УДК 519.686.2

# ГЕНЕРАЦИЯ ТЕСТОВЫХ СЦЕНАРИЕВ НА ОСНОВЕ ФОРМАЛЬНОЙ МОДЕЛИ

*А.А. Летичевский, А.В. Колчин*
Институт кибернетики имени В.М. Глушкова НАН Украины,
03680, Киев, проспект Академика Глушкова, 40.
E-mail: lit@iss.org.ua, kolchin_av@yahoo.com

Описан метод направленного поиска для автоматического построения тестовых сценариев в процессе верификации. Основная цель – достижение семантического соответствия между полученными тестовыми сценариями и функциональными спецификациями к системе. Метод использует определяемые пользователем в виде регулярных выражений цели тестирования и ограничения обхода поведения модели.

A guided search method for automatic test scenario building during verification described. The main goal is to achieve semantic correspondence between obtained test scenarios and functional specifications of a system. The method uses user-defined regular expressions as test purposes and for model behavior traversal bounding.

## Введение

Для проверки соответствия требованиям конкретной реализации разрабатываемых реальных систем выполняется тестирование. Так как при тестировании проверить поведение программы во всех ситуациях невозможно, прибегают к определению критериев покрытия и ограничиваются требованием проверки классов тестовых сценариев, удовлетворяющих таким критериям. Трудоемкость создания тестов по функциональным спецификациям вручную (или с применением симуляторов) не приемлема для систем со сложной моделью поведения. Обостряется необходимость не только верификации моделей систем, но и автоматизации создания тестовых сценариев. Так, в последнее время активно развиваются методы тестирования на основе моделей.

Построение тестовых сценариев на основе моделей (model based testing) использует формальные модели требований и спецификаций. Получаемые в результате тестовые сценарии обычно слабо связаны со специфическими особенностями кода тестируемой системы, но содержат представительный набор ситуаций с точки зрения исходной модели. Несмотря на то, что реализация систем отличается уровнем абстракции от своих моделей, такой подход позволяет автоматизировать процесс генерации тестов из формальных спецификаций системы и значительно сокращает трудозатраты на тестирование. Истоки методологии тестирования на основе моделей лежат в проведенных в 1950-х годах исследованиях возможности определения структуры конечного автомата, моделирующего поведение исследуемой системы, на основе результатов экспериментов с этой системой [1]. Впоследствии были разработаны методы [2, 3], позволяющие на основе модели в виде конечного автомата строить наборы тестов, успешное выполнение которых при определенных ограничениях на проверяемую систему гарантирует эквивалентность поведения этой системы заданной модели. В конце 1980-х годов интерес к тестированию на основе моделей возобновляется, и начинают разрабатываться новые подходы [4, 5], использующие другие виды моделей, помимо конечных автоматов. Основной областью их применения в то время было тестирование реализаций телекоммуникационных протоколов на соответствие стандартам этих протоколов. Примерно в это время разрабатывается стандарт ISO 9646 на такое тестирование, учитывающий возможность использования формальных моделей.

### Системы и методы создания тестовых сценариев на основе моделей

**Тестирование**

К недостаткам верификации можно отнести проблему комбинаторного взрыва числа состояний, а так же тот факт, что свойства проверяются на модели, а не на реальной системе. Для проверки соответствия требованиям реальной системы наиболее очевидным и широко распространенным методом в настоящее время является тестирование – проверка работы построенной системы в различных ситуациях при различных исходных данных. У тестирования есть недостатки:

- тестирование очень трудоемкий процесс;
- полномасштабное интеграционное и системное тестирование выполняется на поздних этапах разработки системы, когда модули системы уже реализованы;
- все реакции системы при выполнении тестов должны быть заранее зафиксированы;
- тестированием можно проверить лишь очень немногие траектории вычисления системы (а их обычно бесконечное число);
- тестирование сложных систем трудно автоматизируется;
- тестированием невозможно выявить редко проявляющиеся ошибки, особенно ошибки синхронизации в параллельных системах, которые могут зависеть от соотношения скоростей обработки процессов.

И хотя, как заметил Дейкстра, тестированием можно доказать только наличие ошибок [6]. Тестирование имеет очень важное преимущество: проверяется реальная система, а не ее абстрактная модель, таким образом, поведение системы может быть проверено в реальной среде с реальными интерфейсами. Так как при тестировании проверить поведение программы во всех ситуациях невозможно, прибегают к определению критериев покрытия и ограничиваются требованием проверки классов тестовых сценариев, удовлетворяющих таким критериям. Трудоемкость создания тестов по функциональным спецификациям (вручную или с применением симуляторов) слишком велика для систем со сложной моделью поведения, а качество таких тестов не приемлемо для систем, в которых надежность критична. Обостряется необходимость не только верификации моделей систем, но и автоматизации создания тестовых сценариев на основе формальных моделей.

**Тестирование на основе формальной модели**

В настоящее время существуют различные подходы построения тестовых сценариев на основе моделей – применяются достаточно развитые методы проверки согласованности автоматов и систем переходов, используются результаты формальной проверки свойств модели, дедуктивный анализ, символьное выполнение. Методы построения тестов на этапе проверки свойств используют формальный анализ модели для классификации тестовых ситуаций и нацеленной генерации тестов. Далее представлен краткий обзор методов проверки согласованности автоматов и систем переходов [7].

– Обычные и расширенные конечные автоматы [8, 9]. Методы построения тестов на основе конечных автоматов наиболее глубоко разработаны, известны их точные ограничения и гарантии полноты выполняемых проверок. Методы, использующие расширенные автоматы, сводят их к обычным, но применяют более детальные критерии покрытия, основанные на использовании данных в расширенных автоматах. Наиболее известными инструментами создания тестов на основе таких моделей являются BZ-TT [9, 10], использующий модели, описанные на языке B, и Gotcha-TCBeans [11], или UML Statecharts в рамках набора инструментов AGEDIS.

– Системы переходов [8]. Такие методы чаще используются при тестировании распределенных систем, поскольку моделирование таких систем с помощью конечных автоматов очень трудоемко. Большинство этих методов не определяют практически применимых критериев полноты и не дают конечных тестовых наборов для реальных систем, поэтому использующие их инструменты опираются на те или иные эвристики для обеспечения конечности набора тестов. Первые методы построения тестов по LTS-моделям были разработаны в работах Бринксмы [12] и Тритманса [13]. Наиболее известные из инструментов, созданных на основе результатов Тритманса, TorX [14] и TGV [15]. Помимо обычных систем переходов используются и временные (расширенные с помощью таймеров), построение тестов на их основе возможно с использованием инструмента UPAAL [16]. Примеры инструментальной поддержки таких методов – инструменты технологии UniTESK [17], созданной в ИСП РАН, и инструмент SpecExplorer, разработанный в Microsoft Research.

Генерация тестовых сценариев поддерживается многими инструментами, использующими автоматные модели целевой системы. Такие инструменты хорошо подходят для верификации систем, при разработке которых используются формальные языки спецификаций, например SDL, LOTOS, Estelle. Более подробные обзоры инструментов, поддерживающих автоматные методы см. в [8]. Далее приведен краткий обзор методов построения тестов на основе формального анализа свойств модели и символьного выполнения [7].

– Методы на основе классов эквивалентности [18–20]. В рамках таких методов тестовые ситуации выбираются как представители классов эквивалентности, задаваемых критерием покрытия. Часто за основу разбиения выбирается используемый критерий покрытия, а ситуации, которые соответствуют одному покрываемому элементу в рамках этого критерия, считают эквивалентными. Далее тесты строятся так, чтобы в каждом классе эквивалентности был хотя бы один тест. Пример такой техники – метод функциональных диаграмм из книги Майерса [21].

– Методы на основе дедуктивного анализа (например, [22]). В этих методах выбираемые тестовые ситуации соответствуют особым случаям в дедуктивном анализе свойств тестируемой системы.

– Методы построения тестов с помощью символьного выполнения (symbolic execution) используют символьное описание пути, проходимого во время выполнения теста по коду программы (или формальных проверяемых спецификаций) в виде набора предикатов. Это описание позволяет выбирать новые тестовые ситуации так, чтобы они покрывали другие пути и строить тесты с помощью техник разрешения ограничений (constraint solving) [23, 24]. Символьное выполнение в комбинации с конкретизацией тестовых данных используется и для построения тестов, нацеленных на типичные дефекты, такие, как использование неинициализированных объектов, тупики и гонки параллельных потоков. Более детальную таксономию методов и инструментов тестирования на основе моделей можно найти в [7, 25, 26].

Системы, позволяющие автоматизировать построение тестовых наборов на этапе верификации, обычно оценивают полноту покрытия тестовым набором по следующим метрикам и критериям: число выполненных операторов программы, ветвей, путей, проверенных значений данных, покрытие граничных значений функций и предикатов модели, переходов между состояниями и др. Вопрос о том в каком соотношении находятся показатели тестового покрытия модели и требований к целевой системе, в общем случае, как правило, не рассматривается. Таким образом семантическое соответствие между полученными тестовыми сценариями, которые удовлетворяют вышеописанным критериям, и основными функциональными спецификациями в целевой системе нет.

В работах [27–29] помимо спецификаций поведения системы, на вход подается сценарий тестирования, называемый обычно целью теста (test purpose), такой сценарий задается пользователем в виде последовательности сообщений, которыми обмениваются компоненты модели (например, в виде MSC [29]) и может быть использован в качестве направления при поиске труднодостижимых состояний. Использование эвристических методов направления поиска для обнаружения ошибок модели так же описаны в [30, 31].

Системы верификации [32–34] предполагают, что свойство, которое необходимо проверить на модели, задано в виде некоторой формулы темпоральной логики. Этим можно было бы воспользоваться, сформулировав такую формулу, которая станет ложной на некотором пути (такой путь будет выдан верификационной системой в качестве контрпримера), и при этом путь будет рассматриваться как тестовый сценарий, покрывающий некоторое требование. Однако, на практике однозначного соответствия между состоянием (значением атрибутов), описываемым такой формулой и желаемой последовательностью событий нет; более того, высокий уровень необходимых знаний в области математической логики, предъявляемый пользователю, часто является существенным ограничивающим фактором, усложняющим внедрение формальных методов в процесс промышленной разработки программного обеспечения. Таким образом, актуальна следующая задача: построить метод направленного поиска поведений модели, целью которого является генерация тестовых сценариев, удовлетворяющих заданным критериям покрытия требований к системе. Метод должен:

- принимать на вход описание целей тестирования в терминах событий (переходов) модели;
- идентифицировать трассы, удовлетворяющие целям тестирования;
- существенно сокращать время поиска трасс, удовлетворяющих целям тестирования;
- иметь средства ослабления эквивалентности трасс для минимизации их результирующего набора.

Далее описаны методы направленного поиска, а так же методы, которые предназначены для решения задачи генерации тестовых сценариев, удовлетворяющих заданным критериям покрытия. Критерии покрытия формулируются в виде специальных регулярных выражений, допускающих слова конечной длины. Такие цели тестирования способны абстрактно задать предполагаемое поведение модели в терминах событий модели, одновременно накладывая ограничения на поиск. Описаны методы фильтраций поведения модели, отсечения несовместимых ветвей поведения модели, автоматического построения тестовых сценариев, а так же стратегии управления поиском и техника ослабления эквивалентности трасс и состояний по правилам, сформулированным пользователем.

## Направленный поиск и создание тестовых сценариев

Описываемый метод позволяет пользователю задавать критерии покрытия, которые будут одновременно служить направлением поиска трасс для тестовых сценариев при обходе пространства поведения модели [35, 36]. Метод подразумевает накладывание ограничений на размер тестового сценария, что дает возможность проверить его допустимость. Критерии покрытия накладывают дополнительные ограничения на поиск, отсекая ветви поведения модели, не удовлетворяющие тестовому сценарию. Метод можно использовать в качестве определяемой пользователем эвристики при решении задач достижимости [37].

Как правило, события формальной модели ассоциируются с именами ее переходов, поэтому рассматриваются формальные модели, в которых множество меток есть множество переходов.

***Определение 1.*** Транзиционной системой $M$ называется тройка $<Q, q^0, T>$, где $Q$ – множество состояний, $T$ – множество имен переходов, $q^0 \in Q$ – начальное состояние.

***Определение 2.*** Путем в $M$ из состояния $q^i$ в состояние $q^j$ называется такая последовательность состояний и переходов $q^i \xrightarrow{t_i} q^{i+1} \xrightarrow{t_{i+1}} q^{i+2} \ldots q^j$, что $q^k \in Q \wedge t_k \in T$ для всех $k \in i..j$.

***Определение 3.*** Трассой в $M$ называется упорядоченная последовательность $t_0, t_1, \ldots t_n, \ldots$ такая, что существует путь $q^0 \xrightarrow{t_0} q^1 \xrightarrow{t_1} \ldots \xrightarrow{t_n} q^n \ldots$

***Определение 4.*** Язык, ассоциированный с $M$ обозначается $\mathcal{L}(M) \subset L^*$ – это набор всех трасс, выходящих из начального состояния $q^0$.

Далее представлен метод направленного поиска поведений модели. В качестве условия покрытия и направления поиска – цели теста («абстрактного» тестового сценария) – используются специальные регулярные выражения над алфавитом имен переходов модели (далее образцы), описывающие множества трасс. Такие выражения позволяют описывать как последовательную, так и параллельную композицию поведения процессов, а так же использовать операции отрицания и недетерминированного выбора переходов.

***Определение 5.*** Пусть заданы непересекающиеся алфавиты $X$ и $Y$, а так же язык $Z$ над $X$. Пусть задано отображение $\lambda : X \to X \cup Y$, определенное как $\lambda(x) = \{AxB \mid A, B \in Y^*\}$. Тогда язык $Z_{\Uparrow Y} = \{\lambda(\gamma) \mid \gamma \in Z\}$ над алфавитом $X \cup Y$ является расширением языка $Z$ до алфавита $Y$.

Расширение дистрибутивно:

$(Z^a \cup Z^b)_{\Uparrow I} = Z^a_{\Uparrow I} \cup Z^b_{\Uparrow I}$;

$(Z^a \cap Z^b)_{\Uparrow I} = Z^a_{\Uparrow I} \cap Z^b_{\Uparrow I}$.

***Определение 6.*** Образцом называется

$a.n$ – переход $a$ на максимальной дистанции $n$, допускает трассы $a \vee X_1 a \vee X_1 .. X_n a$, где $X_1,...,X_n$ – любые непустые символы из $L \setminus a$,

$\sim a$ – запрет перехода $a$, допускает любой символ из $L \setminus a$,

$a;b$, где $a,b$-образцы – конкатенация образцов, допускает множество трасс $(ab)$,

$a \vee b$, где $a,b$-образцы – недетерминированный выбор образцов, допускает как множество трасс $a$, так и трасс $b$.

$a \parallel b$, где $a$ – образец, представляющий язык $Z^a$ над алфавитом множества $X$, $b$ – образец, представляющий язык $Z^b$ над алфавитом множества $Y$, где $X$ и $Y$ – непересекающиеся множества переходов $X \cap Y = \varnothing$. Тогда параллельная композиция образцов представляет язык $Z^{a\parallel b} = Z^a_{\Uparrow Y} \cap Z^b_{\Uparrow X}$.

$loop(a)$, где $a$-образец – итерация образца $a$, т. е. $aa*$.

Переходы, входящие в образец, называются наблюдаемыми. Два наблюдаемых перехода на трассе расположены на дистанции *x*, если между ними расположены *x*-1 переходов. Дистанция первого перехода в образце – расстояние от начального состояния.

Запреты действуют до обнаружения следующего допустимого наблюдаемого перехода и не могут быть последними в образце (за исключением использования внутри итерации).

Параллельная композиция образцов используется для случаев, когда модель имеет несколько параллельно работающих компонент (процессов), причем дистанция учитывается для каждого параллельного участка образца отдельно. Последовательная композиция обладает свойством ассоциативности:

$$a;(b;c) = (a;b);c.$$

Операции параллельной композиции и недетерминированного выбора имеют следующие свойства:

– $a \vee (b \parallel c) = (a \vee b) \parallel (a \vee c)$ – недетерминированный выбор дистрибутивен относительно параллельной композиции;

– $a \parallel (b \vee c) = (a \parallel b) \vee (a \parallel c)$ – параллельная композиция дистрибутивна относительно недетерминированного выбора;

– $a \vee b = b \vee a$, $a \parallel b = b \parallel a$ – коммутативны;

– $a \vee (b \vee c) = (a \vee b) \vee c$, $a \parallel (b \parallel c) = (a \parallel b) \parallel c$ – ассоциативны.

Свойством идемпотентности обладает только операция недетерминированного выбора $a \vee a = a$, так как значение $a \parallel a$ не определено. Операция $loop(a)$ может входить в образец не более одного раза и только в конце образца.

Построенная трасса удовлетворяет образцу, если она содержит наблюдаемые переходы в непротиворечащей заданному образцу последовательности, на дистанциях, не превышающих указанные. Например, образец

$trA.2;(trB.2 \vee \sim trZ;trC.4)$

имеет примеры удовлетворяющих трасс:

$trS,trA,trZ,trB;$

$trA,trX,trC;$

пример не удовлетворяющей трассы:

$trA,trX,trZ,trC,trB.$

Как видно, выражения не содержащие итерации, всегда имеют максимальную длину трассы, удовлетворяющей образцу; более того, в отличие от [28], определить, что трасса не удовлетворяет образцу можно не достигая этого максимума, так как всегда определена максимальная дистанция между событиями. В приведенном примере максимальная длина трассы 6, в то время как трасса $trA,trZ,trC$, имея длину 3 уже не имеет допустимого образцом продолжения, и может быть прервана.

Семантически такие образцы задают «контрольные точки» поведения модели и так же определяют критерий выбора построенных трасс (вариантов конкретного поведения системы) для последующего их использования в качестве тестовых сценариев [37]. Так, предполагаемые поведения моделируемой системы, описанные пользователем, проверяются на допустимость, одновременно накладывая ограничения на поиск. По заданным образцам строится соответствующий детерминированный конечный автомат. Для различных видов регулярных выражений существуют алгоритмы различной вычислительной сложности [38, 39]. Важно отметить, что на практике удовлетворительный результат можно получить, используя жадные алгоритмы линейной сложности.

Обход пространства поведения, реализующий адаптированный алгоритм Тарьяна [40], модифицируется путем добавления перед шагом вглубь вызова специальной функции, реализующей автомат, распознающий образцы, при этом *полным* состоянием модели будет синхронное произведение состояний модели (значений атрибутов) и состояний автомата, распознающего образцы. Если текущая трасса удовлетворяет образцу, она будет сохранена на жестком диске, если не исчерпан соответствующий лимит (пользователь определяет нужное количество трасс по каждому образцу). Если лимит исчерпан, образец исключается из списка актуальных. Если данная функция возвращает на выходе `false`, это означает, что достигнута ситуация, из которой любое продолжение текущей трассы не приведет к обнаружению вхождения очередного символа (т.е. имени наблюдаемого перехода) хотя бы одного актуального образца. Такая ситуация – дополнительный критерий завершения генерации текущей трассы. Таким образом алгоритм выполнит шаг назад. При этом, если трасса содержит исторически максимальной длины префикс хотя бы одного образца, она будет сохранена на диске с соответствующей пометкой. Если полного вхождения какого-либо образца не обнаружено, будет построена трасса, удовлетворяющая максимальному префиксу этого образца. Дополнительным для сохранения трассы является условие вхождения в трассу перехода, не входящего ни в одну из уже сохраненных трасс. В итоге будет построен набор трасс, включающий как предполагаемые пользователем поведения модели, так и все достигнутые переходы модели.

## Использование итерации

Для обнаружения бесконечных циклов используется итерация (замыкание Клини). Допускается использование не более одной итерации и только в конце образца. Необходимо отметить, что под циклом в данном случае понимается цикл полных состояний, т.е. синхронное произведение состояний автомата регулярного выражения и значений атрибутов модели. Например, используя итерацию можно проверить, приходит ли модель системы в устойчивое состояние при отсутствии внешних сигналов. Пусть $trE1$ и $trE2$ – единственные воздействия (переходы) внешней среды, а $trA$ – переход некоторого единственного процесса $ms$. Пусть так же задан образец $trE1.2; trA.4; loop(\sim trE1 \parallel \sim trE2)$. Тогда трасса, удовлетворяющая образцу, должна иметь цикл, в котором нет переходов $trE1$ и $trE2$, а префикс этой трассы должен включать переходы $trE1$ и $trA$ на соответствующих дистанциях. Таким образом, после воздействия среды событием, размеченным переходом $trE1$ и последующей реакцией системы – переходом $trA$, процесс $ms$ не приходит в устойчивое состояние (в цикл не входит ни одно событие от внешней среды).

Важно отметить, что на практике удовлетворительный результат можно получить, используя жадные алгоритмы линейной сложности. Из соображений эффективности реализация метода принимает на вход не один, а множество образцов, для каждого из которых строятся соответствующие автоматы, при этом осуществляется множественный поиск трасс одновременно всего набора образцов, а так же вводятся ограничители на число необходимых трасс по каждому образцу.

## Ослабление эквивалентности трасс

В данном случае ослабление трассовой эквивалентности подразумевает разделение модели на тестируемую и тестирующую инстанции. Такая техника часто применяется в случае, когда в многокомпонентной системе необходимо протестировать одну из компонент, при этом остальные компоненты объединяются в одну инстанцию, которая играет роль некоторой внешней среды для тестируемой компоненты. Например, событие в переходе описывается MSC диаграммой, тогда результирующая трасса записывается в виде MSC, сценария состоящего из последовательности событий соответствующих трассе переходов. Так, для ослабления эквивалентности пользователь выделяет два набора инстанций, после чего две трассы считаются эквивалентными, если в них совпадают поведения тестируемых инстанций. Такой подход позволяет получить значительно уменьшенный набор тестов с необходимым уровнем покрытия для выборочной проверки модулей.

Нижеописанная процедура `Save_Abstract_Trace` строит абстрактные трассы, используя заданную эквивалентность:

**Вход:** Полная трасса `Trace`, множество тестируемых инстанций `Tested_Instances` и идентификатор теста `Test_ID`.
**Выход:** Абстрактная трасса `Abstract_Trace`.

```
Save_Abstract_Trace:= procedure(Trace, Tested_Instances,Test_ID)
                        local(Abstract_Trace, Transition, Event)
begin
   Abstract_Trace ← ∅;
   for Transition ∈ Trace do
      for Event ∈ Transition do
         if (instance(Event) ∈ Tested_Instances) then do
            Abstract_Trace ← (Abstract_Trace ; Event);
   Save_Trace(Abstract_Trace, Test_ID)
end          □
```

На практике актуальна минимизация количества тестов. Процедура `Save_Trace` обеспечивает поиск вхождения одного теста в другой, и, в случае обнаружения, новая трасса не добавляется к тестовым сценариям, либо заменяет ранее добавленную:

**Вход:** Абстрактная трасса `Abstract_Trace` и идентификатор теста `Test_ID`.
**Выход:** Обновленное множество `Stored`.

```
Save_Trace := procedure(Abstract_Trace, Test_ID)local(T)begin
   for T ∈ Stored do begin
      if (Abstract_Trace ⊆ T.trace) then do return;
      if (Abstract_Trace ⊃ T.trace) then do begin
         *T.trace ← Abstract_Trace;
         *T.test_id ← {T.test_id ∪ Test_ID};
         return
      end
   end
   Stored ← {Stored ∪ (Abstract_Trace, Test_ID)}
end        □
```

В результате множество `Stored` будет содержать минимизированный набор пар, состоящих из абстрактной трассы и списка идентификаторов тестовых целей.

## Выводы

Описанный метод является разновидностью ограниченной проверки модели (bounded model checking), в качестве ограничителей использует цели тестирования, определяемые пользователем. В отличие от существующих методов направления поиска и генерации тестов, направлением для поиска служит образец – специальное ограниченное регулярное выражение над алфавитом имен переходов модели. Такой образец служит «шаблоном» тестового сценария, используется для отсечения несовместимых ветвей поведения модели и одновременно для проверки предполагаемого поведения на допустимость, при этом, по сути, осуществляется валидация модели. Использование коротких дистанций и запретов между контрольными точками образцов позволяет эффективно находить труднодостижимые состояния. Включение операции итерации в описание направления позволяет проверять нарушение специфических условий безопасности, например, наличие бесконечных циклов в поведении процесса при отсутствии внешних сигналов. Методы ослабления эквивалентности трасс существенно сокращают количество тестовых сценариев обеспечивающих необходимое покрытие и время поиска.

В отличие от [32–34] желаемые свойства системы формулируются в терминах событий модели; в отличие от [27] метод не подразумевает внесения изменений в постусловия переходов исходной модели. Системы [27–29] выполняют поиск с фиксированным ограничением на длину трассы, таким образом, отсутствие результата не означает, что модель не допускает желаемого тестового сценария. В отличие, описанный метод фиксирует ограничения на тестовый сценарий, при этом гарантирует обнаружение удовлетворяющих трасс.

Необходимо отметить, что метод направленного поиска может быть использован совместно с методами абстракции предикатов, частичного порядка и многими другими методами верификации.

Применение технологии направленного поиска в промышленных проектах дало возможность не только выявить ошибки в спецификациях, но и построить набор тестовых сценариев, обеспечивающих необходимое функциональное покрытие [37].


1. *Мур Э.* Умозрительные эксперименты с последовательными машинами // Автоматы. М.: Изд-во иностр. лит. – 1956. – С. 179–210.
2. *Hennie F.* Fault-detecting experiments for sequential circuits // Proc. of 5-th annual symposium on switching theory and logical design. – 1964. – P. 95–110.
3. *Василевский М.* О распознавании неисправностей автоматов // Кибернетика. – 1973. – № 4. – С. 98–108.
4. *Bourdonov I., Kossatchev A., Petrenko A., Galter D.* KVEST: Automated Generation of Test Suites from Formal Specifications // Proc. of Formal Methods'99, LNCS 1708. – 1999. – P. 608–621.
5. *Петренко А.* Спецификация тестов на основе описания трасс // Программирование. – 1993. – № 19(1). – С. 66–73.
6. *Дейкстра Э.* Дисциплина программирования. – М.: Мир, 1978.
7. *Кулямин В.* Методы верификации программного обеспечения // Всероссийский конкурсный отбор обзорно-аналитических статей по приоритетному направлению "Информационно-телекоммуникационные системы". – 2008. – 117 С.
8. *Broy M., Jonsson B., Katoen J., and oth.* Model Based Testing of Reactive Systems // LNCS 3472. – 2005. – P. 659.
9. *Utting M., Legeard B.* Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann. – 2007. – 456 P.
10. *Ambert F., Bouquet F., Chemin S., Guenaud S.* and oth. BZ-TT: A tool-set for test generation from Z and B using constraint logic programming // Proc. of FATES'2002. – 2002. – P. 105–119.
11. *Farchi E., Hartman A., Pinter S.* Using a model-based test generator to test for standard conformance // IBM Systems Journal. – 2002. – Vol. 41(1). – P. 89–110.
12. *Brinksma E.* A theory for the derivation of tests // Proc. of 8-th International conference on protocol specification, testing and verification. – 1988. – P. 63–74.
13. *Tretmans J. A* Formal Approach to Conformance Testing // PhD thesis, University of Twente, Enschede, The Netherlands. – 1992.

14. *Tretmans J., Belinfante A.* Automatic testing with formal methods // Proc. of 7-th European Conference on Software Testing, Analysis and Review, Barcelona, Spain. – 1999. – P. 8–10.
15. *Fernandez J., Jard C., Jeron T., Viho C.* Using On-the-Fly Verification Techniques for the Generation of Test Suites // Proc. of 8-th International Conference on Computer Aided Verification, LNCS 1102. – 1996. – P. 348–359.
16. http://www.uppaal.com
17. *Кулямин В., Петренко А., Косачев А., Бурдонов И.* Подход UniTesK кразработке тестов // Программирование. – 2003. – № 29(6). – С. 25–43.
18. *Ammann P., Black P.* Abstracting formal specifications to generate software tests via model checking // Proc. of 18-th Digital Avionics Systems Conference, IEEE. – 1999. – Vol. 2. – P. 10.A.6–1–10.A.6–10.
19. Gargantini A., Heitmeyer C. Using model checking to generate tests from requirements specifications // ACM SIGSOFT Software Engineering Notes. – 1999. – Vol. 24(6). – P. 146–162.
20. *Visser W., Pasareanu C., Khurshid S.* Test input generation with Java PathFinder // ACM SIGSOFT Software Engineering Notes. – 2004. – № 29(4). – P. 97–107.
21. *Майерс Г.* Искусство тестирования программ. М.: Финансы и статистика. – 1982. – 174 С.
22. *Engel C., Hahnle R.* Generating unit tests from formal proofs // Y. Gurevich, B. Meyer, eds. Proc. of TAP 2007, LNCS 4454. – 2007. – P. 169–188.
23. *Boyapati C., Khurshid S., Marinov D.* Korat: automated testing based on Java predicates // Proc. of International Symposium on Software Testing and Analysis. – 2002. – P. 123–133.
24. *Gotlieb A., Botella B., Rueher M.* Automatic test data generation using constraint solving techniques // ACM SIGSOFT Software Engineering Notes. – 1998. – Vol. 23(2). – P. 53–62.
25. *Utting M., Pretschner A., Legeard B. A.* Taxonomy of Model-Based Testing // Technical Report, Department of Computer Science, The University of Waikato, New Zealand. – 2006. – 17 P.
26. *Бурдонов И., Косачев А., Пономаренко В., Шнитман В.* Обзор подходов к верификации распределенных систем. М.: ИСП РАН. – 2006. – 61 С.
27. *Bourdonov I., Kossatchev A., Kuliamin V., and Petrenko A.* UniTesK Test Suite Architecture // In Proc. of FME 2002, LNCS 2391, Springer-Verlag. – 2002. – P. 77–88.
28. *Fernandez J., Jard C., Jeron T., Nedelka L., and Viho C.* An experiment in automatic generation of test suites for protocols with verification technology // Science of Computer Programming. – 1997. – Vol. 29. – № 1–2. – P. 123–146.
29. *Grabowski J., Hogrefe D., Nahm R.* Test case generation with test purpose specification by MSCs // In Elsevier Science B.V. (North-Holland), editor, 6th SDL Forum. – 1993. – P. 253–266.
30. *Edelkamp, S., Leue S. and Lafuente A.* Directed explicit-state model checking in the validation of communication protocols // International journal on software tools for technology transfer. – 2003. – № 5. – P. 247–267.
31. *Lafuente A.* Directed Search for the Verification of communication protocols // Doctorial thesis, Institute of computer science, University of Freiburg. – 2003. – 157 P.
32. *Ben-Ari. M.* Principles of Spin // Springer Verlag. – 2008. – 216 P.
33. *Burch J., Clarke E., McMillan K., Dill D., and Hwang L.* Symbolic model checking: 10^20 states and beyond // Information and Computation. – 1992. – Vol. 98. – № 2. – P. 142–170.
34. *Cimatti A., Clarke E. M., Giunchiglia E.,* and others. NuSMV 2: An OpenSource Tool for Symbolic Model Checking // In Proceeding of International Conf. on Computer-Aided Verification, Copenhagen, Denmark. – 2002. – P. 359–364.
35. *Колчин А.* Метод направления поиска и генерации тестовых сценариев при верификации формальных моделей асинхронных систем // Проблемы программирования. – № 4. – 2008. – С. 5–12.
36. *Колчин А.* Направленный поиск в верификации формальных моделей // Тези доп. міжнар. конф. «Теоретичні та прикладні аспекти побудови програмних систем TAAPSD'2007» – Бердянск. НаУКМА, Національний ун-т ім. Т.Г. Шевченка, Ін-т програмних систем НАН України. – 2007. – С. 256–258.
37. *Баранов С., Котляров В., Летичевский А.* Индустриальная технология автоматизации тестирования мобильных устройств на основе верифицированных поведенческих моделей проектных спецификаций требований // Труды междунар. науч. конф. «Космос, астрономия и программирование» – СПбГУ, С-Пб. – 2008. – С. 134–145.
38. *Aho A.* Algorithms for finding patterns in strings // Handbook for theoretical computer science, MIT Press. – 1990. – Vol. A. – P. 257–300.
39. *Smyth B.* Computing Patterns in Strings // ACM Press Books. – 2003. – 440 P.
40. *Tarjan R.* Depth first search and linear graph algorithms // SIAM Journal on Computing. – 1972. – Vol. 1. – № 2. – P.146–160.