static program analysis, and divides the source code into units where highly-intertwined components are grouped together.
Findings: Those units can then be tested in isolation using automated test generation techniques and tools, such as dynamic software model checkers.
Experimental results: We discuss preliminary experimental results showing that automatic software partitioning can significantly increase test coverage.

Key words: bundled software, testing, decomposition, algorithm, graph.