УДК 004.415.532.3

Ускорение направленного автоматического тестирования ПО в практике моделирования СБИС за счет сокращения обходов ветвей условных переходов

А.С. Щербаков

ЗАО «ИНТЕЛ А/О», andreas.s.scherbakov@intel.com

***Аннотация* — Метод направленного перебора, основанный на решении уравнений относительно условий переходов (DART) все шире применяется при тестировании программного обеспечения (ПО). Его существенным недостатком является большое время тестирования в практических приложениях из-за экспоненциального числа итераций. В работе описан эффективный метод сокращения перебора путей без сложного анализа программы, основанный на ограничении повторных проходов ветвей операторов условных переходов.**

***Ключевые слова* — тестирование программ, верификация программ, динамическая верификация, покрытие операторов, покрытие кода, встроенные программы, DART, автоматическое тестирование.**

I. Введение

Ряд современных цифровых устройств содержит в себе не только схемотехнику, но также и программное обеспечение (часто называемое Firmware). Чтобы гарантировать работоспособность таких устройств, требуется верификация такого ПО в комплексе с моделью самого устройства. В итоге речь идет о верификации модели, включающей в себя описание поведения схемы и ПО. Такая задача методологически более близка к проверке ПО, нежели к традиционным методам верификации электронных схем, т.к. множество состояний и переходов, возможных в комбинаторно-триггерной логике и элементах памяти, достаточно просто описывается программным кодом. (Заметим, что обратное в общем случае неверно: работу программы невозможно описать логической схемой ограниченного размера). Это создает потребность в широком применении средств верификации ПО при проектировании устройств. Тем более, что на ранних стадиях такого проектирования часто сами устройства представлены лишь в виде абстрактных поведенческих моделей, являющимися (с некоторыми ограничениями) обычными программами. Схожие потребности возникают и при тестировании готовых устройств аппаратно-программными комплексами (включая тестирование самих алгоритмов таких комплексов).

В итоге методы и средства верификации ПО становятся значимой частью процесса создания сложных цифровых схем, в особенности процессоров и периферии, контроллеров, однокристальных систем и т.п. В данной работе рассматриваются эксперименты по оптимизации алгоритма работы системы динамической верификации программ, разрабатываемой и поддерживаемой коллегами и автором. Хотя данная система в целом является универсальным средством тестирования ПО, ряд специально разработанных возможностей, опционально доступных в системе, обеспечил ее эффективное применение для проверки функционирования проектов “на кремнии”. Следует упомянуть тот факт, что сюда входит широкий диапазон от проверки сложных абстрагированных инвариантных свойств (например, когерентности кэша, целостности данных и др.) до моделирования процесса контроля готовой продукции на фабрике.

Речь в основном пойдет об оптимальных стратегиях тестирования, пригодных для быстрого и полного перебора возможных путей, где могут быть обнаружены ошибки в работе устройства.

II. Алгоритм работы системы тестирования

Алгоритм работы применяемой системы основан на идее тестирования программы путем ее повторяющегося исполнения при различных наборах входных значений с применением технологии Directed Automated Random Testing (DART) [1]. DART (по возможности) обеспечивает выполнение тестируемой программы существенно различными путями от запуска к запуску. «Существенно различными путями» считаются пути, где хотя бы для одного условного перехода управления посещались различные ветви, а также пути, ведущие к нештатным ситуациям (ошибкам) по отношению к успешным путям выполнения. В работе алгоритма ключевым моментом является целенаправленное достижение обхода ветвей условных переходов с помощью решения системы уравнений для условия перехода с учетом присвоений и вычислений, наблюдаемых в предыдущем запуске программы. Для этого используются трассировка вычислений с помощью дополнительных операторов, автоматически добавляемых в исходный код в процессе инструментирования, с последующим формированием системы логических и арифметических уравнений, включающей желаемые

значения условий для условных переходов и различные ограничения модели.

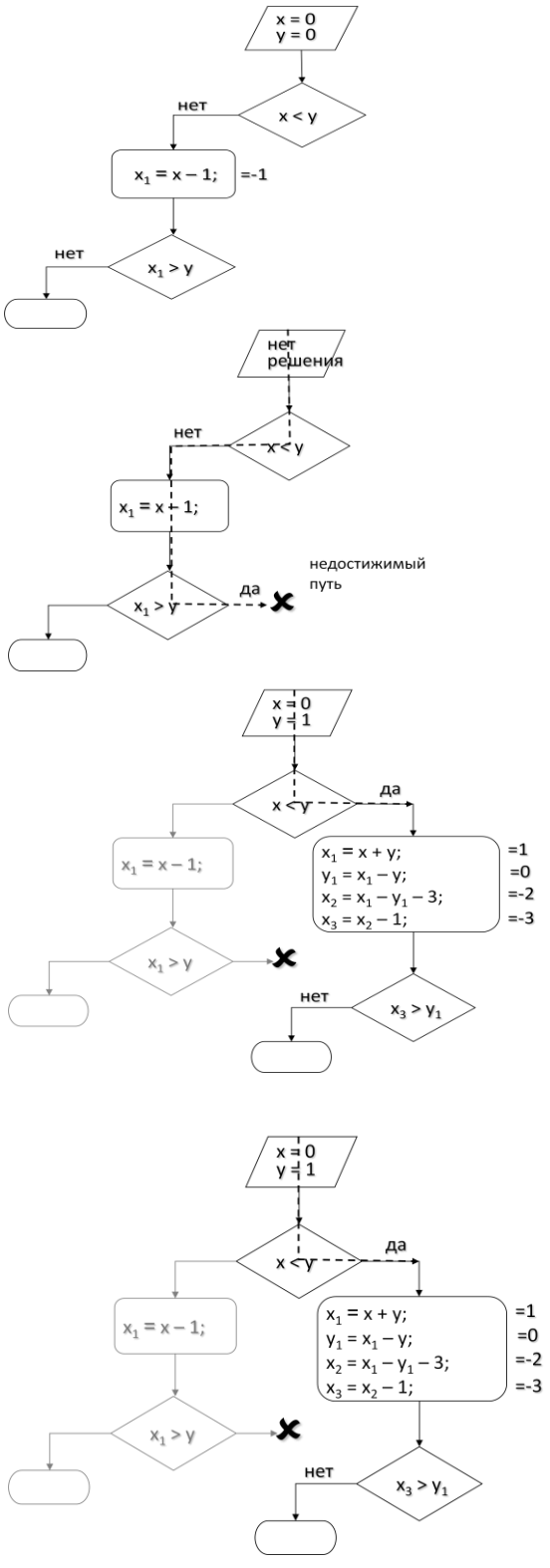

Рис. 1. Последовательные пути выполнения тестируемой программы в примере. Целенаправленно выбираемый префикс пути выполнения отмечен пунктиром

Решение полученных систем уравнений обеспечивается универсальными программами-решателями. В качестве последних применяются Satisfability Module Theory solvers (SMT solvers) [3, 4], поддерживающих входной интерфейс SMT-LIB [5].

На рис. 1 показан пример из 4 последовательных выполнений нижеследующей программы, в результате которых обнаружены значения входных переменных *x* и *y*, приводящие к ошибке:

if x > y then {x := x + y; y := x – y; x := x – y – 3;}
if x>y then error;

В случае, если отслеживание выполненных операторов невозможно (в контексте конуса зависимости какого-либо условия), входные воздействия выбираются случайным образом. Однако данные о зависимостях выходов от входов записываются, накапливая знание о свойствах участков кода, недоступных непосредственному анализу. Такие записи становятся ограничениями модели: в случае, если вновь встречаются известные входные воздействия, значения выходов берутся из истории. Это предполагает повторяемость исполнения программы (non-volatility), что накладывает существенные ограничения в работе, например, с многопоточными приложениями, однако позволяет эффективно использовать подобный механизм обучения для сокращения излишних итераций.

К сожалению, объем статьи не позволяет полностью описать алгоритм работы средства. Общие принципы лежащего в его основе алгоритма изложены в [1, 2].

III. Проблемы выбора путей выполнения

Как видно из описания алгоритма, средство способно исполнять программу так, чтобы каждый раз путь ее выполнения существенно отличался от предыдущих, минимизируя неинформативные проходы по известным путям за счет подбора интересующих значений с помощью SMT Solvers. Однако количество даже существенно различных путей, по которым может исполняться программа, все же слишком велико в большинстве реальных примеров. Дело в том, что в ряде случаев каждая из последовательных точек ветвления может быть пройдена по любой из ветвей независимо от того, какие ветви выбраны в других точках. Рассмотрим простой пример:

```
if x1 < 0 then y := y + 2^0;
if x2 < 0 then y := y + 2^1;
……
if x10 < 0 then y := y + 2^9;
z: = 1000000 / (y  - 363);
```

Здесь возникает 2^{10} = 1024 пути выполнения, приводящие к различным значениям *y* в диапазоне от 0 до 1023. В этом примере исключение каких-либо путей из рассмотрения привело бы к тому, что не все различные возможные значения *y* были бы учтены. Если присвоение у какого-либо из значений приводит

в дальнейшем к ошибке (как здесь: при $y = 363$ – деление на ноль), то мы могли бы упустить возможность выявить эту ошибку; грубо говоря, вероятность ее выявления равна отношению количества рассмотренных путей к числу всех возможных путей.

К счастью, это не всегда так. Большую часть точек ветвления можно считать независимыми. Например:

```
If x1 < 0 then y1 = 1 else y1 = 0;
if x2 < 0 then y2 = 1 else y1 = 0 ;
…..
if x10 < 0 then y10  = 1 else y10 = 0 ;

if y1 > 0 then error;
if y2 > 0 then error;
…..
if y10 > 0 then error;
```

Нет необходимости в переборе всех комбинаций условий для присвоений x_i, так как они влияют на результат (появление ошибки) независимо – мы можем ограничиться лишь двумя итерациями, например, «все условия для x_i истинны» и «все условия для x_i ложны».

С помощью анализа присвоений и использований данных в зависимости от прохождения точек ветвления можно понять, какие комбинации можно проигнорировать при переборе. Такой анализ описан, например, в [6]. Однако, когда дело касается больших программ, это требует существенных ресурсов и далеко не всегда оправдано. В реальных кодах могут быть сотни, тысячи ошибок или нештатных ситуаций, подавляющее большинство из которых выявляются сравнительно несложными вариациями входных значений, и лишь небольшая часть требует поиска весьма необычных комбинаций; поскольку на практике на отладку серьезных программ уходят месяцы, и при этом исходный код постоянно изменяется, направлять существенные усилия на оптимизацию анализа комбинаций оказывается менее актуально, чем на быстрый перебор критически важных путей. Поэтому (хотя анализ данных и зависимостей также находит применение в нашем средстве) в данной работе основное внимание уделяется более простым и быстрым стратегиям поиска путей.

IV. Сокращение повторных обходов

Базовая идея, лежащая в основе предлагаемого алгоритма, предельно проста. Если обе ветви какого-либо условия были задействованы в каких-либо из предыдущих тестовых выполнений программы, их повторное прохождение считается “менее интересным”.

В связи с этим мы должны ввести некоторые понятия.

Мы видели, что после каждого исполнения тестируемой программы алгоритм тестирования принимает решение либо о посещении альтернативной (по отношению к посещенной в последнем выполнении) ветви одного из условных переходов в программе. **Повторным обходом** [ветвей] мы будем называть принятие решения о посещении такой альтернативной ветви в случае, если когда-либо ранее (в рамках данного тестирования) обе ветви уже посещались.

Числом повторных обходов (ветвей) мы будем называть число условий в текущем пути исполнении программы, к которым было применено указанное выше решение о повторном обходе. Путь исполнения каждого следующего запуска программы в норме наследует префикс от пути исполнения предыдущего запуска. За ним следует (целенаправленный) обход ветвей – проход по альтернативной (по отношению к предыдущему запуску) ветви какого-либо условного перехода, затем, возможно, следует суффикс – следующие за выбранной ветвью условные переходы. При этом условные переходы в суффиксе никак не увеличивают число повторных обходов, которое, таким образом, в каждом следующем запуске в максимуме равно этому показателю в предыдущем запуске плюс 1, а в минимуме – нулю.

Вернемся к алгоритму сокращения числа обходов. В самом простом варианте повторное прохождение вообще не допускается. При этом алгоритм работает, конечно, быстро, но вероятность найти нестандартную ситуацию мала. Однако, если мы разрешим присутствие двух, трех и более повторных обходов, время работы алгоритма будет расти, как и шансы найти ошибку (полнота). Можно ли оптимальным выбором разрешенного числа обходов и алгоритма таких обходов получить существенную оптимизацию скорости тестирования программ без серьезной потери полноты тестирования? Исследования, проведенные автором, показали, что в большинстве практических примеров (взятых из программ тестирования готовых вычислительных узлов и из моделирующих кодов архитектуры СБИС) покрытие 95-100% принципиально выполнимых ветвей кода достигается уже при достаточно небольших значениях числа разрешенных повторных обходов. Поэтому такая оптимизация возможна, что позволило усовершенствовать средство верификации программ и стало основой дальнейших экспериментов со стратегией распределения разрешенных обходов.

Для измерения полноты покрытия ветвлений исходного кода применялись как счетчики, встроенные в тестирующее средство, так и внешний инструмент (gcov [7]), для подключения которого была добавлена возможность воспроизведения тестовой последовательности в независимой среде. В качестве тестовых примеров применялись блоки реальных программ, моделирующих тестирование цифровых схем.

V. Оптимизация распределения обходов

Возникает вопрос: можно ли обеспечить выбор путей выполнения программы без анализа зависимостей данных, обеспечивающий достаточно

полное покрытие возможных ситуаций? Чтобы ответить на этот вопрос, было проведено (на том же наборе примеров) сравнение ожидаемого времени выявления ошибок или достижения заданного покрытия операторов при использовании различных алгоритмов итерирования ветвей условных переходов (мы называем каждый из таких алгоритмов «стратегией поиска»). При этом предполагалось, что пользователь самостоятельно (опытным путем с учетом стадии отладки программы и имеющегося бюджета времени) выбирает некий параметр - верхнюю границу числа повторных обходов (re-alternation bound), влияющий на общее число тестов с различными путями выполнения. Задачей сравнения стратегий поиска было нахождение стратегии, которая обеспечивает оптимальное (в среднем) распределение заданного ресурса – числа путей исполнения (которое в достаточной степени пропорционально времени тестирования). Для обеспечения сравнимости результатов предполагалось, что определяющим детальность поиска параметром являлась не сама “граница числа повторных обходов” (которая может иметь различную нормировку в разных стратегиях), а соответствующее ей время выполнения теста (в которое, как описано выше, входит время вызовов тестируемой программы, время вызовов SMT solver и время работы ведущей программы; впрочем, последнее в нашем случае сравнительно мало).

В первоначальных экспериментах использовалось случайное распределение «квоты» **Q** разрешенных повторных обходов ветвей среди активных в данный момент условий. Активными условиями в данном контексте являются все встреченные на предыдущем запуске тестируемой программы условия, для которых обе ветви уже посещались в более ранних запусках в рамках данной последовательности тестов. Алгоритмически эта идея реализована следующим образом. Пусть мы имеем множество условий в последнем запуске программы, из них для **N** условий обе ветви уже посещались. Сначала рассматривается наиболее глубокое условие (DFS-поиск). Если какая-либо из его ветвей еще не посещалась, мы всегда производим попытку обхода ветвей (это верно для всех описанных ниже подходов). Если же обе ветви посещались ранее, мы случайным образом с вероятностью $\mathbf{a}\cdot\frac{Q}{N}$ (максимум 1, в том числе при **N**=0; **a** – норма) принимаем решение о попытке посещения другой ветви данного условия. Иначе, если решение не принято, мы переходим к предыдущему условию (вверх по графу ветвлений).

Альтернативные методы распределения повторных обходов предполагали их концентрацию в верхних или нижних частях графа ветвлений. Алгоритм концентрации в верхней части предполагает, что, если в последнем запуске программы встречено **N** активных условий, то из них выбирается не более **Q** последовательных верхних в дереве для повторного обхода. Этот подход показал менее приемлемые результаты, чем случайное распределение.

Напротив, выбирая условия для повторного обхода ветвей в нижней части дерева выполнения, в подавляющем большинстве случаев удавалось достичь ускорения поиска ошибок в программе и покрытия операторов. Хотя, конечно, мы можем легко привести специально подготовленный контрпример, в реальных тестовых примерах такой метод практически всегда «побеждает» случайное распределение. Здесь играет роль то, что в реальных программах нестандартные ситуации, приводящие к ошибкам, обычно формируются в результате комбинирования переходов управления в точках, локализованных достаточно компактно; вот почему группирование близко расположенных условий для перебора всех вариантов более эффективно, чем группирование отдаленных условий: тем не менее, для детального поиска нестандартных ситуаций в программе мы должны применять и тесты с группированием условий, расположенных в удаленных частях кода (случайно или с применением оценки их зависимости).

На рис. 2 показано нарастание покрытия тестом операторов ветвлений исходного кода с временем в одном из примеров (без ограничения повторных обходов). Видно, что затраты времени экспоненциальны по отношению к покрытию; за 3 часа работы удается покрыть только около 70% кода. На рис. 3 – такие же графики при различных границах числа повторных обходов (в линейном масштабе; время измерения ограничено 2500-2700 с). Полнота покрытия в 94,5% достигается всего за 2630 с при границе 15. В самом «быстром» тесте полнота в 93,2% достигается менее чем за полминуты.

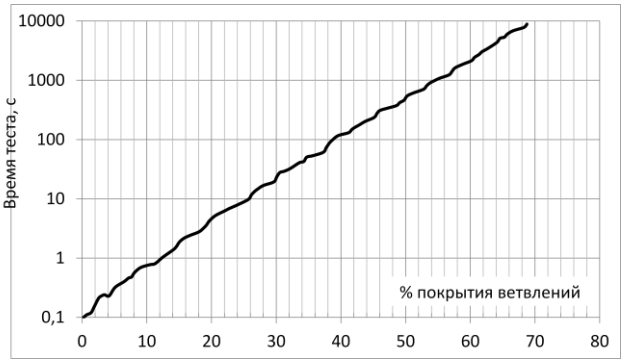

Рис. 2. Пример зависимости времени тестирования от полноты покрытия ветвлений в алгоритме без ограничения числа обходов ветвей

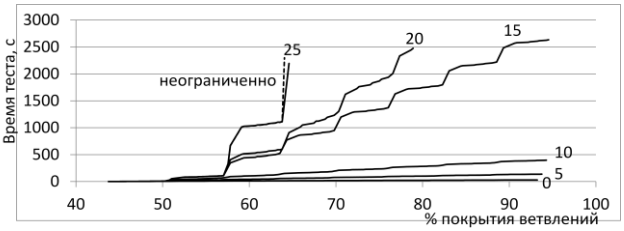

Рис. 3. Примеры зависимости времени тестирования от полноты покрытия ветвлений при различных значениях разрешенного числа повторных обходов

VI. Влияние размера циклов в программе на эффективность метода

Особенно хорошие результаты метод показывает при тестировании программ для моделирования электронных схем или “встроенного” в них ПО. Отличительной особенностью таких программ является преимущественно ациклическое исполнение (здесь нужно оговориться, что циклы, хотя и встречаются в коде, как правило, имеют фиксированное либо ограниченное (достаточно малым числом) число итераций, т.е. они могут быть раскрыты (синтезированы) в набор операторов без циклов). Если тело цикла содержит оператор условного перехода, то перебор всех путей выполнения **M**..**N** итераций цикла потребует **N**-**M** вложенных обходов ветвей такого оператора (завершить; продолжить, завершить; продолжить, продолжить, завершить и т.д.) - при этом путь выполнения может содержать до (**N**-**M**-1) повторных обходов. В данном случае выбор максимального числа разрешенных повторных обходов фактически регулирует разрешенную границу «подробно исследуемых» итераций цикла. В ряде случаев известно, что выполнение более **k** итераций цикла не может породить новых ситуаций по сравнению с полученными при выполнении от 0 до **k** итераций; в таких случаях, выбором соответствующей границы для числа повторных обходов можно существенно ускорить тестирование без потери полноты.

Важно заметить, что ограничение повторных обходов в данном контексте не означает ограничение числа итераций цикла, поэтому, если ошибка в программе, не обусловленная специфическим выбором комбинаций ветвей условного перехода в теле цикла в последовательных итерациях, возникает только при числе итераций, большем **m**, она будет все же обнаружена. Рассмотрим следующий пример (q – свободная входная переменная):

```
for i := 1 to 5000
  if i > 100 and i=q then error;
```

Если бы мы искусственно ограничили число итераций, например тремя, понятно, что обнаружить ошибку *error* было бы невозможно, т.к. она может возникнуть, начиная с 100-й итерации. Однако ограничение повторных обходов никак не влияет на ее обнаружение, т.к. уже первый обход ветвей условия «i > 100 and i=q» ведет к передаче управления к *error*.

В другом примере, однако, ошибка при тех же условиях может остаться необнаруженной:

```
s:=0;
for i=1 to 5000
  if q>i then s:=s+1;
if (s=100) then error;
```

Действительно, при максимальном числе повторных обходов, равном 3, мы можем получить тесты со следующими значениями q: 0; 5001; 5000; 4999; 4998 (первое выбрано по умолчанию; затем - обход самых “дальних” ветвлений – поиск в глубину). Теперь в пути выполнения уже есть 3 повторных обхода условия «q>i» (при i=4998, i=4999, i=5000), поэтому дальнейшие попытки добиться обходов ветвей этого условия при i=0...4997 не предпринимаются. Но для передачи управления к *error* необходимо, чтобы для q было выбрано значение 101, что в нашем случае не достигается.

VII. Учет контекста вызовов

Еще одним случаем, когда эффект подхода может снижаться, является вызов сложных блоков в разных контекстах. Простейшим примером являются арифметические или тригонометрические функции, которые могут быть использованы в самых разных по функциональности блоках кода в рамках одной тестируемой программы. Интуитивно понятно, что история выполнения таких функций при вызове из одного блока никак не релевантна решениям, принимаемым по поводу локального тестирования другого блока. Исходя из этого, появилась идея разделить учет повторных обходов для вызовов функций в разных контекстах.

При этом все данные о посещении потоком управления точек ветвления собираются и хранятся отдельно для каждого уникального стека вызовов функций.

Пусть, например, функция F содержит в теле условие *c* . Данная функция может вызываться из функции *A* либо из функции *B*. Тогда предлагаемый порядок учета обходов можно выразить в следующей эквивалентной формулировке. Создадим *F’* - копию функции *F*, содержащую, соответственно, в теле *c’* – копию условия *c*, и заменим все вызовы *F* из *B* на вызовы *F’*, а вызовы *F* из *A* оставим без изменений. Таким образом, условие *c*, выполняемое в разных контекстах (*A*→*F* и B→F), распадается на разные (с точки зрения учета обходов ветвей условий) условия *c* и *c’*, что обеспечивает независимость решений о допустимости повторных обходов в различных контекстах.

На самом деле, конечно, копирования функций не производится, а добавляются связанные со стеком вызовов ключи при идентификации точек ветвлений так, что одна и та же точка ветвления рассматривается в различных контекстах как различные точки ветвления для вышеописанных алгоритмов.

Данный подход кардинально снижает риск необнаружения ошибок из-за «засорения» базы данных о повторных обходах побочными вызовами задействованных функций; однако он страдает существенным недостатком. Стек вызовов может иметь самые различные элементы, не связанные с вызовом *F* из *A* или *B*. Например, это может быть *D*→*E*→*F*→*A* или *G*→*H*→*F*→*A*. Если функции *D, E, G, H* не влияют на рассматриваемые нами условные переходы и их последствия, такие стеки следовало бы считать эквивалентными *A*→*F*. В противном случае различие префиксов стека приводит к тому, что

повторные посещения потоком управления интересующего нас “суффикса” стека ...→*A*→*F* рассматриваются как разные контексты. В результате в большом количестве примеров повторные обходы практически не идентифицируются, что сводит на нет оптимизацию производительности за счет ограничения числа таких обходов.

В более редких случаях такой же эффект связан с появлением различных нерелевантных элементов стека между “интересными”, например если *F* вызывается из *A* или *B* не непосредственно, а через некие функции *K* или *L*, эффект которых для вычислений, происходящих в *A*, *B* и *F*, отсутствует или одинаков.

Исчерпывающий анализ эквивалентности контекстов потребовал бы рассмотрения всех возможных потоков данных и управления, что, как упоминалось выше, приводит к неоправданному в большинстве применений росту необходимых ресурсов. Для того чтобы несложным способом избежать «ложной» неэквивалентности контекстов, автором было предложено ограничивать размер рассматриваемого стека несколькими (самыми внутренними) элементами. Тестирование реальных примеров показывает, что выбор такого числа в диапазоне 3...5 обеспечивает адекватный учет ветвлений в различных контекстах и, соответственно, позволяет полностью реализовать преимущества метода ограничения числа повторных обходов.

VIII. Контексты ветвлений: дальнейшая детализация

Упомянутый метод выделения ветвлений, происходящих в различных контекстах, в самостоятельные единицы для учета повторных обходов, можно расширить и за рамки разделения по стеку вызова функций. Ветвления, происходящие повторно в рамках одного и того же стека, также можно относить по тем или иным признакам к разным контекстам, что при удачном алгоритме выделения обеспечивает отнесение независимых ветвлений к одному контексту, а зависимых – к разным. Это дало бы возможность уменьшить разрешенное число повторных обходов (в идеале – до нуля) без потери полноты покрытия кода, т.е. обеспечило бы дальнейшее ускорение тестирования программы. Во время написания данной статьи автор проводит эксперименты по эффективным алгоритмам отождествления контекстов ветвления. Однако даже в “ручном” режиме (когда пользователь может помечать различные условия как независимые или, наоборот, каждое повторение одного и того же условия при выполнении программы - как зависимые условия) полнота тестирования легко может быть доведена почти до 100% при небольшом объеме работы. Как более удобный вариант, реализуется возможность помечать переменные, через присвоение которых осуществляется зависимость между условиями.

IX. Заключение

Метод тестирования программного обеспечения с использованием метода целенаправленного выбора входных значений (DART) показал себя весьма универсальным и перспективным (особенно принимая во внимание бурное развитие приложений для решения уравнений – SMT solvers). Однако, как следствие экспоненциальной зависимости количества тестов от количества ветвлений программы, его применение для большинства серьезных практических примеров в настоящее время проблематично. Предлагаемые автором методы ограничения повторных обходов ветвей условий в тестируемой программе позволили кардинально улучшить практическую применимость метода, что показывают проведенные эксперименты и рост востребованности средства, реализующего данный подход, пользователями. Без применения ресурсоемких алгоритмов статического или динамического анализа потоков данных достигнуто многократное ускорение процесса тестирования без существенной потери его полноты. По степени использования в конкретном случае метод легко масштабируется, давая возможность наиболее рациональным образом использовать имеющиеся ресурсы. Также он хорошо комбинируется с другими алгоритмами сокращения перебора путей выполнения. Дальнейшее улучшение соотношения полноты тестирования к затраченному времени в рамках данного подхода возможно при его селективном применении на основе оценки зависимостей между условиями переходов в программе.

Литература

[1] Patrice Godefroid. DART: Directed Automated Random Testing (joint work with Nils Klarlund and Koushik Sen) Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation). Chicago, June 2005. P. 213-223.

[2] Patrice Godefroid. Compositional dynamic test generation. Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. New York, 2007.

[3] Nieuwenhuis R., Oliveras A., Tinelli C. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T) // Journal of the ACM. 2006. V. 53. P. 937–977.

[4] Fränzle M., Herde C., Ratschan S., Schubert T., Teige T. Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure // JSAT Special Issue on SAT/CP Integration. 2007. P. 209–236.

[5] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard - Version 2.0 // Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England). 2010.

[6] Rupak Majumdar and Ru-Gang Xu. Reducing test generation with information partitions // Lecture Notes in Computer Science. 2009. Volume 5643/2009. P. 555-569. DOI: 10.1007/978-3-642-02658-4_41.

[7] Holger Blasum, Frank Görgen, Jürgen Urban. Gcov on an embedded system // Workshop: GCC for Research in Embedded and Parallel Systems. Brasov. 16 Sept, 2007.