УДК 519.686

Е. Е. Хатько

Московский физико-технический институт (государственный университет)
ООО «Даймонд Кволити»

Об одном методе тестирования «мобильных» приложений

В статье предложен метод тестирования пользовательского интерфейса «мобильных» приложений, основанный на применении расширенных конечных автоматов. Суть метода заключается в выделении состояний конечного автомата путем декомпозиции приложения и последующей генерации тестов с применением алгоритма траверса графов. Рассмотрена одна из реализаций алгоритма генерации тестового набора.

Ключевые слова: Тестирование, конечные автоматы, алгоритм траверса, обход графа.

Введение

Тестирование на основе моделей основано на построении модели приложения с последующей генерацией тестовых наборов. Модель – есть некоторое упрощенное описание тестового приложения на формальном языке или с помощью диаграмм.

Построение модели само по себе не решает задачи тестирования приложения. Для получения готовых тестовых сценариев необходимо применить некоторый алгоритм, позволяющий вывести тестовые сценарии из построенной модели. Для оценки качества и завершенности процесса тестирования программного продукта существует понятие *тестового покрытия*. В данной статье в качестве тестового покрытия будет рассматриваться *покрытие всех возможных действий пользователя в контексте данной программы* – нажатий на кнопки и активные элементы экрана, ввод информации в устройство. Следует отметить, что пользователь может вводить в устройство любые данные, но в результате введения данных приложение переходит в одно из предопределенных состояний в соответствии со своей внутренней логикой работы. Таким образом, все вводимые данные можно разбить на классы, так что в рамках одного класса, приложение будет всегда переходить в одно и то же состояние. Такие классы входных данных будем называть *классами эквивалентности*. В зависимости от логики работы алгоритма генерации можно получать различные тестовые покрытия. В данной статье предложен алгоритм построения моделей приложений и рассмотрен алгоритм генерации тестовых сценариев, основанный на применении неинформированного алгоритма поиска кратчайшего пути на графе A^*. Особенность алгоритма генерации заключается в способности работать с динамически меняющимися графами, которые являются представлением расширенных конечных автоматов. Модели должны быть построены таким образом, чтобы обеспечивать *полное тестовое покрытие* – т.е. содержать достаточно информации, чтобы прохождение всех сгенерированных тестов означало покрытие всех возможных действий пользователя и рассмотрение всех классов эквивалентности входных данных. Помимо полного тестового покрытия модель должна обеспечивать генерацию *минимального набора тестовых сценариев*, т.е. сгенерированные тесты должны содержать наименьшее количество действий для достижения необходимого тестового покрытия.

MVC-приложения

В последнее время технология разработки Model View Controller (MVC) получила очень большое применение благодаря высокой степени расширяемости и более простой поддержке приложений. В таких приложениях всегда можно определить следующие компоненты:

Модели БД (Models) – интерфейсы для работы с некоторой базой данных. Модель БД обеспечивает работу с данными, Чтобы протестировать модель БД, нужно разбить входные данные на классы эквивалентности и проверить работу приложения для каждого такого класса:

Рис. 1. Введение данных из различных классов эквивалентности

Контроллеры (Controllers) – компоненты, описывающие логику приложения. Контроллеры задают логическую структуру приложения, следовательно, и структуру пользовательского интерфейса.

Виды (Views) – графическое отображение состояния приложения. В моделировании MVC-приложений виды обычно соответствуют состояниям конечного автомата.

В общем случае работа приложения происходит по следующей схеме:

Пользователь видит на экране некоторый вид. Вид содержит элементы пользовательского интерфейса (кнопки, поля ввода и пр.), которые позволяют совершать различные действия (запросы пользователя). Запрос, генерируемый пользователем, приходит в соответствующий контроллер, контроллер обрабатывает запрос и, возможно, обращается к базе данных (через модель), далее генерируется следующий вид, который увидит пользователь как результат своего запроса.

Очевидно выбрать для MVC-приложений моделирование с помощью конечных автоматов. При таком выборе состояние приложения характеризуется **текущим видом и состоянием базы данных.** Каждый запрос пользователя соответствует переходу из одного состояния в другое. Таким образом: «состояние автомата – это композиция вида приложения и состояния базы данных, а переход из одного состояния в другое – это обращение пользователя к контроллерам с помощью элементов пользовательского интерфейса».

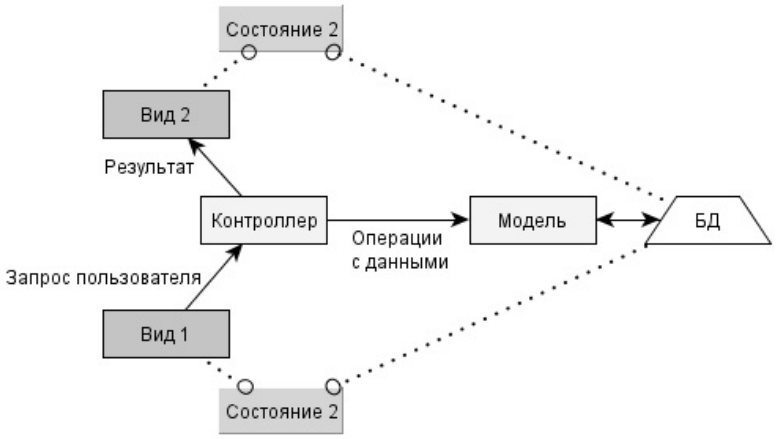

Рис. 2. Состояния конечного автомата

Вид – это результат пользовательской команды, выводимый приложением на экран. На рисунке 1 изображены виды: Вид 1, Вид 2. Каждое приложение по своей логической структуре имеет основные виды и вспомогательные.

Расширенная конечно-автоматная модель

При построении моделей реальных приложений было выявлено, что количество основных видов приложений – конечное небольшое число (порядка 10 для «мобильных» прило-

жений), но состояний базы данных может быть гораздо больше, что приводит к значительному увеличению количества состояний автомата. Решение проблемы состоит в следующем: модель строится на основе видов, так чтобы состояния конечного автомата соответствовали основным видам приложения с точностью до нескольких. Информация о базе данных вносится в каждый вид при помощи некоторых параметров, а переходы между видами становятся условными, в зависимости от значений параметров. Такой конечный автомат называется расширенным:

Расширенная конечно-автоматная модель – это усовершенствованная модель конечного автомата. В традиционном конечном автомате переход из состояния в состояние связан с набором входных булевых условий и набором выходных булевых функций. В расширенной модели переход может быть выражен “IF” выражением. Если все условия перехода соблюдены, то происходит переход, переводя автомат в следующее состояние, при этом производятся требуемые операции с данными. [1]. Формально:

расширенный конечный автомат-набор (S, V, P, s_0, P_0, I, n_I, X, T), где

S – конечное множество *состояний* автомата;

V – множество, возможно бесконечное, значений внутренних данных автомата (например, состояния БД);

P – отображение конечного набора $[1..n]$ индексов в V, P:$[1..n]$ $>$ V; значение P на индексе i называется значением i-й переменной автомата, которое также обозначается p_i.

s_0 – элемент S, называемый *начальным состоянием*;

P_0 – отображение $[1..n]$ индексов в V, называемое *начальными значениями переменных*;

I – конечное множество, элементы которого называются *операциями* или *стимулами*, само I называют *входным алфавитом* автомата (например, запросы пользователя к приложению);

n_I – отображение I в неотрицательные числа, определяет число параметров для каждого стимула;

X – множество, возможно бесконечное, значений параметров стимулов;

T – множество *переходов* автомата; каждый переход t включает *начальное управляющее состояние* s_1, стимул I, *условие перехода* g_t(guard condition) – предикат на множестве $V^n \times X^n{}_i$, *конечное управляющее состояние* s_2, и *действие* a_t – некоторое отображение $V^n \times X^n{}_i$ в множество V^n, определяющее новые значения переменных.

Выполнение расширенного автомата отличается от выполнения обычного тем, что помимо текущего состояния имеются текущие значения переменных, при приходе стимула с набором аргументов охранное условие определяет, может ли быть выполнен данный переход при текущем наборе значений переменных и заданных значениях параметров стимула. Выполняемый переход выбирается недетерминированно из всех, помеченных данным стимулом, начинающихся в данном управляющем состоянии и имеющих выполненное охранное условие. При выполнении некоторого перехода новое управляющее состояние автомата равно конечному управляющему состоянию перехода, новые значения переменных определяются при помощи его действия – новое $p_i = a_t(p_1, \ldots, p_n, x_1, \ldots, x_{nI})$, значения параметров реакции – по соответствующему отображению в переходе [2].

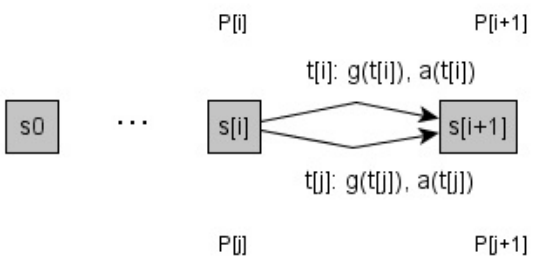

Рис. 3. Расширенный конечный автомат

Структура модели

Под хорошей моделью будем понимать модель, удовлетворяющую следующим критериям:

1) **Простота** – модель должна быть настолько простой, чтобы затраты на её построение могли бы окупиться. Хорошая простая модель не требует больших вложений на обучение персонала. Модель должна быть интуитивно понятной каждому сотруднику.

2) **Детальность** – модель должна быть настолько детальной, чтобы с её помощью можно было бы описать все состояния и параметры приложения для проведения полноценного тестирования.

3) **Тестируемость** – модель должна быть построена таким образом, чтобы при генерации тестов можно было бы получить тестовые наборы, пригодные для ручного тестирования (а в последствии, и автоматизированного).

4) **Автоматизируемость** – модель должна поддерживать потенциальную автоматизацию тестирования. То есть переходы между состояниями должны быть настолько «низкоуровневыми», чтобы их можно было передавать на вход инструмента автоматизации тестирования.

Для выбора правильного уровня абстрактности нужно произвести декомпозицию приложения, т.е. разбиение на более простые составные части.

Выделение состояний автомата

Чтобы модель MVC-приложения удовлетворяла вышеописанным критериям, нужно для начала провести разбиение приложения на составные части, как показано на следующей схеме:

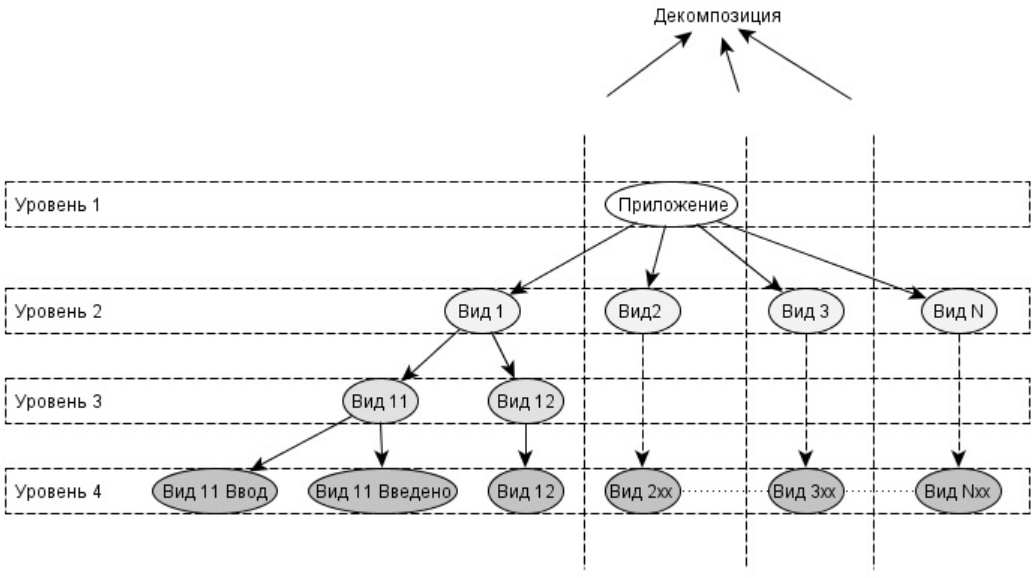

Рис. 4. Декомпозиция приложения

1) Первый уровень – уровень приложения.

2) Второй уровень – приложение разбивается на виды, соответствующие видам приложения в технологии MVC. Переход между видами осуществляется при помощи команд пользователя. Этот уровень описывает приложение с точки зрения пользователя.

3) Третий уровень – у каждого вида могут быть подвиды. Подвид может соответствовать новому состоянию базы данных приложения в этом же виде или некоторому системному диалогу, напоминанию и т.п.

4) Четвертый уровень – уровень классов эквивалентности. На этом уровне определенные подвиды предыдущего уровня разбиваются на 2 состояния – **ВидВвод** и **ВидВведено**, переходы между которыми соответствуют различным классам эквивалентности параметров расширенного конечного автомата. ВидВвод соответствует состоянию автомата, когда ввод данных ещё не был совершен. ВидВведено соответствует состоянию автомата, в котором пользователь уже ввел некоторые данные.

Алгоритм построения модели

1) Изучить документацию и выделить основные виды приложения – это элементы второго уровня.

2) Каждый основной вид может иметь такие сущности, как диалоги, напоминания и различные состояния базы данных. Каждый подвид должен соответствовать одной из перечисленных сущностей.

3) Если подвид предполагает ввод данных или выбор одного из предлагаемых значений, его нужно разбить на два: **ВидВвод** и **ВидВведено**. Далее между двумя данными состояниями нужно поставить столько переходов, сколько классов эквивалентности содержат вводимые данные. Каждый переход должен соответствовать одному из классов эквивалентности.

4) Построить следующий конечный автомат (S, V, P, s_0, P_0, I, n_I, X, T):

S – множество видов четвертого уровня приложения в результате проведения декомпозиции;
s_0 – начальное состояние автомата;
P_0, – начальные значения переменных;
V – множество состояний базы данных приложения;
P – отображение устанавливает значения переменных в каждом виде;
I – множество возможных действий пользователя;
X – множество значений вводимых данных.

Подготовка модели к генерации тестовых сценариев

Для обеспечения тестируемости приложений на вход тестового генератора необходимо подавать не всю модель целиком, а каждый вид из второго уровня по отдельности:

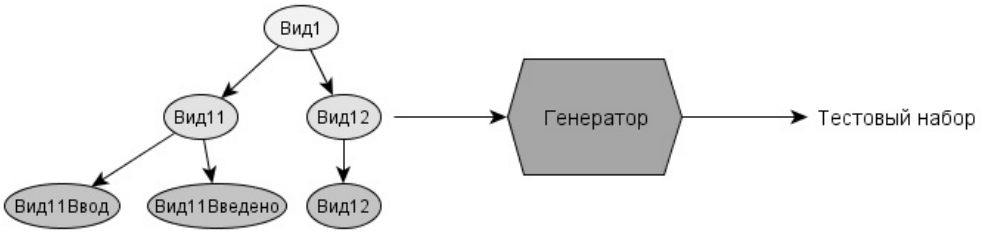

Рис. 5. Входные данные генератора

Данное условие не является обязательным, оно улучшает читабельность генерируемого тестового набора, что немаловажно в случае проведения ручного тестирования.

В дальнейшем все состояния автомата на четвертом уровне, относящиеся к одному виду второго уровня, будем называть *готовым набором состояний* (на данном рисунке это

состояния [Вид11Ввод, Вид11Введено, Вид12]). Вид, являющийся корнем дерева состояний готового набора, будем называть **основным видом** готового набора состояний (Вид 1 на рисунке).

Параметризация автомата

В каждом переходе расширенной КА-модели можно вводить параметры (p_i). Значения параметров устанавливают условия для следующих переходов (g_t). В любом переходе между видами ВидВвод и ВидВведено должен содержаться один и тот же параметр, описывающий вводимые данные. Изменения параметра на каждом переходе описывается действиями переходов (a_t). Значения параметра в каждом переходе должно соответствовать одному из значений из класса эквивалентности входных данных. При этом количество всех переходов между видами ВидВвод и ВидВведено равно количеству классов эквивалентности для данного параметра.

Начальные и конечные состояния

Вводится фиктивное начальное состояние, чтобы указать генератору начальную точку маршрута. В случае автоматизации тестирования нужно указать действия, необходимые для приведения КА к этому состоянию. Введение данного состояния обусловлено требованием инициализации алгоритма генерации тестовых сценариев: «начальное состояние должно иметь единственный переход в следующее состояние».

В результате некоторых действий пользователя автомат может перейти к следующему основному виду. Для обеспечения тестируемости приложений следующий основной вид рекомендуется сделать тупиковым состоянием для данного готового набора состояний и добавить единственный фиктивный переход в начальное состояние. Этот переход соответствует началу нового тестового сценария. В случае ручного тестирования этот переход можно обозначить как «_НОВ._ТЕСТ_». В случае автоматизации тестирования нужно указать – какие действия необходимо выполнить для перехода из конечного состояния в начальное, например «Перезапуск приложения». На рис. 4 изображена модель приложения с введенными начальными и конечными состояниями.

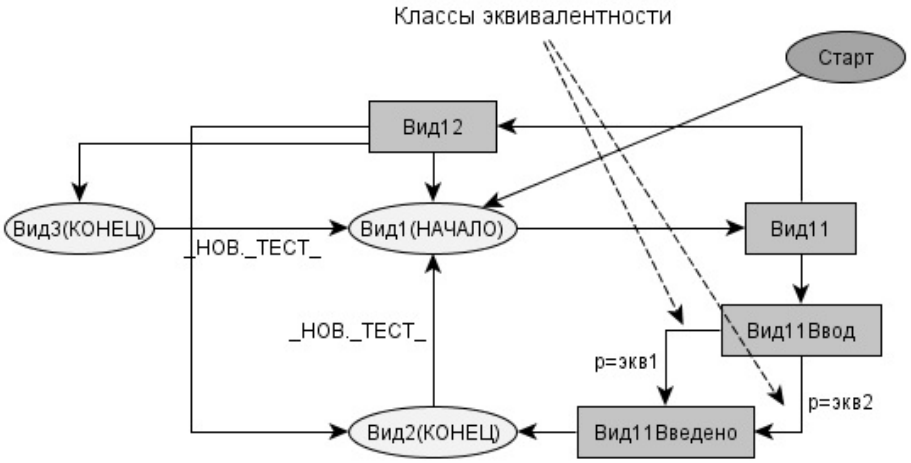

Рис. 6. Модель приложения

Генерация тестовых сценариев

Когда модель построена, каждый путь из некоторого «начального» состояния в другое состояние соответствует тестовому сценарию, поскольку переход содержит действия, которые можно произвести над тестовым объектом. Алгоритмы обхода графов позволяют генерировать тестовые наборы. Реализованный алгоритм обхода можно назвать *генератором тестовых сценариев*.

Условие окончания обхода

Как правило, целью тестирования интерфейса пользователя является проверка всех функций с различными наборами входных данных. В таком случае условием окончания обхода графа является 100% - ное покрытие всех переходов. Следует заметить, что данное условие также является достаточным для покрытия всех состояний системы, поскольку при моделировании MVC-приложений КА представляются в виде связных графов. При выборе данного критерия как критерия окончания тестирования возникает проблема выхода из тупиковых конечных состояний. При попадании в тупиковое состояние тестовый генератор должен выполнить одно из следующих действий:

1) Сообщить о попадании в тупик и предложить достигнутое покрытие – как максимально достижимое покрытие переходов.

2) Перейти в начало обхода по фиктивному переходу.

Алгоритм выбора пути

Алгоритм A^* [3, 4] является алгоритмом поиска пути с наименьшей стоимостью. Алгоритм является жадным алгоритмом, работающим по первому совпадению. Помимо этого алгоритм использует эвристическую функцию для определения дальнейшего пути. В данной работе был использован оптимизированный алгоритм A^*, который содержит информацию о параметрах графа и может выбирать маршрут с учетом условных переходов конечного автомата.

Функция стоимости F является основополагающей для работы алгоритма. Находясь в некотором узле графа, эта функция определяет – какой переход выбрать следующим. Выбирается тот переход, для которого значение F наименьшее.

Изучив задачу о нахождении Эйлерова пути на графе [5], предлагается ввести следующие условия для выбора «следующей» ветви, находясь в «текущей» (условия расположены в порядке значимости) [6]:

1) Выбирается ветвь, которая ещё не покрыта (для всех ветвей вводится флаг *непокр._ветвь* – равный 0 – если ветвь непокрыта и 1 – в противном случае);

2) Выбирается ветвь, входящая в узел, который имеет наибольшее значение $\delta = (исход. - вход.)$, где *исход.* – количество исходящих ветвей, *вход.* – количество входящих ветвей в узел;

3) Выбирается ветвь, условие которой «удаляет» наименьшее количество непокрытых ветвей (*ветви_удал._условием*);

4) Выбирается ветвь, условие которой делает недоступными наименьшее количество непокрытых ветвей (ветви становятся недоступными благодаря удалению других ветвей и 3 пункта) (*ветви_недост._при_усл.*);

5) Выбирается ветвь, ведущая в узел с наибольшим количеством непокрытых исходящих ветвей (*непокр._исх._ветви*).

Приоритет ветвей устанавливается при помощи специального масштабного коэффициента: $K = \max\limits_{i \in Узлы} (вход._i + исход._i)$ – по всем узлам, где $вход._i/исход._i$ – количество входящих/исходящих ветвей для i-го узла. Таким образом:

$$F = непокр.\_ветвь \times K^3 - \delta \times K^2 + ветви\_удал.\_условием \times K +$$

$$+ ветви\_недост.\_при\_усл. \times K - непокр.\_исх.\_ветви.$$

Смысл функции F заключается в том, что при определении следующей ветви выбирается ветвь с наименьшим значением F.

Данный выбор функции стоимости позволяет реализовать алгоритм генерации тестовых сценариев на основе расширенных конечных автоматов.

Для оценки эффективности алгоритма применялась следующая методика: для каждого узла графа сравнивались значения посещений узла в результате работы алгоритма $N(A^*)$ со значением максимума из исходящих и входящих ветвей узла $N(EULER) = \max(in, out)$.

$$Eff = \left\{1 - \frac{N(A^*) - N(EULER)}{N(EULER)}\right\} \times 100\%.$$

Для оценки были выбраны два графа без тупиковых узлов: G1 – с большим количеством ветвлений и малым количеством узлов и G2 – приближенный к дереву:

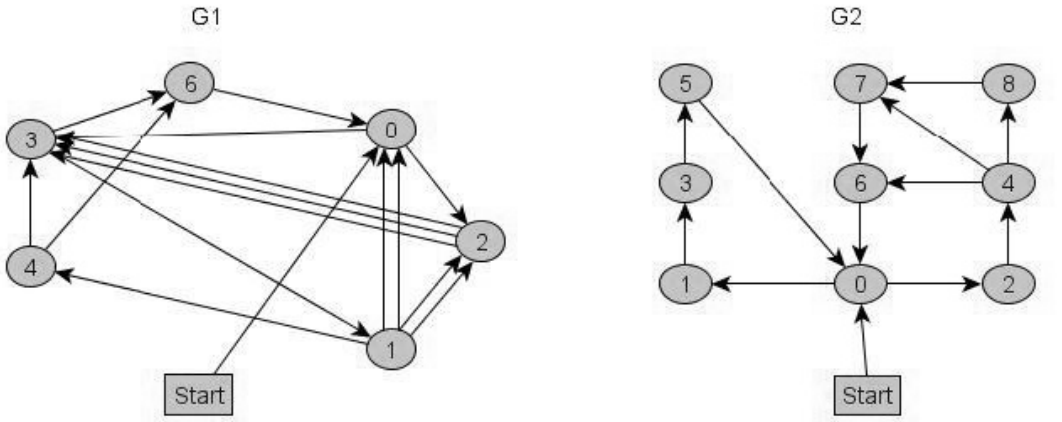

Рис. 7. Тестовые графы

Были получены следующие результаты:

Путь для графа G1: [Start,0,2,3,1,4,6,0,3,6,0,2,3,1,2,3,1,2,3,1,0,2,3,1,0,2,3,1,4,3]

Эффективность равна 64%

Путь для графа G2: [Start,0,2,4,8,7,6,0,1,3,5,0,2,4,7,6,0,1,2,4,6]

Эффективность равна 50% [6]

Следует отметить, что в проведенном исследовании эффективность Eff является относительной величиной и позволяет сравнивать работу алгоритма для различных КА относительно друг друга. 100%-ная эффективность достигается в том случае, если количество посещений каждого узла равно количеству исходящих или входящих ветвей в графе КА (в зависимости от того – что больше). Очевидно, что она достигается только в случае существования эйлерова пути на графе. Поэтому данная оценка априори является заниженной для произвольных графов.

Заключение

В статье описан метод тестирования пользовательского интерфейса «мобильных» приложений, основанный на применении расширенных конечных автоматов. Суть метода заключается в выделении состояний конечного автомата путем декомпозиции приложения и последующей генерации тестов с применением алгоритма траверса графов. Предлагается одна из реализаций алгоритма генерации тестового набора. Тестирование приложений проводится по следующей общей схеме.

Благодаря использованию алгоритма A^*, результирующий тестовый набор является минимальным в том смысле, что он позволяет протестировать приложение с наименьшим количеством действий. Благодаря использованию правил перехода между видами,

достигается полное покрытие по классам эквивалентности входных данных, обеспечивается необходимая детальность модели. Введение фиктивных переходов позволяет требовать 100% покрытия переходов, а также увеличить простоту модели и её тестируемость в случае ручного тестирования. Детальность и введение фиктивных переходов обеспечивают автоматизируемость модели.

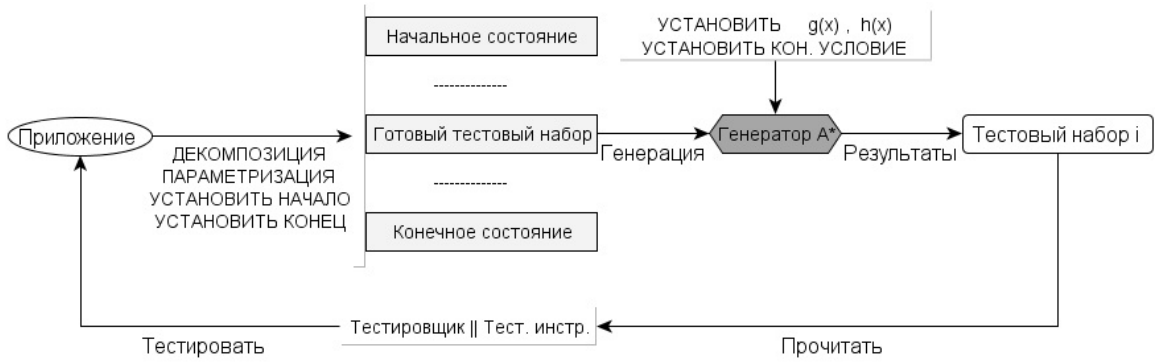

Рис. 8. Общая схема тестирования

Направления дальнейших исследований

Предложенный в статье генератор не позволяет автоматически проводить декомпозицию приложения, предложенное разбиение видов на ViewEnter и ViewEntered эффективно, но приводит к усложнению модели, нужно найти компромисс между сложной моделью с простыми состояниями и простой моделью со сложными состояниями. Также в генерируемых тестах не указано – когда и как делать проверку результатов.

Целесообразны дальнейшие исследования в следующих направлениях:

1) Обеспечение тестируемости средствами генератора, т.е. реализация автоматической декомпозиции приложения.

2) Добавление информации о способах проверки результатов в модель.

3) Автоматическое определение состояний, которые нужно проверять.

4) Разработка тестового генератора, способного работать с описанными моделями.

5) Генерация тестов готовых для автоматического выполнения без дополнительной подготовки.

Литература

1. Computer Programming Software Terms, Glossary and Dictionary. [Электронный ресурс] - http://www.networkdictionary.com/software/e.php?PHPSESSID=9926acc9b2e4f8f05ced05a620abcfe9. Retrieved 2006-12-13.
2. *Кулямин В.* Тестирование на основе моделей. - [Электронный ресурс] - http://panda.ispras.ru/~kuliamin/lectures-mbt/Lecture04.pdf
3. *Lester P.* Алгоритм A* для новичков. (перевод), [Электронный ресурс] - http://www.policyalmanac.org/games/aStarTutorial_rus.htm.
4. *Patel A.* The A* Algorithm. [Электронный ресурс] - http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#S3.
5. *Edmonds J.* Matching, Euler tours and the Chinese postman. - PlaceTypeUniversity of PlaceNameWaterloo, 1972, placecountry-regionCanada.
6. *Хатько Е.Е.* Один способ реализации алгоритма генерации тестов в тестировании на основе моделей. – Труды 53 науч. конф. МФТИ. «Современ. проблемы фунд. и приклад. наук.» Ч. 1. – М.: МФТИ, 2010. – С. 92–94.

Поступила в редакцию 03.12.2011.