УДК 004.052

АВТОМАТИЗАЦИЯ ТЕСТИРОВАНИЯ ПРОГРАММ, РАЗРАБАТЫВАЕМЫХ НА НЕБЕЗОПАСНЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ (НА ПРИМЕРЕ ЯЗЫКА С).

Якимов И.А.,
научный руководитель канд. техн. наук Кузнецов А.С.
Сибирский Федеральный Университет

Введение

Для облегчения процесса тестирования создаются библиотеки и статические анализаторы, генераторы тестов по спецификациям, которые будут рассмотрены ниже.

После рассмотрения существующих подходов будет описан инструмент, предлагаемый к разработке. В качестве целевого языка выбран C99, который на данный момент используется в основном во встраиваемых системах.

Следует отметить, что проведение тестирования не является достаточным для доказательства верности работы программы, но хорошее тестирование может устранить большинство наиболее опасных ошибок [1].

Статический анализ без формальных спецификаций

Статические анализаторы без формальных спецификаций, например Cppcheck, svace позволяют проверять программу еще до ее запуска, опираясь исключительно на исходные коды. Они покрывают задачи проверки надежности и безопасности.

Статические анализаторы просты в применении - программисту не нужно писать дополнительный код для выявления ошибок. Но такой подход сужает круг решаемых задач, в частности, тестирование логики становится невозможным.

Тестирование на основе модели

Существуют системы тестирования на основе формальных спецификаций (моделей), например CTESK, реализующий технологию UniTESK.

Существенных ограничений на свойства тестируемого программного обеспечения нет. Однако применение CTESK наиболее эффективно при тестировании прикладных программных интерфейсов систем. В других случаях задача решается путем разработки промежуточного слоя программного обеспечения и применения CTESK к его прикладному интерфейсу [2].

В общем случае описание подобных спецификаций является задачей нетривиальной и, как показывает практика, многие программисты предпочитают использовать библиотеки модульного тестирования и обычные статические анализаторы.

Модульные тесты

Примерами «фреймворков» для модульного тестирования могут, служит, например CUnit, cmockery. Стратегией модульного тестирования является написание специальных процедур для проверки модулей программы. При этом средства создания модульных тестов, которые зачастую представлены в форме подключаемых к проекту библиотек, не используют, информацию о структуре программы.

Преимуществом модульных тестов является простота использования, но прописывать их приходится вручную, на языке целевой программы (который изначально разработан для других целей), что, в конечном счете, сводится к рутинной однообразной работе. Именно этот процесс и предполагается автоматизировать.

Постановка задачи

После рассмотрения наиболее распространенных подходов к тестированию можно приступить к описанию проектируемой системы, которая должна:

- Позволять тестировать логику, надежность, целостность;
- Анализировать исходный код программы для генерации или проверки тестовых случаев;
- Позволять описывать тестовые случаи простым и интуитивно понятным способом.

Таким образом, от анализа сложившейся ситуации и выработки требований мы можем перейти к вопросам реализации.

Единица трансляции

Прежде чем перейти к вопросам реализации, необходимо рассмотреть вопрос о том, что такое единица трансляции. Она состоит из последовательности внешних объявлений, каждое из которых представляет собой либо объявление, либо определение функции [3].

Стратегия автоматизации

Для автоматической генерации может быть выбрана следующая стратегия: генерация отдельной единицы трансляции для тестов, которая будучи скомпилированной отдельно от тестируемого блока программы, объединяется с ним при помощи компоновщика.

Система анализирует код тестируемого модуля и генерирует входные данные так, чтобы тесты прошли все возможные потоки выполнения. Но при этом верификации операторов не происходит. Анализатор только генерирует входные данные. Программист сам задает поведение функции посредством таблицы, которая кратко описана дальше. Возможен и альтернативных подход – в этом случае анализатор выступает в роли некой экспертной системы, которая выдает предупреждение в случае, если программист описал недостаточное количество входных данных для тестирования всех возможных путей выполнения.

Специализированный язык

Очевидно, что для разрабатываемой системы нужен внутренний язык. Существует множество языков сценариев, которые можно встраивать в приложения (Lua, JavaScript и т.д.). Но т.к. задача тестирования в корне отличается от написания пользовательских программ, то разумным решением будет создание специализированного языка описания тестов, полностью отвечающего поставленной цели.

Объекты тестирования

Язык С является процедурным. Поэтому объектами модульного тестирования должны служить процедуры (функции). О них речь и пойдет дальше.

Табличные функции

По аналогии с математикой, функцию, как объект тестирования, можно задать таблично.

F: {{I1, O1},{I2, O2},…{In, On}},

I : {{параметр1, параметр2 …}, {глобальная_переменная1, …}},

O : {{возврат} , {выходной_параметр1, …}, { глобальная_переменная1, …}}.

F – имя функции, I – вход, O – выход.

Т.е. для каждой тестируемой функции описывается ее поведение для ряда случаев, определяемых посредством входным параметров. При этом глобальные переменные так же рассматриваются как входные параметры. Система должна проверить соответствие функции, написанной программистом с функцией, заданной таблично.

Таблицы могут использоваться в двух случаях.

Для генерации тестов, в данном варианте каждый тестовый случай соответствует очередной строке таблицы.

Для генерации функций-заглушек, речь о которых пойдет далее.

Bison-подобная нотация описания синтаксиса строки таблицы:

```
table_row:
      param_list
      “=>”
      value
      param_list
      ‘;’
      ;
param_list:
      NAME ‘=’ value ‘,’ param_list
      | NAME ‘=’ value
      |
      ;
```

Пример описания таблицы

В качестве примера табличного описания функции возьмем функцию возведения в степень.

```
pow(int x, int n)
{
      int i = 0;
      if (n < 0) return -1;
      if ( x == 0 && n == 0) return -1;
      if ( x != 0 && n == 0) return 1;
      for (i; i < n; i++)
            x*=x;
      return x;
}
```

Описание таблицы на декларативном языке:

```
pow:
x = 0, n = -1   => -1;
x = 0, n = 0    => -1;
x = 2, n = 0    => 1;
x = 2, n = 2    => 4;
```

Корректность начального состояния

Для корректности выполнения тестов нужно реализовать процедуру инициализации (сброса) состояния программы перед запуском очередного теста, т.е. например если тестируемая функция изменяет глобальную переменную, то перед тестированием следующей функции нужно привести программу (модуль) к состоянию по умолчанию, либо состоянию, заданному пользователем. Такое описание состояния должно поддерживаться встроенным языком системы.

Независимое тестирование модулей

Для обеспечения независимого тестирования модулей можно реализовать механизм формирования фиктивных функций. Распространенным подходом является использование mock-функций[4], которые работают по принципу очереди, т.е. программист прописывает для каждого теста перед вызовом тестируемой функции, посредством вызова функции will_return (на примере google cmockery) нужное значение возврата для функций-заглушек в порядке их следования в тестируемой функции. Таблицы функций реализуют схожее поведение, но иным образом, в данном случае жестко прописывать порядок не нужно.

Пусть, например, имеется функция function, которая вызывает внешнюю функцию syb_function.

```
<тип> function(...)
{
   extern sub_function(...);
   ...
   sub_function(...);
   ...
}
```

Пусть пользователь хочет протестировать function, независимо от sub_function. В этом случае система ищет таблицу, определенную для sub_function, и генерирует заглушку. Далее тестируемый модуль компонуется с объектным файлом, содержащим фиктивную функцию вместо реальной.

Проблема указателей неизвестного типа

Подходя к концу, хотелось бы коротко охарактеризовать проблему, свойственную языку С и ему подобным.

Работа с указателями являет собой некоторую сложность. Мы можем проверять значение переменной, на которую ссылается указатель, если он образован от базового или пользовательского типа. Но существует частный случай указателей – void*, тип которых не определен. В этом случае структура программного объекта, на который он ссылается неизвестна. Система тестирования должна располагать средствами добавления метаинформации о структуре типов, на которые он указывает. Однако следует помнить, что указатели void* зачастую используются для имитации наследования, которое отсутствует в языке С в явном виде. Эту проблему можно решить, если ввести возможность описания наследования пользовательских типов (структур и объединений), на которые ссылается void* в языке системы.

Заключение

В результате исследования был выработан подход к созданию системы автоматической генерации тестов. Были определены цели, описаны ключевые концепции. Дальнейшая работа будет направлена на поиск, синтез и обоснование алгоритмов анализа исходных текстов, генерации таблиц функций и, на их основе, непосредственно тестов. Так же планируется произвести поиск дополнительных путей автоматизации процесса модульного тестирования.

Использованные источники:

1. Стив МакКонел - «Совершенный код»;
2. http://www.unitesk.ru;
3. Брайан Керниган, Деннис Ритчи - «Программирование на С», второе издание;
4. http://cmockery.googlecode.com;