Окончание. Начало в № 9`2014

Применение реинжиниринга при проектировании встраиваемых систем

Анна СЕРГЕЕВА
annserge@rambler.ru

Использование систем перезаписи кода для тестового покрытия

Как уже упоминалось, программные средства преобразования кода, такие как, например, DMS Software Reengineering Toolkit, решают широкий спектр задач, связанных с процессом проведения реинжиниринга программных систем. Среди них, помимо непосредственного перевода исходного кода программ с одних языков на другие, существует еще немалое число возможностей сбора и обработки разного рода аналитических данных.

Одной из интересных аналитических задач, реализуемых с помощью подобных систем преобразования, является оценка и измерение тестового покрытия разработанного программного кода (test coverage). Практика разработчиков различных программ показывает, что применение систем преобразования кода в качестве инструментов автоматизации тестового покрытия достаточно удобно и эффективно. При использовании DMS успешно решаются задачи тестового покрытия кода, написанного для встраиваемых программных систем.

Такие инструментальные средства предоставляют статическую и подробную информацию о том, какие части программного приложения (например, операторы или ветви) были проверены в процессе исполнения предустановленного набора тестов, а какие нет. Эта информация является крайне важной для команды специалистов, поскольку помогает им выполнить оценку качества и готовности разрабатываемого кода для действительного использования их программ на стороне конечных пользователей.

Инструменты тестового покрытия также применимы для локализации функциональных возможностей программных приложений. При таком варианте использования инструмент тестового покрытия находит и указывает ту часть программного кода, которая исполняется для предоставления конкретной выбранной оператором функции приложения. Следует отметить, что это достаточно эффективный способ выяснения конкретных функциональных возможностей в крупномасштабных системах с запутанной структурой.

Особенности тестового покрытия для промышленных систем

При проведении анализа качества разрабатываемого программного кода одной из важных характеристик полноты тестирования является показатель покрытия ветвей (альтернатив) кода исполненными тестами (так называемыми тестовыми пробами).

Для кода, созданного на наиболее распространенных языках (C, Java, Verilog) и исполняемого на наиболее распространенных платформах (Solaris, UNIX), для сбора подобной информации существует целый ряд специально предназначенных инструментальных средств.

Однако при работе с нестандартными платформами (например, в случае применения языка С для встраиваемых систем) подобрать подобные инструменты будет уже достаточно затруднительно. Аналогичная ситуация складывается и для не самых широко используемых, в частности, для интерпретируемых языков кодирования (JavaScript и другие).

В данной статье приводится однозначный и понятный метод внедрения тестовых проб в программный код. Метод интересен тем, что очень удобен для реализации систем преобразования исходного кода (source-to-source), имеющих промышленное применение.

Надо отметить, что подобные системы имеют несколько реализаций и успешно используются на практике. И неоспоримым их преимуществом является то, что такие инструменты тестирования покрытия кода могут легко применяться ко всем видам программного обеспечения во всех видах сред исполнения.

Подходы к обеспечению тестового покрытия

Организация и исполнение процесса тестирования программного обеспечения сами по себе становятся довольно сложной задачей. А иметь твердую уверенность в том, что программный код протестирован достаточно тщательно, зачастую бывает еще сложнее.

При проведении оценки покрытия кода широко используется метод суммарного учета покрытия ветвей кода. Здесь в качестве критерия оценки вычисляется доля базовых блоков кода, которые подвергались тому или иному виду тестирования, по отношению к общему числу блоков кода в тестируемой системе.

И на первый взгляд выполнить такую оценку не слишком сложно. Нужно разделить полный код программы на базовые блоки, в каждый из которых вставить тестовую пробу, запустить на исполнение предустановленный набор тестов, а затем по результатам исполнения тестов собрать информацию обо всех пройденных пробах в коде.

На практике же подобные манипуляции с кодом довольно затруднительны, поскольку из-за различия синтаксиса языков программирования достоверно распределить код по базовым блокам для автоматического программного инструмента — непростая задача.

Существует два принципиальных подхода к обеспечению тестового покрытия: модификация объектного кода и модификация исходного кода.

Модификация исходного кода

В подходе, работающем с модификацией исходного кода, все достаточно несложно и понятно. Программа разбивается на базовые блоки, и в каждый из них вставляется одна или несколько строк кода, работающих как тестовые пробы и сигнализирующих о том, что данный блок кода исполнен.

В листинге 7 приведена программа (классический пример вычисления чисел Фибоначчи) на языке С, в которую добавлены и выделены курсивом тестовые пробы:

```
int fibcache[1000]; // изначально обнулен

int fib(int i) // быстрое вычисление чисел Фибоначчи
{ int t;
   visited[1]=1;
   switch (i)
   { case 0: visited[2]=1;
      case 1: visited[3]=1;
         return 1;
      default:
         visited[4]=1;
         if (fibcache(i))
            { visited[5]=1;
            return fibcache(i);}
         else { visited[6]=1;
            t=fib(i-1);
            fibcache(i)=t+fib(i-2)
            return fibcache(i);
            };
   };
   visited[7]=1;
};
```

Листинг 7. Модификация исходного кода

Каждый базовый блок получает уникальный идентификатор, который связан исходным файлом данного блока и номером строки в этом файле.

При инициализации программы все флаги сбрасываются, и далее в ходе исполнения программы устанавливаются для пройденных блоков кода. Затем, с помощью дополнительных средств, выполняется сбор и обработка результатов.

Дополнительно так можно получать информацию о том, остались ли части программы, для которых не выполнено ни единого теста. Эта информация имеет высокую степень важности, ведь именно такой, не подвергнутый тестированию код зачастую может оказаться недостоверным.

В сущности, в процессе идентификации таких базовых блоков кода программа подвергается подробному синтаксическому и семантическому анализу, далее разбивается на блоки, в которые вносятся соответствующие тестовые пробы.

А это означает потребность в работе с полноценным внешним интерфейсом данного языка, что для большинства используемых на практике языков превращается в довольно сложную работу. (Здесь следует принимать во внимание, что решения на базе манипуляции со строками по аналогии с PERL не являются надежными ввиду различных лексических правил, условной компиляции и др.).

Модификация объектного кода

При подходе, работающем с модификацией объектного кода, проблема понимания синтаксиса языка отсутствует (по сравнению с модификацией исходного кода), зато приобретает актуальность задача понимания и патчинга машинного кода для данной конкретной программной платформы. Также необходимо коррелировать машинные инструкции со строками исходного кода.

Следует отметить одно весомое преимущество такого подхода. Оно заключается в том, что тестовое покрытие здесь реализуется практически для всех языков, компилируемых в рамках данной платформы. А потому этот метод прекрасно подходит и широко используется для большинства распространенных стандартных программных платформ, поскольку затраты на его реализацию быстро амортизируются.

Ограничения для нестандартных программных систем

Но, к большому сожалению, метод модификации объектного кода при всех его достоинствах не может применяться для нестандартных сред исполнения или для нестандартных или интерпретируемых языков. Но ведь именно так и реализовано огромное число программ, имеющих широкое промышленное применение. Получается, что обычные методы для таких систем перестают работать. Но как же тогда решать проблему измерения качества тестирования кода?

Очевидным решением было бы использование механизма внедрения тестовых проб в исходный код таких программ. Однако убеждать каждого производителя включать во внешний интерфейс каждого разрабатываемого им языка программирования механизм внедрения проб попросту нецелесообразно.

В такой ситуации организация — разработчик программ для встраиваемых систем будет вынуждена самостоятельно подбирать себе подходящий инструментарий компилятора и использовать его для достаточного покрытия кода тестовыми пробами.

И здесь от специалистов службы тестирования потребуется дополнительное овладение сложными навыками, которые, к слову сказать, не являются частью их основной деятельности. К тому же это займет немалую часть их рабочего времени. Вот почему данный подход не пользуется особой популярностью.

Практика показывает, что более удобным средством внедрения тестовых проб в код является использование автоматизированных систем преобразования исходного кода (source-to-source transformation systems).

Разумеется, здесь от специалистов-исполнителей также потребуется определенная квалификация, но овладеть ею значительно проще, чем в первом случае, и занимает этот процесс, как правило, гораздо меньше времени. В результате хорошо настроенная система позволит легко и удобно преобразовывать исходный программный код для вставки необходимых тестовых проб.

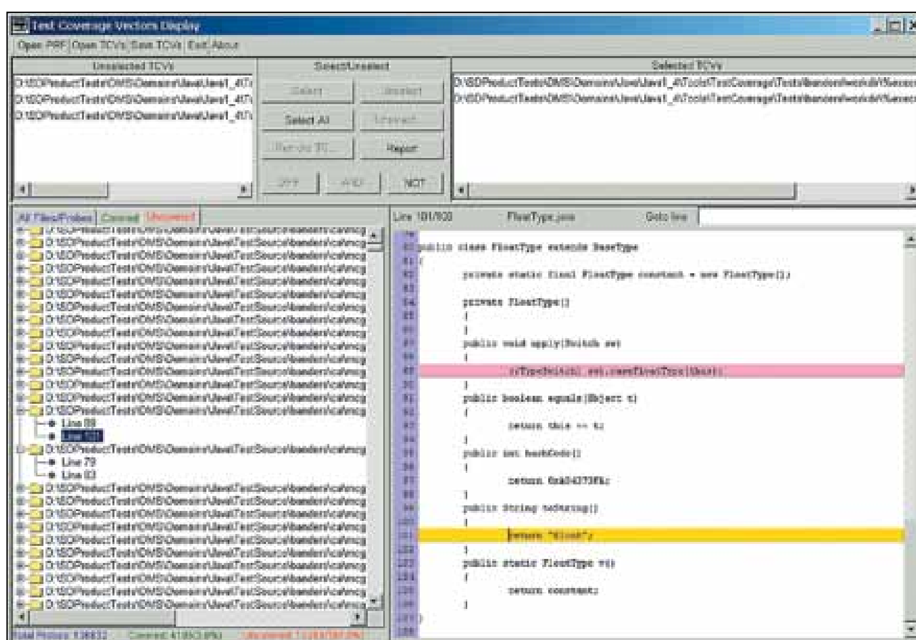

Рис. 3. Просмотр отчета данных тестового покрытия в системе DMS

Решение задачи тестового покрытия средствами инструментария DMS

Рассмотрим, как задача тестового покрытия может быть решена средствами инструментария, предоставляемого системой преобразования исходного программного кода DMS Software Reengineering Toolkit, о котором уже шла речь в статье.

DMS работает по следующему алгоритму:

- До запуска компиляции и исполнения код исследуемых исходных файлов разбивается на базовые блоки. В каждый из них вставляются тестовые пробы, написанные на языке исходных файлов.
- Во время исполнения программы происходит срабатывание тестовых проб в тех блоках кода, которые были исполнены. Таким образом, формируются так называемые данные тестового покрытия (coverage data).
- После завершения исполнения программы все данные тестового покрытия попадают в файл отчета о тестовом покрытии.
- Также данные тестового покрытия отображаются на пользовательский интерфейс системы DMS (рис. 3). По этим данным инженер, выполняющий тестирование системы, может видеть, какой код был исполнен, а какой нет. Также ему предоставляется полная статистика по собранным данным о тестовом покрытии.

При этом система DMS способна обеспечивать оценку тестового покрытия для программ, написанных на любых процедурных языках кодирования.

Пример преобразования кода для оценки тестового покрытия

Приведем примеры преобразования кода на языке С для обеспечения тестового покрытия средствами системы DMS. На первом этапе составляется перечень правил, используемых для перезаписи кода (листинг 8):

```
external pattern new_place
      (x:statement_sequence).

rule mark_function_entry
   (result:type,
      name:identifier,
      decls:declaration_list,
   stmts:statement_sequence) =
"\result \name { \decls \stmts };"
      rewrites to
"\result \name
   { \decls
      { visited[\new_place\(\stmts\)]=1;
         \stmts }};".

rule mark_if_then_else
   (condition:expression;
      tstmt:statement;estmt:statement) =
"if (\condition)\tstmt else \estmt;"
      rewrites to
"if (\condition)
      { visited[\new_place\(\tstmt\)]=1;
         \tstmt}
   else {visited[\new_place\(\estmt\)]=1;
      \estmt};".

rule mark_switch_case
   (condition:expression,
      stmts:statement_sequence) =
"case \e: \stmts"
      rewrites to
"case \e:
      { visited[\new_place\(\stmts\)]=1;
         \stmts }".
```

Листинг 8. Правила для преобразования кода программ на языке С

Для процедурных языков определение этих правил достаточно несложно, поскольку синтаксис таких языков предусматривает явное указание всех точек передачи управления.

Также составляется спецификация вычислителя атрибутов для определения используемых констант (листинг 9):

```
ATTRIBUTES { SetOfReal Constants; }
Statement = 'return' Expression ;
   <<UseConstants>> { Statement.Constants = Expression.Constants; }
Expression = Sum ;
   <<UseConstants>> { Expression.Constants = Sum.Constants; }
Sum = Sum '-' Term ;
   <<UseConstants>> { Sum[0].Constants = UnionReals(Sum[1].
Constants,Term.Constants); }
Sum = Sum '+' Term ;
   <<UseConstants>> { Sum[0].Constants = UnionReals(Sum[1].
Constants,Term.Constants); }
Term = '(' Expression ')' ;
   <<UseConstants>> { Term.Constants = Expression.Constants };
Term = IDENTIFIER ;
   <<UseConstants>> { Term.Constants = EmptySetOfReals(); }
Term = NATURAL ;
   <<UseConstants>> { Term.Constants = SingletonRealSetFromNat
ural(NATURAL.); }
Term = DOUBLE ;
   <<UseConstants>> { Term.Constants = SingletonRealSetFromDo
uble(DOUBLE.); }
Term = IDENTIFER '(' ArgumentList ')' ;
   <<UseConstants>> { Term.Constants = ArgumentList.Constants; }
ArgumentList = Expression ;
   <<UseConstants>> { ArgumentList.Constants = Expression.
Constants; }
ArgumentList = ArgumentList ',' Expression ;
   <<UseConstants>> { ArgumentList[0].Constants =
UnionReals(ArgumentList[1].Constants,
Expression.Constants); }
```

Листинг 9. Спецификация вычислителя атрибутов для определения используемых констант

По сути, для каждого выделенного в коде синтаксического фрагмента передачи управления нужно выполнить всего одно преобразование. И каждое такое преобразование состоит из двух этапов.

Сначала для каждого выполняемого преобразования выделяется новая нумерованная область рабочего пространства. Функция построения абстрактных деревьев new_place генерирует новый идентификатор области при каждом вызове, присваивает этот идентификатор обрабатываемой ветке конкретного дерева и связывает его с файлом исходного кода и номером строки в этом файле.

Для того чтобы выполнить такую обработку для всего кода программной системы, используемый инструмент должен генерировать эти идентификаторы таким образом, чтобы они оставались уникальными для всего пространства исследуемых исходных файлов. (Как упоминалось ранее, для системы DMS это пространство может исчисляться десятками тысяч исходных файлов, обрабатываемых за один сеанс преобразования кода.)

На втором этапе в код внедряются тестовые пробы. И здесь принципиально важно соблюдение следующего принципа: если в обрабатываемом участке кода встречается условный блок и он содержит инструкцию передачи управления (листинг 10), то тестовая проба должна включаться в код обязательно до любого оператора, следующего за условным блоком (листинг 11).

```
if (<условие>)
   { x=y;
      return;
   }
<дальнейший код>
```

Листинг 10. Передача управления из условного блока кода

```
if (<условие>)
   { x=y;
      return;
   }
visited[<указатель_местонахождения_пробы>]=true;
<дальнейший код>
```

Листинг 11. Внедрение тестовой пробы после условного блока кода

Это ограничение касается незначительного числа правил и необходимо для учета всех случаев передачи управления, включая случаи передачи управления из условных блоков. Для краткости в данной статье не рассматриваются подробности этих преобразований. Применение таких трансформаций к исходному программному коду дает результат, представленный в листинге 7 (в котором показан пример вычисления чисел Фибоначчи). Аналогичным образом достаточно легко можно придумать и другие интересные типы проб. Например, при обработке преобразований можно выполнять инкремент счетчика visited и таким способом использовать пробы не для тестового покрытия, а для профилирования.

Дополнительные замечания по инфраструктуре

Приведем перечень дополнительных замечаний по инфраструктуре систем преобразования, применяемых для реализации тестового покрытия. Для того чтобы извлечь действительную пользу из проведенного тестового покрытия, помимо внедрения в код тестовых проб, необходимо провести несколько дополнительных действий. И это достаточно простые действия, выполнить их можно и вручную.

Во-первых, для исследуемой программной системы следует написать небольшой дополнительный код, который нужно поместить в несколько разных мест исходного кода (вставить этот код можно или вручную, или с помощью скрипта). А именно:

- Однострочное объявление массива visited. Инструмент преобразования кода может сообщить о наибольшем идентификаторе new place number, исходя их размера массива после внедрения тестовых проб.
- Функция инициализации, которая сбрасывает массив visited при запуске тестирования исследуемой системы.
- Механизм сбора данных, который записывает результат заполнения массива visited и хранит его в некотором месте после завершения тестирования. В частности, для встраиваемых программных систем можно помещать эту информацию в некоторое внешнее хранилище используемой среды разработки.

Во-вторых, система хранения результатов должна накапливать информацию о результатах нескольких проведенных тестов. Так, для тестового покрытия достаточно просто складывать по оператору OR значение последнего полученного экземпляра массива visited в суммарный массив was_visited_by_some_test. Кроме того, для создания более подробного отчета можно подсчитывать и количество тестов, проведенных над конкретными блоками кода.

Для профилирования же значение массива visited нужно складывать поэлементно и добавлять к суммарному результату.

В-третьих, для интерпретации результатов покрытия необходимо обеспечить перекрестные ссылки для идентификаторов местоположения тестовых проб и соответствующих исходных файлов кода. В частности, в системе DMS есть удобная настройка конфигурации, позволяющая автоматически добавлять привязку всех внедряемых проб во всех файлах.

И наконец, потребуется какой-либо несложный инструмент отображения результатов проведенной работы. Это могут быть значения массива was_visited_by_some_test и прочая статистическая информация по исследуемой системе, полностью или по обозначенному диапазону конкретных файлов.

Можно использовать функциональные возможности системы DMS, обеспечивающие наглядное отображение результатов. Или представить файлы в удобочитаемом виде в формате HTML, а также выполнить многоцветную подсветку участков кода по принципу пройдено/не пройдено в ходе тестирования.

Принимая во внимание квалификацию специалистов группы тестирования, реализация и внедрение всей этой дополнительной инфраструктуры представляется достаточно несложной. На рис. 3 представлена реализация такого инструмента отображения результатов тестового покрытия на языке Java. Здесь можно просмотреть результаты покрытия по любому выбранному файлу, увидеть строки кода, в которые были вставлены тестовые пробы, а также оценить статус покрытия для конкретных тестовых проб.

Дополнительно данный инструмент отображения включает следующую полезную возможность — калькулятор двоичных битовых векторов для объединения векторов по OR, AND и AND NOT. Это помогает инженеру, проводящему тестирование, легко определять, какие наборы тестов перекрывают друг друга и какие тесты являются дубликатами других тестов в рамках одного или нескольких наборов тестов.

Наконец, этот инструмент предоставляет сводную информацию о результатах покрытия, которая может использоваться для удобного хранения результатов и построения отчетов.

Практические результаты тестового покрытия от DMS

Описанная технология реализована разработчиками DMS в инструментальных средствах преобразования исходного программного кода и позволяет решать задачи тестового покрытия кода, написанного на следующих языках:

- ANSI 89 C. Для обработки директив препроцессора используется специальный внешний интерфейс. Для других диалектов языка, таких как GNU C, Microsoft Visual C++ и ANSI 99 C, подразумевается ряд простых расширений. Поддержка языка C++ требует большего числа правил, но базовые принципы те же.
- ANSI COBOL 85.
- Java 1.1 и Java 1.3.
- PARLANSE (внутренний язык реализации системы DMS).

Поддержка каждого из языков требует применения отдельного набора преобразований для внедрения тестовых проб, характерных для конкретного языка. Однако все они используют один и тот же инструмент отображения результатов проведенного тестового покрытия (он описан в предыдущем разделе, и его интерфейс показан на рис. 3). С его помощью были успешно проведены практические испытания работы с очень объемными программными системами, включающими более 3500 исходных файлов, что потребовало внедрения порядка 77 000 тестовых проб. Анализ испытаний показал довольно неплохие результаты: перекрытие тестовых проб в узких местах составило около 50%, а в среднем по всему приложению не превысило 15%.

Тестовое покрытие кода является важной мерой оценки качества разрабатываемых программных систем.

Возможность получения информации о покрытии для не самых распространенных языков кодирования или сред исполнения программ всегда сопряжена с большим числом определенных сложностей. Это вызвано необходимостью иметь в распоряжении достаточно сложную технологию патчинга объектного кода или же комплексного синтаксического анализа.

В сложившейся ситуации системы преобразования исходного программного кода, имеющие промышленное применение, предлагают достаточно удобную реализацию тестового покрытия. Предоставляются следующие возможности:

- Полный набор определений конкретного обрабатываемого языка кодирования.
- Написание небольшого набора правил для преобразований исходного кода (по принципу source-to-source).
- Реализация небольшого набора дополнительных сопроводительных процедур для инициализации, сбора, отображения и анализа статистической информации.

В приведенных примерах показаны принципы построения практических инструментов тестового покрытия для отдельно взятых языков и сред исполнения. Это хороший способ получить полезные статистические данные о качестве тестирования, а также важную информацию о том, какие участки кода не подвергались тестированию. Таким образом, можно обеспечивать тестовое покрытие для большого числа приложений, для которых подобные инструменты ранее не были доступны.

Заключение

Применение реинжиниринга для создания, развития и усовершенствования программных и аппаратных систем обеспечивает возможность удобного масштабирования систем и помогает в планировании и реализации долгосрочных целей проектирования, обновления и поддержки разрабатываемых приложений. А умелое использование соответствующих инструментальных средств позволяет упростить работу за счет автоматизации наиболее трудоемких ее этапов.

Рассмотрены функциональные возможности системы программного реинжиниринга DMS Software Reengineering Toolkit. Перечислены основные достоинства системы, в числе которых поддержка разностороннего усовершенствования встраиваемых программных приложений, имеющих промышленное применение. Поддерживается обработка кода приложений, написанного на большинстве известных языков кодирования и работающих практически на любых программных платформах.

В качестве одной из ключевых возможностей системы DMS рассмотрено конфигурируемое автоматическое преобразование исходного программного кода (перевод кода с одного языка на другой или оптимизация кода в рамках одного домена). А также описано прикладное применение этой возможности для решения такой существенной задачи, как выполнение и анализ тестового покрытия разрабатываемого программного кода с целью обеспечения его надлежащего качества.

Приведены практические примеры и результаты производственных испытаний работы системы DMS.

Разумеется, такие системы, как DMS и аналоги, предоставляют гораздо более широкий спектр инструментальных и функциональных возможностей для автоматизации реинжиниринга, чем рассмотрено в рамках данной статьи.

Но, тем не менее, все они служат одной цели — созданию и долгосрочному развитию и поддержке надежных, быстродействующих и высокопроизводительных программных приложений и аппаратуры, которые так востребованы при реализации в современных встраиваемых системах.

Литература

1. Сергеева А. Тестирование работоспособности промышленного компьютера // Компоненты и технологии. 2014. № 2.
2. Сергеева А. Одновременная разработка программ и аппаратуры для встраиваемых систем при помощи симулятора аппаратуры Vista Virtual Prototyping // Компоненты и технологии. 2014. № 3.
3. Chikofsky E., Cross J. Reverse Engineering and Design Recovery: A Taxonomy. — IEEE Software, 1990.
4. www.reasoning.com
5. www.strategoxt.org
6. www.semdesigns.com
7. Об инструментарии DMS Software Reengineering Toolkit — http://www.semdesigns.com/products/DMS/DMSToolkit.html
8. О проекте JOVIAL2C — http://www.tgdaily.com/technology/42426-us-upgrades-stealth-bomber-software