УДК 004.415.532.3

Ускорение направленного тестирования встроенного и внешнего ПО СБИС путем учета потока данных при ограничении вариативности траекторий выполнения

А.С. Щербаков

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

***Аннотация* — Метод направленного перебора, основанный на решении уравнений относительно условий переходов (DART), широко применяется при тестировании 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 обнаружении изменений и зависимостей символических выражений, составляющих условия переходов, в комбинации с ограничением числа различий в траектории выполнения программы.**

***Ключевые слова* — тестирование программ, верификация программ, динамическая верификация, покрытие операторов, обеспечение качества ПО, покрытие кода, покрытие ветвлений, встроенные программы, DART, автоматическое тестирование, генерирование тестов.**

I. Введение

Встроенное программное обеспечение (Firmware) стало важной и постоянно растущей в объеме и сложности частью современных проектов микропроцессоров и систем на кристалле. Специфика обеспечения качества встроенного программного обеспечения требует, чтобы оно было на 100% свободно от ошибок, так же, как и аппаратная часть. Зачастую ошибки в таком ПО, обнаруженные в продукте, не могут быть исправлены на стороне потребителя, и единственной возможностью поправить ситуацию является отзыв и замена всего продукта. Поэтому встроенное ПО требует применения специальных методик обеспечения качества, которые могут полностью гарантировать отсутствие ошибок в коде. Существующие в обычной практике методики верификации программных продуктов не могут обеспечить требуемого качества для встроенного ПО.

Другая причина, которая заставляет искать возможности для инноваций в решениях для верификации программ, заключается в постоянно растущем размере самих программных приложений. Традиционные методы не могут обеспечить достаточной полноты и уровня качества верификации за разумное время. В результате проверка ПО занимает все более значительную часть жизненного цикла продукта, что угрожает его своевременному выходу на рынок. Эффективная автоматизация этого процесса стала насущной задачей индустрии программного обеспечения.

Эта статья основана на экспериментах, имевших целью оптимизировать алгоритм динамической верификации программ, который используется в средстве верификации программного обеспечения, разработку и поддержку которого осуществляли коллеги и автор. Хотя данное средство предназначено для широкого спектра тестируемых программ, осуществленные оптимизации позволили наиболее эффективно использовать его в аппаратных проектах для тестирования встроенного программного обеспечения.

II. Обзор средства тестирования

В данной работе представлены идеи, использованные при усовершенствовании внутрикорпоративного опытно-экспериментального средства для тестирования модулей ПО с кодом на языках C и C++. В основе его работы лежит алгоритм направленного поиска путей исполнения (Directed Automated Random testing – DART) [1,2]. Тестируемая программа запускается множество раз с различным набором входных данных, для передачи которых служит специальный интерфейс (harness). Суть алгоритма заключается в том, чтобы по возможности заставить программу в каждом запуске теста выполняться разными путями. Под путем исполнения мы понимаем последовательность ветвей операторов условных переходов (включая выходы из итерации циклов и т.д.), выбранных при выполнении программы. Чтобы достичь этого, после каждого запуска составляется система уравнений, удовлетворение каждого из которых соответствует выбору запланированной ветви соответствующего оператора условного перехода. Уравнения составляются в базисе входных данных программы с помощью отслеживания присвоений переменных в символическом виде, производимого специальными инструментирующими вставками кода в теле тестируемой программы. Заметим, что такое

инструментирование обеспечивает комбинированное двойное выполнение программы (символическое и конкретное – *concolic*, от *concrete + symbolic*).

В базовом варианте для обеспечения полноты тестирования при итерировании путей выполнения происходит перебор всех возможных путей поиском в глубину. Учитывая, что количество итераций в таком алгоритме может быть экспоненциально по отношению к количеству условных переходов в пути выполнения (а само это число может быть неограниченным, как в цикле без статического ограничения максимальной границы числа итераций), понятно, что число тестирований может быть слишком велико для обеспечения полного тестирования таким путем при имеющихся вычислительных ресурсах. Поэтому постоянно происходит разработка и оптимизация различных стратегий, позволяющих обеспечивать приемлемую полноту тестирования без обхода большей части возможных путей выполнения тестируемой программы. Такова цель и у рассматриваемой в данной работе кардинальной оптимизации стратегии планирования путей.

III. Вводные замечания и определения

A. *Линейные участки и условные переходы*

Для удобства изложения и реализации алгоритма мы разбиваем (без потери общности) тестируемую программу на линейные участки (не содержащие условных переходов фрагменты кода) и условные переходы. Линейный участок также не содержит в себе целевых адресов переходов. Таким образом, линейный участок может быть выполнен только целиком.

B. *Линейные участки с присваиванием переменных*

Каждый оператор присваивания тестируемой программы инструментируется соответствующим кодом, генерирующим символическое уравнение (в базисе входных переменных теста), позволяющее проследить все вычисленные выражения, входящие в результат присваивания. Этот механизм был дополнен ссылками на участки программы (через позицию в исходном коде). Таким образом, для каждого выражения (в частности, для условия какого-либо условного перехода) мы знаем минимальную (в узком смысле, т.е. без учета сути выражений) последовательность ветвей программы, прохождение которой в данном порядке потоком управления и привело к данному символическому значению нашего условия перехода. Если в дальнейшем мы выберем путь выполнения, в котором последовательность участков с присваиваниями для условия перехода такая же самая, мы получим то же символическое значение условия. (Это, однако, не означает численного равенства, т.к. значения входных переменных могли измениться).

C. *Предмет оптимизации*

По сути, алгоритм DART преследует цель обхода всех ветвей каждого условно перехода, при этом решение для инвертирования конкретного условия ищется как в рамках обнаруженного для него символического выражения (решением уравнения с помощью SAT Solver), так и попыткой прийти к тому же переходу путем обхода предшествующих ветвей в другом порядке, т.е. путем построения другой последовательности присваиваний для данного условия. В данной работе в фокусе рассмотрения находится именно вторая тактика. «Традиционный» DART производит обход всех комбинаций ветвей. В предлагаемом методе производятся только такие обходы предшествующих ветвей, которые могут привести к новым последовательностям участков с присваиванием для интересующих условий перехода.

D. *Планирование пути*

Алгоритм тестирования DART в каждой итерации сначала задается целью исполнить программу так, чтобы последовательные ветви условных переходов выполнялись определенным образом (во всяком случае, до некоторой заданной глубины). Мы будем называть вычисление такого желаемого пути выполнения программы *планированием пути*. В классическом DART каждый следующий путь планируется исходя из предыдущего путем изменения (инвертирования) направления одного из условных переходов при сохранении направления всех предшествующих. Это приводит к покрытию только одной комбинации условных переходов за запуск программы, т.е. к необходимости перебора всех комбинаций с запуском для каждой комбинации, если мы добиваемся полного покрытия. В отличие от такого полного перебора, в предлагаемом подходе пути планируются путем анализа доступности узлов, посещение которых потоком управления потенциально увеличивает покрытие тестируемого кода.

E. *Интересные участки и условия*

Целью планирования путей фактически является достижение максимального покрытия ветвей кода. Поэтому условные переходы, обе ветви которых были посещены при тестировании, не представляли бы никакого интереса для повторного посещения, если бы такое повторное посещение потенциально не вело к присвоению новых символических значений переменным, составляющим условия переходов, не все ветви которых еще были посещены. Исходя из этого замечания, мы будем называть *интересными* условные переходы, имеющие еще не посещенную ветвь, и линейные участки, содержащие присваивания переменных, возможно, влияющие на условия *интересных* переходов.

IV. Предлагаемый алгоритм

В базовом варианте алгоритм перечисления планируемых путей выполнения схож с классическим

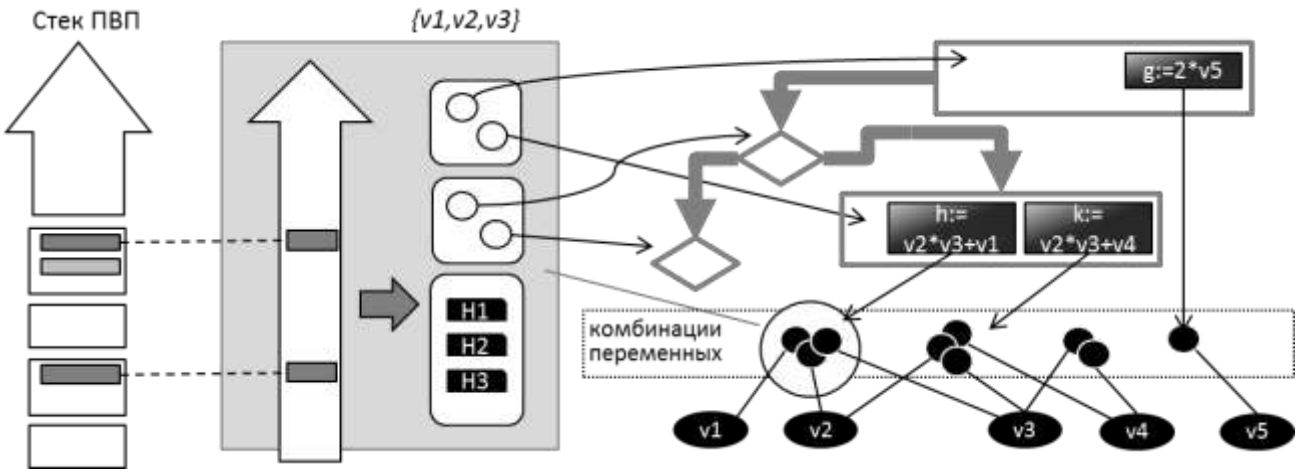

Рис. 1. Структуры, отображающие информацию о присвоении и использования данных в тестируемой программе

DART. Вначале тестируемая программа выполняется с произвольными входными данными. Затем делается попытка инвертировать значение условного перехода самого глубокого ветвления с помощью подбора новых входных данных путем решения уравнений для указанного условия. Если это удается, программа запускается с новыми данными, и мы повторяем итерацию. Если нет, мы пытаемся инвертировать условие предшествующего ветвления, поднимаясь по пути последнего выполнения до тех пор, пока условие не будет инвертировано. Когда ни одно условие нельзя инвертировать, процесс завершается. Заметим, что на практике этот процесс легко распараллеливается, что не влияет на суть происходящего.

Отличие предлагаемого алгоритма состоит в том, что попытка инвертирования условия в некотором узле происходит только при соблюдении дополнительного условия транспорта модифицированных данных. Чтобы отслеживать это условие, для каждого узла графа мы аккумулируем и храним связи переменных с позициями в коде (рис. 1). После каждого выполнения тестируемой программы для каждого символического представления интересных условий мы находим все входящие комбинации переменных программы и добавляем в таблицу ссылки от комбинаций переменных на участки кода и условия, где они были использованы. Также запоминаем ссылки от участков кода на присвоенные в процессе их выполнения комбинации переменных. Комбинации переменных формируются динамически из операндов, совместно участвующих в арифметико-логических выражениях.

Кроме того, для каждой участвовавшей комбинации переменных находим в стеке фрагментов программы подстек фрагментов, содержащих присвоения данным переменным, запоминаем уникальный идентификатор полученного подстека или хэш-функцию для последующего быстрого сравнения подстеков.

З а м е ч а н и е 1. Здесь и далее имеются в виду не входные переменные, а промежуточные присвоенные переменные программы (заметим, что выражения, тождественно равные какой-либо функции от “чистых” входных переменных, учитывать не обязательно, так как их символическое значение не зависит от пути выполнения программы).

З а м е ч а н и е 2. Идея использования подстеков основывается на том факте, что символическое значение переменной относительно входящих в нее входных и промежуточных переменных однозначно определяется последовательностью выполнения участков кода, содержащих ее присваивания (в предположении воспроизводимости (неволатильности) выполнения программы, на котором основаны методы DART).

З а м е ч а н и е 3. Мы называем последовательность исполненных фрагментов стеком, принимая во внимание стековый характер планирования путей в алгоритме DART, где планирование следующих путей происходит с использованием префикса, унаследованного из предыдущего пути. Исходя из удобства реализации алгоритма, подстеки для комбинаций переменных синхронизированы со стеком фрагментов программы через одновременный для всех релевантных стеков вызов процедур добавления и извлечения элементов.

З а м е ч а н и е 4. Для уменьшения числа запусков программы можно хранить также информацию о том, какие интересные условия зависят от данной комбинации переменных; когда условие перестает быть интересным (обе его ветви посещены), мы можем более не принимать во внимание новые варианты присвоений входящих в условие переменных. Однако такая оптимизация (требующая распространения значений по графу выполнения

программы) редко оправдана, так как число интересных условий, как правило, остается весьма большим до самого завершения теста (в результате наличия многочисленных условий корректности, добавленных при инструментировании кода, вторые ветви которых посещаются только в случае ошибок).

В нашем алгоритме, прежде чем запустить очередной прогон теста со следующим планируемым префиксом пути выполнения, необходимо убедиться в выполнении критерия полезности данного теста. Такой критерий считается удовлетворенным, если выполнено хотя бы одно из двух условий:

a) Предлагаемый префикс пути содержит новую (еще не посещенную) ветвь условного перехода.

b) Для какой-либо из комбинаций переменных найден новый подстек, и при этом хотя бы один из использующих ее участок кода (или условие перехода) достижим(о) на каком-либо пути выполнения, являющимся продолжением предлагаемого префикса.

Для обеспечения анализа достижимости участков используется граф потока управления, формируемый инструментирующим приложением; также возможно (и более предпочтительно, как дающее гарантированно достоверный результат) динамическое формирование наблюдаемого графа выполнения. (Также возможно дополнительное условие достижимости интересных условий, см. замечание 4). Для ускорения анализа достижимости рационально хранить матрицу инцидентности графа выполнения [4].

Если условие (b) выполнено для текущего префикса пути, мы дополнительно пытаемся уточнить путь выполнения, насколько это возможно, присоединяя к предлагаемому префиксу выполнения условные переходы в направлениях, по-прежнему ведущим к интересным участкам или условиям. Это предотвращает повторный DFS-обход уже исследованных условных переходов, не ведущих к использованию новых присвоенных символических значений переменной.

В табл. 1 показано действие алгоритма на примере простейшей программы, приведенной в виде диаграммы на рис. 2. Предполагается, что алгоритм DART выбирает для каждой входной переменной (a и b) вначале нулевое значение, то есть для нашего случая в порядке обхода графа переходов: (a,b) = (0,0); (0,1); (1,0); (1,1). В нашем примере при таком порядке обхода до предпоследней итерации неясно, что условие C3 может принимать символическое значение «c» вместо 0. Видно, как учет зависимостей по присвоениям переменных позволяет все же принять решение о необходимости выполнения теста при (a,b) = (1,1). Именно получение переменной x нового значения, еще не включенного в список хеш-функций ее подстеков, и доступность зависящего от него участка кода L2 по пути выполнения, начинающегося с префикса (C1=1; C2=1), служит признаком необходимости запуска теста по такому пути. После того, как в предпоследней итерации символическое значение C3=c выявлено и включено в систему уравнений для численного решения, Solver находит условие посещения ветви, ведущей к «ошибке»: c=3 (последняя итерация).

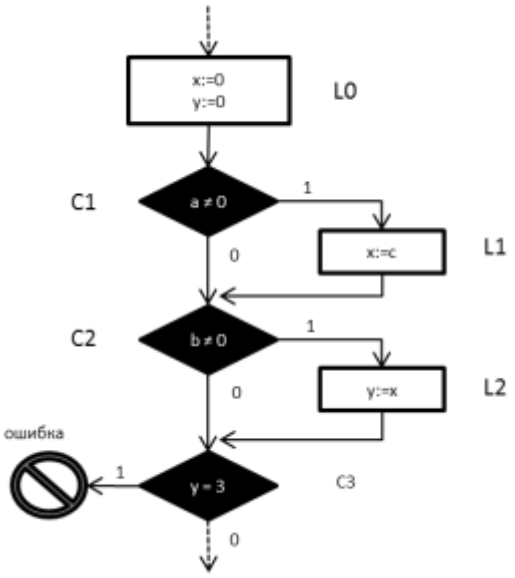

Рис. 2. Пример тестируемой программы

V. Ограничение числа повторных чередований ветвей

Метод выборочного запуска тестов, описанный в предыдущей главе, сам по себе редко помогает избежать экспоненциальной зависимости времени теста от размера программы, хотя и понижает показатель экспоненты в несколько раз. Чтобы ограничить экспоненциальный рост без значительной потери покрытия тестируемого кода, одновременно используется критерий, ограничивающий комбинации исследуемых повторных обходов ветвей условных переходов множеством комбинаций, отличающихся не более N значениями условий. Фактически, учет переменных позволяет увеличить N при сохранении числа тестов, что приводит к уменьшению непокрытого кода (статистически) в $\alpha^{(N-N0)}$ раз.

Для реализации критерия ведется подсчет планируемых ветвей условных переходов, отличающихся от «базовых», в стеке планируемых путей. Если достигнуто значение N, дальнейшее добавление условий к стеку запрещается [5].

VI. Переименование узлов

В практической программе позиция оператора в коде не всегда может однозначно идентифицировать его место в реальной последовательности выполнения операторов. Причиной этого являются циклы и вызовы функций. Например, в случае цикла фиксированной длины в 4 итерации соответствующий участок реального графа выполнения состоит из четырех

Таблица 1

Последовательный выбор путей выполнения в примере программы (Рис. 2)

№	Префикс пути выполнения	Решение о запуске теста (непосещенная ветвь/иная причина)	Входные переменные			Обнаруж. использования		Обнаруж. присваивания		Подстек присваивания	
			a	b	c	{x}	{y}	{x}	{y}	{x}	{y}
1	()	Да (начало)	0	0	0		C3	L0	L0	(L0)	(L0)
2	(C1=0;C2=0;C3=1)	Да (C3=1)	нет решения								
3	(C1=0; C2=1)	Да (C2=1)	0	1	0	L2			L2		(L0;L2) - новый
4	(C1=0;C2=1;C3=1)	Да (C3=1)	нет решения							(L0;L1)-новый	
5	(C1=1)	Да (C1=1)	1	0	0			L1			(L0)
6	(C1=1; C2=0;C3=1)	Да (C3=1)	нет решения								
7	(C1=1,C2=1)	Да (в L2 используется новый подстек для {x})	1	1	0						(L0;L2)
8	(C1=1,C2=1; C3=1)	Да (C3=1)	1	1	3						

последовательных фрагментов одного вида. Форма в виде циклического подграфа, которую можно найти при простом анализе исходного кода, слишком абстрактна. Например, если некоторая переменная z, изначально равная z0, увеличивается в каждой итерации цикла на 2, то после первой итерации она будет равна z0+2, затем z0+4 и т.д. Если мы оперируем циклическим представлением в графе, то мы все эти варианты значений отнесем к одному и тому же узлу (узлам). Тогда, если случится, что при другом префиксе пути выполнения на входе нашего цикла мы будем иметь z=z0+2, мы не сможем отличить его от предыдущего значения, полученного на второй итерации. В результате, мы не обнаружим появления новых символических значений переменных и не добавим к рассмотрению новые интересные узлы там, где это было бы в итоге рационально для поиска путей, ведущих к покрытию новых ветвей (т.е. не обнаружим возникнувшую неэквивалентность нового пути выполнения с предыдущими).

Чтобы избежать этого, мы переименовываем узлы для итераций цикла (как один из вариантов алгоритма) так, что разным итерациям цикла соответствуют разные узлы, а цикличность графа устраняется. Число рассматриваемых итераций может динамически варьироваться по мере тестирования (как в классическом DART) и ограничено сверху. Переименование узлов в циклах не всегда эффективно и может не давать заметного увеличения покрытия, но при достаточном объеме памяти его использование оправдано.

Еще более важно различать узлы, относящиеся к вызовам функции в различном контексте, так как значения переменных, обращающиеся в теле функции, вызванной из разных мест кода, могут быть никак не связаны. Переименовывая узлы для разных контекстов, мы фактически осуществляем inlining функции. Однако приходится учитывать и негативное влияние такого учета контекста, а именно то, что эквивалентные контексты могут оказаться формально различными. Пусть, например, в первом случае одинаковый блок кода с достаточно глубоким стеком вызовов вызывается в теле процедуры A; затем он вызывается из процедуры B. При этом с большой вероятностью можно утверждать, что большая часть кода, вызываемая в глубине стека, может быть рассмотрена как эквивалентная (т.е. поведение, обнаруженное в первом экземпляре, релевантно и второму). Строгий анализ таких ситуаций требует двойственного рассмотрения этих контекстов – как эквивалентных “изнутри” и неэквивалентных “снаружи” – что слишком затратно. Хорошие результаты дает упрощенный эвристический метод, когда контекст для переименования узлов идентифицируется ограниченным (не более 3 или 4) числом самых “глубоких” записей стека вызовов функций [5].

VII. Тестирование многониточного кода

Весьма полезным расширением метода является тестирование многопоточных (multithread) программ. В то время как многопоточность создает принципиальные трудности в традиционном DART подходе из-за неповторяемости пути выполнения программы и невозможности непосредственно рассматривать дерево путей выполнения, в нашем

случае моделирование многониточности достигается достаточно просто.

Рассмотрим простой, но распространенный пример многониточного кода, когда множество экземпляров некоторого блока операторов выполняется в параллельном режиме (параллельный цикл), что соответствует, например, директиве **parallel_for** из пакета Intel Threading Building Blocks (TBB). Мы можем представить такой фрагмент кода в виде обычного последовательного цикла, при этом линейные фрагменты и условия программы, соответствующие различным итерациям параллельного цикла, помечаем соответствующими уникальными идентификаторами. После завершения тестирования программы будем искать в накопленных записях зависимости через присвоение значений переменным между условиями или линейными участками, относящимися к различным итерациям. Если C(x) и I(x) – идентификаторы параллельного цикла и одной из его итераций, соответственно, обрамляющих участок (условие) x программы, F(c) – множество участков (условий) программы, зависящих от комбинации переменных c, A(f) – множество переменных, присваиваемых на участке f, то необходимо искать f и c, для которых

$$\exists f' : L(f') = L(f) \wedge I(f') \neq I(f) \wedge \exists\, c \in A(f) : f' \in F(c).$$

Такая задача решается обходом графа связей F и A. Список предупреждений об обнаруженных зависимостях предоставляется пользователю. Каждая из них сигнализирует о зависимости результата от порядка исполнения параллельного кода, которая должна быть устранена при отладке. Заметим, что в некоторых случаях (коммутативные операции, такие как константный инкремент, аккумулирование по «И» («ИЛИ») и другие) такая зависимость может допускаться.

VIII. Результаты

Описанный метод позволяет в несколько раз увеличить максимальное число рассматриваемых вариаций пути выполнения тестируемой программы за одинаковое время тестирования за счет избегания значительной (не менее 90% в большинстве случаев) части вариаций, не приводящих к существенному изменению вычисленных условий внутри программы. Алгоритм, в частности, применялся при отладке аппаратно-программного комплекса для тестирования готовых кристаллов процессоров. В программе моделировались последовательности выходных данных системы самотестирования. Ранее большую часть кода нужно было адаптировать вручную для обеспечения приемлемого времени теста, фактически дублируя зависимости данных вставкой инструкций для средства DART-верификации. Опыт показывает, что с использованием предлагаемого усовершенствования потребность в таких “State-of-Art” решениях отпадает (время одного теста всего в 1,5-3 раза больше при существенной экономии труда инженеров).

Не требуя в большинстве перебора комбинаций направлений переходов для составления картины зависимости данных, представленный алгоритм легко интегрируется с различными эвристиками, ограничивающими число тестов. Кроме того, по сравнению с методами, основанными на разбиении данных на разделы (например, [3]), он более подробно учитывает состав последовательностей присваивания (дерево в сравнении с многомерным кубом). Все это позволяет проводить сравнительно быстрое тестирование с числом повторных чередований условий переходов более 30, что для исследованных примеров Firmware программ и модульных тестов обеспечивает нахождение ошибок с вероятностью более 99% (ошибки добавлялись в случайные строки кода, равновероятно для всех строк в телах функций, было использовано 60 примеров кода).

Метод показал высокую эффективность для выявления ошибок в программах тестирования СБИС, встроенном ПО и программах обработки текста на уровне регулярных выражений. В то же время коды, содержащие сложные арифметико-логические вычисления или большие объемы взаимно зависящих данных, могут показывать результаты по времени тестирования, мало отличающееся от полученного при использовании базового алгоритма DART. Для таких случаев, по всей видимости, рационально использование алгоритмов, не основанных на переборе ветвей, а непосредственно преобразующих задачу тестирования в системы уравнений для SAT Solvers [2].

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] Rupak Majumdar and Ru-Gang Xu. Reducing test generation with information partitions // Lecture Notes in Computer Science. 2009. V. 5643/2009. P. 555-569.

[4] Асанов М., Баранский В., Расин В. Дискретная математика: Графы, матроиды, алгоритмы. Ижевск: НИЦ "Регулярная и хаотическая динамика". 2001. 288 с.

[5] Щербаков А.С. Ускорение направленного автоматического тестирования ПО в практике моделирования СБИС за счет сокращения обходов ветвей условных переходов // V Всероссийская научно-техническая конференция «Проблемы разработки перспективных микро- и наноэлектронных систем - 2012». Сборник трудов / под общ. ред. академика РАН А.Л. Стемпковского. М.: ИППМ РАН, 2012. С. 89-94.