Модел. и анализ информ. систем. Т. **21**, № **6** (2014) 83–93

УДК 004.054+004.4’23

Использование метода ограниченной проверки моделей для генерации тестов

Петров М.А., Гагарский К.А., Беляев М.А., Ицыксон В.М.

Санкт-Петербургский государственный политехнический университет
195251, Россия, г. Санкт-Петербург, Политехническая ул., 29

e-mail: {maxim.petrov,gagarski,belyaev}@kspt.icc.spbstu.ru, vlad@icc.spbstu.ru

получена 15 сентября 2014

Ключевые слова: автоматическая генерация тестов, динамическое символьное выполнение, ограниченная проверка моделей, контракты кода, SMT

В настоящее время автоматическая генерация тестов исследуется всѣ более и более активно, поскольку является решением многих проблем, связанных с тестированием программного обеспечния (ПО), таких как необходимость написания тестов и обеспечения тестового покрытия с учѣтом человеческого фактора.

Наиболее перспективным методом автоматической генерации тестов является динамическое символьное исполнение (dynamic symbolic execution, DSE), выполняемое с помощью автоматического решателя ограничений, например SMT-решателя. Данный метод во многом похож на метод ограниченной проверки моделей, который также предполагает построение моделей из исходного кода, проверку логических свойств на них и обработку полученной модели.

В данной работе рассматривается метод генерации модульных тестов для языка C, основанный на методе ограниченной проверки моделей. Показано, что эти методы имеют много общего и могут быть реализованы с использованием общих базовых компонентов. Реализован прототип средства генерации тестов, основанный на работающем средстве ограниченной проверки моделей Borealis.

Прототип средства был опробован на множестве примеров и показал хорошие результаты с точки зрения полноты тестового покрытия и избыточности тестового набора.

Введение

В современном мире программное обеспечение (ПО) всѣ шире применяется в различных областях человеческой деятельности, включая медицинское оборудование, космическую отрасль и атомные электростанции. Ошибки разработчика могут привести к значительным потерям и ущербу.

Стандартом де-факто в автоматизации обеспечения качества ПО является тестирование, проведение которого связано с рядом проблем. Во-первых, тестирование требует человеческих ресурсов и времени для написания и поддержки тестов.

Во-вторых, проблемой является человеческий фактор. Программисты склонны делать те же предположения (зачастую неправильные) о данных в тестах, какие они делают и при написании программного кода [21].

Одним из возможных решений вышеописанных проблем является автоматическая генерация тестов, активно исследуемая в последние годы [22, 12, 9]. В настоящее время наиболее широко используемым подходом к автоматической генерации тестов является *динамическое символьное выполнение* (dynamic symbolic execution, DSE) [26]. Данный подход основан на вычислении входных данных, обеспечивающих желаемое поведение тестов при выполнении, при помощи автоматического решателя ограничений (например SMT-решателя). Данный подход во многом похож на метод ограниченной проверки моделей (bounded model checking, BMC), хотя и преследует другие цели (BMC предназначен для поиска дефектов в ПО и нарушений пользовательских контрактов кода). Сочетание BMC и генерации тестов в одном программном средстве вполне естественно, поскольку генерация тестов и верификация ПО дополняют друг друга при обеспечении качества ПО.

Данная статья посвящена разработке подхода к автоматической генерации тестов для программ на языке C на основе ограниченной проверки моделей. В качестве основы для построения прототипа средства генерации тестов используется средство ограниченной проверки моделей Borealis [1]. В данной статье показано, что изменения, необходимые для адаптации средства BMC для решения задачи генерации модульных тестов, минимальны в сравнении с отдельной реализацией двух средств. Подход был опробован на множестве примеров и показал свою эффективность.

Оставшаяся часть статьи организована следующим образом. В певром разделе кратко описываются вопросы SMT, ограниченной проверки моделей и генерации тестов. В разделе 2 подробно описывается разрабатываемый подход. В разделе 3 приведены детали реализации разрабатываемого средства. В последнем разделе приведены результаты экспериментов.

1. Основы

1.1. Метод ограниченной проверки моделей

Метод проверки моделей — широко известный подход к проверке корректности и безопасности программного кода без его исполнения, использующий полное исследование пространства состояний. Хотя метод проверки моделей работает хорошо в системах с конечным числом состояний, его эффективность падает в больших программах из-за неконтролируемого экспоненциального роста пространства состояний. Один из способов решения этой проблемы — метод ограниченной проверки моделей [7]. Этот метод основан на сокращении пространства состояний путчм ограничения длины анализируемых путей программы, например ограничения числа итераций цикла и глубины рекурсии. Ограниченная модель может быть преобразована в формулу в логике первого порядка, а затем решена при помощи логического решателя, обычно SMT-решателя, поддерживающего необходимые теории.

В последние годы метод ограниченной проверки моделей является активно исследуемой областью. Было представлено множество средств, основанных на данном методе (наиболее известные из них: CBMC [8], SMT-BMC [2], LLBMC [19],

ESBMC[10]) и предназначенных для обнаружения дефектов в ПО, нарушений контрактов и других проблем с программным кодом.

Средство, используемое как базовое в данной работе, — проект Borealis [1], основанный на инфраструктуре компиляторов LLVM и SMT-решателе Z3. Он способен искать дефекты, специфицировать и проверять контракты, написанные на языке C. Контракты задаются двумя способами: с помощью языка аннотаций, основанного на комментариях в программах на языке C и схожего с языком ACSL [3], а также с помощью встроенных в программный код вызовов специальных процедур.

1.2. Использование динамического символьного исполнения для генерации тестов

DSE [13, 24, 25] — современный метод генерации входных тестовых данных для тестирования функций методом белого ящика. Данный метод основан на исследовании ветвей для функций с известным исходным кодом путчм вывода входных данных, ведущих к выполнению различных ветвей кода. Цель DSE — решение этих задач эффективно.

DSE — метод, ставший развитием случайного тестирования с обратной связью [23], который, в свою очередь, основан на случайном тестировании [14] и решает широко известные проблемы случайных тестовых данных [5]. Средства генерации тестов, использующие DSE, имеют доступ к исходному коду программы, таким образом, DSE — тестирование методом «белого ящика». Информация, полученная из исходного кода, может быть использована для ограничения числа комбинаций входных данных, таким образом уменьшая сложность алгоритмов генерации тестов.

Существует множество подходов к генерации тестовых входных данных, основанных на коде программы. Одним из наиболее эффективных подходов является преобразование программы в логическую формулу, а возможных ветвей в логические ограничения, которые могут быть решены с помощью решателя ограничений (например SMT-решателя). Данный подход зачастую рассматривается как использование метода проверки моделей для генерации тестов. Более того, проблемы использования DSE для программ на языке C (глобальное изменяемое состояние, операции с памятью, циклы, вызовы функций) схожи с проблемами BMC. На данном подходе основывается средство Pex [25], являющееся де-факто ведущим средством генерации тестов для платформы `.Net`.

В данной работе предполагается, что ограниченная проверка моделей, основанная на SMT, и динамическое символьное исполнение, основанное на SMT, — похожие механизмы и могут быть реализованы с использованием одних и тех же базовых компонентов. Они отличаются способом формирования ограничений и тем, как используется полученная модель. BMC для обнаружения ошибок использует контракты и дефекты как ограничения, в то время как DSE использует ограничения, основанные на структуре потока управления программы. Полученная модель при поиске контрактов используется только для вывода контрольных сообщений, DSE же может использовать эту модель для генерации исходного кода тестов.

Ещч одна задача генерации тестов — генерация тестовых оракулов — не может быть осуществлена автоматически и должна быть основана на спецификациях. Эти

спецификации во многом похожи на проверяемые постусловия в BMC и могут быть использованы и при генерации тестов.

2. Использование BMC для генерации тестов

Данная статья посвящена общей идее реализации утилиты для генерации модульных тестов с использованием готовой реализации BMC средства. Более продвинутые методы генерации данных и проверки результатов не рассматриваются, поскольку не реализованы на данный момент.

2.1. Использование контрактов при генерации тестов

Качество автоматической генерации тестов может быть значительно улучшено с помощью задания ограничений на входные данные для того, чтобы исключить генерацию тестовых значений вне области определения параметров. Ограничения на выходные данные позволяют создавать тестовые оракулы. Под обе эти задачи хорошо подходят контракты для функций в форме предусловий (*requires*) и постусловий (*ensures*). Эти спецификации широко используются в BMC для задания желаемого поведения кода как при разработке с использованием тестов [4], так и с использованием контрактов [20].

Аннотации кода в форме предусловий, постусловий и утверждений часто используются BMC средствами. Наиболее известные тестовые наборы для BMC (например NECLA [15] и SVCOMP [6]) больше сконцентрированы на проверке контрактов, чем на поиске программных дефектов. Эти возможности можно использовать для получения информации о контрактах во время генерации тестов.

2.2. Требования к системе генерации тестов

Для оценки результата генерации тестов используется два основных параметра: тестовое покрытие и избыточность тестов. Для измерения покрытия используется покрытие по операторам, поскольку оно позволяет соблюсти баланс между качеством тестирования и количеством тестов. Лучших результатов можно достигнуть, используя покрытие по путям исполнения, но это может привести к потребности в огромном числе требуемых тестов (например, для функции с 38 операторами `if` потребуется 2^{38} тестов). Покрытие по операторам хорошо подходит в большинстве случаев и может быть достигнуто, используя меньший тестовый набор (по сравнению с другими видами покрытий). Избыточность измеряется путчм выделения в наборе лишних тестов, то есть таких, которые никак не влияют на покрытие и могут быть удалены из полученного набора.

Рассмотрим простой пример аннотированной функции на языке C:

```
// @ensures \result >= 0
// @ensures \result == \arg || \result == -\arg
int abs(int a) {
    if (a >= 0) return a;
    else return -a;
}
```

Функция `abs` — пример простейшего из возможных (ромбовидного) графов потока управления. Он может быть полностью покрыт двумя тестовыми случаями, например, $a = \{1, -1\}$. Если средство генерации тестов сгенерирует три тестовых случая, то один из них будет лишним и общая избыточность тестового набора будет равна 33%. Если средство генерации тестов сгенерирует несколько тестовых случаев, среди которых будут только положительные, (например, $a = \{1, 2, 10042\}$), то оно достигнет только 50% покрытия по операторам. Цель этой работы — достичь 100% покрытия по операторам для произвольных программ на языке C при минимизации избыточности генерируемых тестовых наборов.

2.3. Извлечение предикатов и тестовые данные

Реализация BMC, используемая в данной работе (см. разд. 3), основана на представлении состояния программы как набора логических предикатов. Каждый предикат по сути является логической формулой одного из двух типов: предикат пути или предикат состояния. Предикат пути используется для разделения различных путей выполнения и напрямую соотносится с ветвлениями на графе потока управления функции. Предикаты состояния — это все остальные типы программных конструкций, которые не влияют на поток управления. Состояние предикатов — это либо просто набор предикатов, либо последовательность, либо выбор из нескольких возможных состояний, разделчнный условием из предиката пути. Используя эту модель, программа может быть представлена как одно составное состояние, которое можно упростить до логической формулы и, в то же время, избежать лишнего дублирования этих предикатов. Также она предоставляет простой способ отображения инструкций исходного кода в предикаты состояния, если исходный код представлен в форме статического однократного присваивания (static single assignment, SSA).

Каждая формула внутри предиката представляется в логике первого порядка с использованием битовых векторов, неинтерпретируемых функций и теорий массивов. Предикаты являются контексто-зависимыми из-за необходимости моделирования памяти и глобальных переменных. Решение использует теорию массивов для имитации памяти (по сути, создачтся новый SMT-массив для каждого нового состояния памяти) и для превращения состояния предикатов в единое SMT-выражение, необходимо интерпретировать предикаты на этом множестве массивов. Глобальные переменные представляются как особые элементы памяти и поэтому не требуют какой-либо специальной обработки.

Для обеспечения покрытия по операторам требуется создать набор SMT-формул, по одной на каждую точку исполнения после каждого условного оператора в программе. Эта задача сводится к построению состояний предикатов от точки начала функции до каждого предиката пути, а затем к генерации с помощью SMT-решателя модели, которая сопоставлена с простым путчм, покрывающим это состояние. Заметим, однако, что каждый путь состоит из нескольких блоков графа потока управления и не исключено, что покрытие нескольких предикатов пути приведчт к избыточности тестового набора. Если предикат невозможен (то есть решатель вернул UNSAT в результате обработки соответствующей формулы), то код, соответствующий этому предикату, считается «мчртвым», следовательно, он не может быть покрыт ни одним тестом.

3. Детали реализации

Прототип основывается на BMC проекте Borealis [1], который использует Clang [17] для разбора исходного кода, LLVM [18] для анализа кода и Z3 [11] в качестве решателя. Структура прототипа показана на рисунке 1.

Borealis взаимодействует с исходным кодом с помощью проходов LLVM — взаимозависимых операций над внутренним представлением LLVM. Средство генерации тестов реализовано в виде двух проходов. Первый проход вызывает процедуры Borealis для получения входных данных для каждого базового блока LLVM IR. Второй проход отвечает за сохранение полученных тестовых наборов в виде кода на языке C и добавление тестовых оракулов на основе данных о контрактах из Borealis. В качестве фреймворка тестирования используется `CUnit` [16].

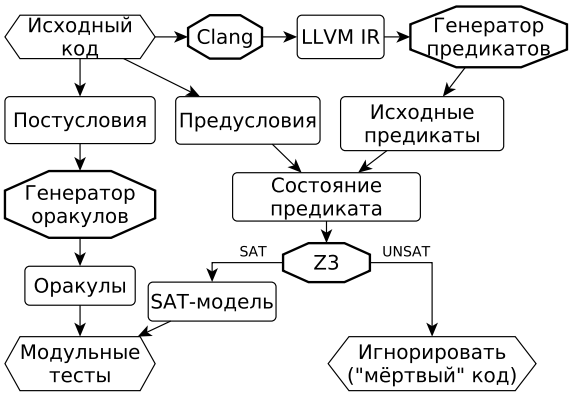

Рис. 1. Структура прототипа

3.1. Извлечение входных данных

Программа в LLVM представляется в виде LLVM IR, являющегося набором базовых блоков — последовательностей инструкций, которые не влияют на поток управления. Базовые блоки соединяются инструкциями ветвления (всегда находятся в конце базовых блоков) и так называемыми φ-функциями (всегда находятся в начале базовых блоков). Для обеспечения покрытия по операторам требуется запросить у Borealis состояние для начала каждого базового блока.

В целях уменьшения избыточности тестового покрытия принимаются дополнительные меры. Сквозные базовые блоки (то есть те, которые всегда выполняются безусловно) игнорируются, если только вся функция не состоит из одного базового блока. Блоки со входом в функцию пропускаются, так как всегда выполняются при вызове функции. Ещ̧ одной мерой является игнорирование одинаковых тестовых случаев, сгенерированных для разных блоков.

3.2. Обработка сложных типов данных

Для успешной генерации тестовых случаев необходимо сопоставить набор входных данных, полученный от SMT-решателя, с агрументами функции. Для простых типов, таких как `int`, `char`, `double` и т. п., это не вызывает затруднений, поскольку в

LLVM они представляются соответствующими типами, и имена аргументов совпадают. Структуры в LLVM могут представляться двумя способами: они могут представлятся набором целочисленных аргументов с общим размером, совпадающим с размером исходной структуры, либо указателем на структуру. Таким образом, результаты, полученные от SMT-решателя для структур, использовать для генерации тестов затруднительно. В связи с этим в прототипе реализовано разворачивание структур в отдельные элементы, то есть если в аргументах функции встречается структура, то для генерации тестов она преобразуется в набор аргументов, соответствующий еҷ полям. После генерации тестовых значений они собираются обратно в исходную структуру. Такой подход работает для структур любой вложенности, а также для массивов фиксированной длины, которые являются полями структуры.

3.3. Генерация исходного кода тестов

Каждый набор входных данных, полученный от SMT-решателя, является тестовым случаем. Для каждого тестового случая генерируется один вызов тестируемой функции.

Предусловия функции описывают ограничения для еҷ входных параметров. Они добавляются в предикаты состояний автоматически с помощью методов построения состояний и не требуют специальной обработки для задачи генерации тестов.

Постусловия используются для генерации тестовых оракулов. Они используют значения входных аргументов и возвращаемое значение (`\result`). Возвращаемое значение функции сохраняется в локальной переменной. Постусловия упрощаются и переводятся в операторы сравнения, которые вставляются в модульные тесты как оракулы.

Для сгенерированных файлов с исходными кодами тестов генерируется `Makefile` для сборки и запуска тестов.

3.4. Работа с пользовательскими тестовыми оракулами

В случае, если тестируемые функции аннотированы контрактами с постусловиями, эти постусловия преобразуются в условия на языке C, проверяемые с помощью макроса `CU_ASSERT`. Если аннотации отсутствуют, то автоматически можно сгенерировать тесты, обеспечивающие покрытие функции по операторам, но не проверяющие корректность работы функции. В таком случае требуется ручное вмешательство пользователя. Однако все модификации исходного кода будут потеряны при повторной генерации тестов. Сохранение внесҷнных пользователем оракулов непосредственно в тестах затруднительно, поскольку при перегенерации требуется провести сложный анализ существующих тестов, выявив изменҷнные и ветви графа потока управления, которым соответствуют изменҷнные тесты. Также требуется вносить тестовые оракулы отдельно в каждый тестовый случай.

Для решения данных проблем в разрабатываемом генераторе тестов имеется возможность генерации вызова пользовательской функции-оракула. Также автоматически генерируется код заглушки для неҷ. Данная функция является общей для всех тестовых случаев из тестового набора для функции. Пользовательский оракул принимает на вход все аргументы функции, а также еҷ возвращаемое значение. В

случае выполнения пользовательских контрактов оракул должен возвращать значение «1», в противном случае — «0».

Автоматически сгенерированная заглушка для пользовательского оракула выглядит следующим образом:

```
int absOracle(int a, unsigned int result) {
    // Put your oracle code here.
    return 1;
}
```

Пример пользовательского оракула для функции `abs`:

```
int absOracle(int a, unsigned int result) {
  if (a >= 0)
    return (result == a);
  else
    return (result == -a);
}
```

Вызов функции-оракула производится во всех тестовых случаях после вызова тестируемой функции. При повторной генерации тестов сохраняются ранее сгенерированные и изменчнные пользователем оракулы, а также добавляются заглушки оракулов для новых функций.

4. Экспериментальные исследования

В данном разделе приведены примеры сгенерированных тестов для функции `abs` из раздела 2.2. Тестовые данные удовлетворяют предусловиям, а постусловия проверяются с помощью функции `CU_ASSERT`.

```
void testAbs_0(void) {
    int a = -1;
    int res = abs(a);
    CU_ASSERT((res > 0));
    CU_ASSERT(((res == a) || (res == -(a))));
    CU_ASSERT(testAbsOracle(a, result));
}
void testAbs_1(void) {
    int a = 1;
    int res = abs(a);
    CU_ASSERT((res > 0));
    CU_ASSERT(((res == a) || (res == -(a))));
    CU_ASSERT(testAbsOracle(a, result));
}
```

Разработанный прототип средства был опробован на множестве примеров. Результаты приведены в таблице 1. Примеры, взятые для тестирования, являются искусственными. Как видно, покрытие по операторам составило 100% для всех тестовых примеров. Значение избыточности в 75% (максимальное значение в таблице) означает, что для 10 полезных тестовых примеров генерируется приверно 7 избыточных, что является приемлемым результатом.

Заключение

Данная работа посвящена разработке метода генерации модульных тестов на основе ограниченной проверки моделей и прототипа программного средства, реализующего

Таблица 1. Результаты экспериментальных исследований

Тестируемая функция	Тестовые случаи	Избыточные тестовые случаи	Покрытие по операторам (без уччта «мчртвого» кода)	Покрытие по путям	Избыточность
Сумма натуральных чисел	7	0	100%	<1%	0%
Модуль числа	2	0	100%	100%	0%
Квадрат числа	1	0	100%	100%	0%
Проверка на ччтность	1	0	100%	100%	0%
Искусственная функция с множеством ветвлений	7	2	100%	80%	29%
Пример с избыточными тестами	4	3	100%	75%	75%
Пример с двумя ветками `if` с одинаковыми условиями	3	2	100%	50%	67%
Пример с «мчртвым» кодом	2	1	100%	100%	50%

данный метод. Разработанный прототип был опробован на наборе простых тестовых программ и показал удовлетворительные характеристики, такие как тестовое покрытие и избыточность.

В настоящее время разработанный прототип не поддерживает работу с памятью, внутрипроцедурные эффекты, происходящие в результате тестирования, и использует очень простые алгоритмы уменьшения избыточности.

В будущем планируется убрать существующие ограничения путчм разработки алгоритмов моделирования памяти и провести исследования по минимизации наборов тестовых случаев.

Список литературы

1. Marat Akhin, Mikhail Belyaev, and Vladimir Itsykson. Yet another defect detection: Combining bounded model checking and code contracts // PSSV'13. 2013. P. 1–11.

2. Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers // *Int. J. Softw. Tools Technol. Transf.* 2009. 11(1). P. 69–83.

3. Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto // *ACSL: ANSI/ISO C Specification Language. Preliminary Design, version 1.4, 2008.*, preliminary edition, may 2008.

4. Kent Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

5. Boris Beizer. *Software testing techniques.* Dreamtech Press, 2003.

6. Dirk Beyer. Competition on software verification // TACAS'12. Springer, 2012. P. 504–524.

7. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs // TACAS'9. 1999. P. 193–207.

8. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs // TACAS'04. 2004. P. 168–176.

9. David M Cohen, Siddhartha R Dalal, Jesse Parelius, and Gardner C Patton. The combinatorial design approach to automatic test generation // *IEEE software.* 1996. 13(5). P. 83–88.

10. Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software // ASE'09. 2009. P. 137–148.

11. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. 2008. P. 337–340.

12. RA DeMilli and A. Jefferson Offutt. Constraint-based automatic test data generation // *Software Engineering, IEEE Transactions on.* 1991. 17(9). P. 900–910.

13. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing // *ACM Sigplan Notices.* ACM, 2005. Vol. 40. P. 213–223.

14. Richard Hamlet. Random testing // *Encyclopedia of software Engineering*, 1994.

15. Franjo Ivančić and Sriram Sankaranarayanan. NECLA static analysis benchmarks.

16. Anil Kumar and J St Clair. Cunit-a unit testing framework for c // *Diposnıvel em: http://cunit. sourceforge. net/doc/index. html. Acesso em*, 25, 2005.

17. Chris Lattner. LLVM and Clang: Next generation compiler technology // *The BSD Conference*, 2008.

18. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation // CGO'04. 2004. P. 75–86.

19. Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR // VSTTE'12. 2012. P. 146–161.

20. Bertrand Meyer. Applying 'design by contract' // *Computer.* 1992. 25(10). P. 40–51.

21. Glenford J Myers, Corey Sandler, and Tom Badgett // *The art of software testing.* John Wiley & Sons, 2011.

22. Clementine Nebut, Franck Fleurey, Yves Le Traon, and J-M Jezequel. Automatic test generation: A use case driven approach // *Software Engineering, IEEE Transactions on.* 2006. 32(3). P. 140–155.

23. Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation // *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007. P. 75–84.

24. Koushik Sen, Darko Marinov, and Gul Agha. *CUTE: a concolic unit testing engine for C* // ACM, 2005. Vol. 30.

25. Nikolai Tillmann and Jonathan De Halleux. Pex—white box test generation for .Net // *Tests and Proofs*. Springer, 2008. P. 134–153.

26. Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution // IEEE, 2009. P. 359–368.

Using a Bounded Model Checker for Test Generation: How to Kill Two Birds with One SMT-solver

Maxim Petrov, Kirill Gagarski, Mikhail Belyaev, Vladimir Itsykson

Saint-Petersburg State Polytechnic University
Polytechnicheskaya street, 29, Saint-Petersburg 195251 Russia

Keywords: automated test generation, dynamic symbolic execution, bounded model checking, satisfiability modulo theories, function contracts

The automated test generation has received a lot of attention in the last decades as it is one of possible solutions to software testing inherent problems: the need to write tests and providing test coverage in presence of human factor. The most promising technique of generating a test automatically is the dynamic symbolic execution assisted by an automated constraint solver, e. g., an SMT-solver. This process is very similar to the bounded model checking, which also has to deal with generating models from a source code, asserting logic properties in it and process the returned model. This paper describes a prototype unit test generator for C based on a working bounded model checker called Borealis and shows that these two techniques are very similar and can be easily implemented by using the same basic components. The prototype test generator was sampled on a number of examples and showed good results in terms of test coverage and test excessiveness.

Сведения об авторах:
Петров Максим Алексеевич, СПбГПУ, аспирант;
Гагарский Кирилл Алексеевич, СПбГПУ, исследователь;
Беляев Михаил Анатольевич, СПбГПУ, аспирант;
Ицыксон Владимир Михайлович, СПбГПУ, доцент