Моделирование и анализ информационных систем. Т. 23, №6 (2016), с. 729–740
Modeling and Analysis of Information Systems. Vol. 23, No 6 (2016), pp. 729–740

DOI: 10.18255/1818-1015-2016-6-729-740

УДК 004.415.53

Метод синтеза тестов с гарантированной полнотой по модели расширенного автомата

Ермаков А.Д., Евтушенко Н.В.[1]

получена 2 сентября 2016

Аннотация. Расширенные автоматы активно используются при построении тестов для программного обеспечения на основе формальных моделей. Однако полнота тестов, построенных по расширенному автомату на основе покрытия путей, переменных и т.п., остается практически неизвестной; более того, как известно, такие тесты не обнаруживают большое количество часто встречающихся функциональных ошибок в программных реализациях системы, поведение которой описано таким расширенным автоматом. В данной работе для построения тестовых последовательностей мы предлагаем использовать шаблонную реализацию расширенного автомата в языке Java. Поскольку программа составлена по шаблону, то ошибки в программе напрямую переносятся на ошибки в расширенном автомате. В работе предлагается метод построения множества тестовых последовательностей, обнаруживающих функциональные ошибки в шаблонной реализации расширенного автомата. На первом шаге тест, построенный по расширенному автомату одним из известных методов, проверяется на полноту относительно ошибок, сгенерированных инструментом μJava в шаблонной программной реализации. После этого для каждого необнаруженного тестом программного мутанта строится мутант эталонного расширенного автомата; на следующем шаге по некоторой конечно-автоматной абстракции генерируется последовательность, различающая два расширенных автомата (если такая последовательность существует), которая добавляется в строящийся тест. Построенный таким образом тест является полным относительно ошибок, сгенерированных инструментом μJava. Если соответствующий конечный автомат, построенный посредством моделирования расширенного автомата, получается слишком сложным, или построить такой конечный автомат не представляется возможным, то полнота построенного теста не гарантируется. Однако экспериментально показывается, что исходный тест, расширенный такими различающими последовательностями, обнаруживает значительно больше функциональных ошибок в программных реализациях системы, для которой расширенный автомат используется в качестве спецификаци.

Ключевые слова: мутационное тестирование, расширенный автомат, конечно-автоматная абстракция, μJava

Для цитирования: Ермаков А. Д., Евтушенко Н.В., "Метод синтеза тестов с гарантированной полнотой по модели расширенного автомата", *Моделирование и анализ информационных систем*, **23**:6 (2016), 729–740.

Об авторах:
Ермаков Антон Дмитриевич, orcid.org/0000-0003-0838-8014, аспирант,
Томский государственный университет,
пр. Ленина, 36, г. Томск, 634050 Россия, e-mail: antonermak@inbox.ru

Евтушенко Нина Владимировна, orcid.org/0002-0007-1896-0876, профессор, д-р. техн. наук,
Томский государственный университет,
пр. Ленина, 36, г. Томск, 634050 Россия, e-mail: nyevtush@gmail.com

Благодарности:
[1]Работа выполнена при финансовой поддержке проекта РНФ № 16-49-03012.

Введение

Программные реализации, используемые, в том числе, в критических системах, становятся более сложными, и гарантировать полноту их тестирования без использования математических моделей практически невозможно. При синтезе тестов с гарантированной полнотой активно используются модели с конечным числом переходов [1,2], в частности модель расширенного автомата, которая достаточно близка к программной реализации в языке высокого уровня. Несмотря на большое количество публикаций об автоматическом построении такой модели, в опубликованных статьях все примеры обычно ограничиваются небольшими программами, и достаточно часто такой расширенный автомат строится инженером по тестированию «вручную», исходя из неформальных требований к функционированию тестируемой программы. Соответственно, практически невозможно сопоставить ошибки в построенной модели, относительно которых и будет гарантирована полнота построенного теста, с ошибками в исходной программе, и для генерации тестовой последовательности, обнаруживающей такую функциональную ошибку, приходится сравнивать программу-спецификацию с программой-мутантом для построения соответствующей различающей последовательности [3,4]. В настоящей работе мы предлагаем метод построения по расширенному автомату множества тестовых последовательностей, которые обнаруживают все функциональные ошибки в шаблонной реализации расширенного автомата в языке Java.

Расширенный автомат [5] расширяет классическую автоматную модель внутренними (контекстными) переменными, входными и выходными параметрами. Известно, что тесты, построенные на основе таких расширенных систем переходов, оказываются достаточно качественными, однако остается много функциональных ошибок в тестируемых программных реализациях, которые построенными тестами не обнаруживаются [4,6]. Поэтому мы предлагаем повысить полноту тестов, построенных по расширенному автомату, рассматривая наиболее часто встречаемые ошибки в программных реализациях, на основе «шаблонной» реализации расширенного автомата в языке Java. С помощью инструмента μJava [7] генерируется множество мутантов для «шаблонной» программной реализации, на которые подается проверяющий тест, построенный к настоящему моменту; первоначальный тест может быть построен одним из известных методов, в том числе это может быть множество последовательностей, сгенерированных случайным образом. Если мутант тестом не обнаруживается, то соответствующая ошибка легко преобразуется в ошибку в расширенном автомате, и, таким образом, различающая последовательность строится не для двух программных реализаций, что, как известно, достаточно сложно [4], а для двух автоматных моделей, что значительно проще [8]. Частично результаты по подобному подходу были опубликованы в [9]; в настоящей работе мы обсуждаем возможности использования предложенного подхода для обнаружения функциональных ошибок в расширенных автоматах.

Структура работы следующая. В разделе 1 вводятся необходимые определения и обозначения. В разделе 2 обсуждаются способы построения различающих последовательностей для двух расширенных автоматов, в нашем случае для автомата-спецификации и интересующего нас мутанта. Предлагаемый метод построения проверяющего теста описывается в разделе 3. В разделе 4 рассматривается пример те-

стирования программной реализации протокола SCP (Simple Connection Protocol), для которого и иллюстрируется предлагаемый подход. В заключении мы кратко обсуждаем возможные направления дальнейшей работы.

1. Определения и обозначения

Под *конечным автоматом* понимается пятёрка $S = (S,\ I,\ O,\ T_S,\ s_0)$, где S – непустое конечное множество *состояний* с выделенным начальным состоянием s_0, I – непустое конечное множество *входных* символов, называемое входным алфавитом, O – непустое конечное множество *выходных* символов, называемое выходным алфавитом, $T_S \subseteq I \times S \times S \times O$ – отношение *переходов* [8]. Расширенный автомат дополняет конечный автомат контекстными (внутренними) переменными, входными и выходными параметрами, и условиями, при которых переход может быть выполнен. Формально [5] под расширенным автоматом M понимается пятерка $M = (S,\ s_0,\ X,\ Y,\ T,\ V)$, где S – непустое конечное множество состояний автомата, X – непустое конечное множество входных символов, Y – непустое конечное множество выходных символов, V – конечное, возможно, пустое множество контекстных переменных, T – множество переходов между состояниями из S. Каждый переход в расширенном автомате есть семёрка $(s,\ x,\ P,\ o_M,\ y,\ u_M,\ s')$, где s и s' суть начальное и финальное состояния перехода; $x \in X$ есть входной символ и D_{inp-x} обозначает множество входных векторов, компонентами которых являются значения параметров, соответствующих входному символу x (далее *входные параметры*); $y \in Y$ – выходной символ, и D_{out-y} обозначает множество выходных векторов, компонентами которых являются значения параметров, соответствующих выходному символу y (далее *выходные параметры*); P, o_M и u_M – функции, определенные над входными параметрами и контекстными переменными из V. Предикат P: $D_{inp-x} \times D_V \to \{0,\ 1\}$, где D_V – множество контекстных векторов, то есть векторов, компонентами которых являются значения контекстных переменных, описывает условия, при которых данный переход может быть выполнен; функция $o_M : D_{inp-x} \times D_V \to D_{out-y}$ описывает значения выходных параметров в результате выполнения перехода; функция $u_M : D_{inp-x} \times D_V \to D_V$ описывает значения контекстных переменных в результате выполнения перехода.

Пара «состояние, вектор значений контекстных переменных» называется *конфигурацией*, пары «входной символ, вектор значений входных параметров» и «выходной символ, вектор значений выходных параметров» называются *параметризированными* входным и выходным символами соответственно. Начальная конфигурация расширенного автомата обычно предполагается известной. Переход в расширенном автомате может быть выполнен, если только соответствующий предикат принимает значение «Истина» в данной конфигурации на данном параметризированном входном символе. Таким образом, в отличие от классических систем с конечным числом состояний, в расширенном автомате не каждый переход может быть выполнен в текущем состоянии (хорошо известная проблема выполнимости последовательности переходов в расширенном автомате). Вполне возможно, что для выполнения конкретного перехода понадобится сначала выполнить ряд других переходов, на-

пример, для достижения переменной-счетчиком нужного значения, и только потом можно будет выполнить требуемый переход.

Под *функциональными ошибками* в расширенном автомате понимаются ошибки переходов / выходов, присвоения значений переменным, ошибки в предикатах и т.п. Известные методы построения проверяющих тестов по модели расширенного автомата ориентируются на покрытие путей, переменных, условий и т.п. [4, 6], но как показано в диссертации С. Ники [4], полнота таких тестов относительно функциональных ошибок очень низкая, не превышает 70 %. На основе компьютерных экспериментов с различными протокольными реализациями в [6] показывается, что полнота достаточно длинных случайных тестов относительно функциональных ошибок практически совпадает с полнотой тестов, построенных на основе покрытия множеств различных переходов расширенного автомата.

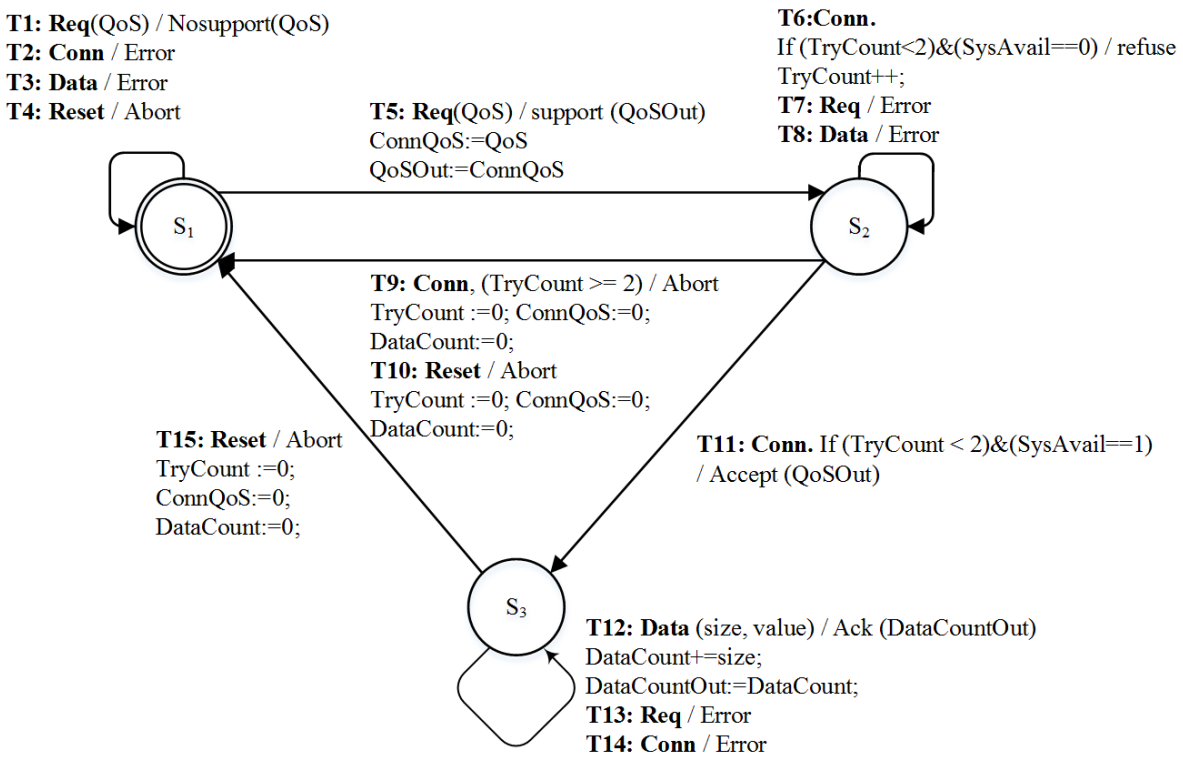

Рис. 1. Расширенный автомат для SCP

Fig. 1. EFSM for SCP

В качестве примера рассмотрим расширенный автомат, представляющий собой описание протокола SCP (Simple Connection Protocol) [10,11]; расширенный автомат (рис. 1) имеет 3 состояния, которые, вообще говоря, описывают различные режимы функционирования. Состояние S_1 описывает режим ожидания запроса на соединение, S_2 соответствует состоянию процесса установки соединения, а состояние S_3 соответствует передаче данных. Входные символы описывают стандартные команды протокола: *Req* – запрос, *Conn* – подключение, *Data* – передача данных, *Reset* – сброс; мы также используем входной параметр *Support*, который равен 1 в случае готовности установить соединение уровня QoS, и 0 при отсутствии готовности. Вход-

ной параметр *SysAvail* соответствует 1, если система свободна для подключения, и 0, если занята. Выходные параметры суть *Nosupport, Error, Abort, Support, Refuse, Accept, Ack*. Контекстная переменная *TryCount* соответствует счетчику неудачных попыток установки соединения. Несмотря на то, что этот протокол является в некоторой степени «игрушечным», он достаточно хорошо иллюстрирует многие аспекты протокольных реализаций.

Заметим, что, в отличие от построения различающих последовательностей для мутантов программного обеспечения, построение различающих последовательностей для конечных автоматов намного проще, и в следующем разделе мы кратко обсуждаем известные методы построения различающей последовательности для двух расширенных автоматов на основе различных конечно-автоматных абстракций, отмечая, для каких функциональных ошибок какие абстракции являются наиболее подходящими.

2. Построение различающих последовательностей для расширенных автоматов

В настоящем разделе мы обсуждаем, каким образом можно построить различающую последовательность для двух расширенных автоматов. Реализация общего метода из [5], который гарантирует построение различающей последовательности для двух неэквивалентных детерминированных расширенных автоматов, является достаточно трудоемкой, и отметим, авторы ничего не говорят о решении проблемы выполнимости. Все остальные методы построения различающих последовательностей для расширенных автоматов являются достаточно эффективными эвристиками, использующими конечно-автоматные абстракции [11–15], для которых отсутствует проблема выполнимости, и построение различающей последовательности достаточно просто осуществляется на основе построения пересечения двух конечных автоматов, которые могут быть детерминированными или недетерминированными, частичными или полностью определенными.

Построение различающей последовательности на основе пересечения двух расширенных автоматов. В [5] рассматривается задача построения различающей последовательности для двух различных расширенных автоматов; в нашем случае спецификации и мутанта. Для описания всех возможных различающих последовательностей строится *различающий автомат*, в котором специально введенное состояние *fail* представляет все последовательности, различающие начальные конфигурации автомата-спецификации и автомата-мутанта, не различимого со спецификацией уже построенным тестом. Поскольку рассматривается различимость двух расширенных автоматов, в которых множества контекстных переменных не пересекаются, в нашем случае контекстные переменные мутанта необходимо переименовать. В этом случае все различающие последовательности представляются специальным состоянием *fail*, и тем самым задача построения различающей последовательности сводится к хорошо изученной задаче достижимости. С практической точки зрения использование такого различающего автомата имеет ряд недостатков. Во-первых, речь идет только о детерминированных полностью определенных спецификациях, что не всегда имеет место, в частности, для протокольных

спецификаций. Во-вторых, не всякая последовательность переходов, переводящая различающий автомат в состояние *fail*, выполнима, т.е. проблема выполнимости в работе не решается. В-третьих, остается вопрос о равенстве параметризированных выходных символов в случае, когда такие символы являются даже простыми функциями, а не просто назначаются равными некоторой константе. Таким образом, несмотря на общность предлагаемого подхода, использование различающего автомата-пересечения для построения различающих последовательностей для двух расширенных автоматов является достаточно трудоемким.

Построение различающей последовательности на основе конечно-автоматных абстракций двух расширенных автоматов. Наиболее известными можно считать следующие две конечно-автоматные абстракции для расширенных автоматов. В первом случае поведение расширенного автомата в начальной конфигурации моделируется на входных параметризированных последовательностях до достижения в конечном автомате заданного числа состояний или на всех таких последовательностях длины не более l. Если известно, на каком переходе отличаются автомат-спецификация и исследуемый автомат-мутант, то на каждом шаге моделирования можно выбирать следующий переход с использованием некоторого «жадного» алгоритма, приближаясь как можно ближе к мутированному переходу. Проведенные эксперименты показывают, что при условии, что два расширенных автомата отличаются между собой в небольшом количестве переходов (один-два мутированных перехода в спецификации), обычно достаточно рассматривать $l = 2$, 3 для построения различающей последовательности. Кроме того, при достаточно большом ограничении на число состояний обход графа переходов соответствующего конечного автомата обнаруживает одиночные ошибки в предикатах и функциях для контекстных переменных и выходных параметров [13].

Во втором случае из расширенного автомата просто удаляются все предикаты, входные и выходные параметры и соответствующие функции. Как показано в [14], в этом случае (адаптивная) различающая последовательность строится для двух недетерминированных, возможно ненаблюдаемых автоматов. Методы построения различающих последовательностей для таких автоматов существуют [14,15], и как показывают проведенные эксперименты, несмотря на экспоненциальные общие верхние оценки длины таких последовательностей, в большинстве случаев длина таких последовательностей близка к числу состояний автомата. Более того, адаптивные различающие последовательности существуют чаще и оказываются обычно короче; однако в этом случае и проверяющие тесты также должны быть адаптивными. Следует отметить, что поскольку в такой абстракции удаляются все предикаты, контекстные переменные, входные и выходные параметры, то такая абстракция может быть эффективно использована только при построении различающих последовательностей для ошибок в выходных символах и переменных состояний. Например, для расширенного автомата на рис. 1 ошибка в финальном состоянии перехода T5 (вместо S_2 состояние S_1) немедленно обнаруживается при подаче входного символа *Req*.

Построение различающей последовательности на основе предикатных абстракций двух расширенных автоматов. Недетерминированные автоматы появляются и при рассмотрении предикатных абстракций расширенных автома-

тов [16]; при построении различающих последовательностей такие абстракции оказываются наиболее эффективными при отсутствии входных параметров.

Пусть β есть множество k предикатов, определенных относительно контекстных переменных и входных параметров расширенного автомата M. Если S – множество состояний расширенного автомата и D_W – множество всех конфигураций и параметризированных входных векторов, то предикатная β-абстракция определяется как a_β: $S \times D_W \to S \times \{(0,1)\}^k$, где $a_\beta(s, \mathbf{w}) = (s, b_1, \ldots, b_k)$, $\mathbf{w} \in D_W, b_i \in \{0,1\}, i = 1..k$. По определению, $b_i = 1$, если и только если предикат $B_i(\mathbf{w})$ принимает значение «Истина». Для выбранного множества $\mathbf{P_x}$ (параметризированных) входных символов множество входных символов предикатной абстракции содержит пары $(x, \mathbf{p_x}), \mathbf{p_x} \in \mathbf{P_x}$, и все непараметризированные входные символы. Для каждой конфигурации $(s, \mathbf{v})$, входного символа $(x, \mathbf{p_x}) \in \mathbf{P_x}$ и предиката P перехода (s, x, P, y, u_p, s') из состояния s, такого что $(\mathbf{v}, \mathbf{p_x})$ обращает P в истину и $u_M(\mathbf{v}) = \mathbf{v}'$, в множество переходов предикатной абстракции включается переход $((s, \mathbf{a}), x, y, (s', \mathbf{a}'))$, $(s, \mathbf{a}) = a_\beta(s, \mathbf{v})$ и $(s', \mathbf{a}') = a_\beta(s', \mathbf{v}')$. В качестве примера рассмотрим предикатную абстракцию для расширенного автомата на рис. 1 относительно множества предикатов $(TryCount < 2)$ и $(SysAvail = 0)$. При ошибке в условии $(SysAvail = 0)$ предикатная абстракция не останется в состоянии S_2, а из состояния S_2 перейдет в состояние S_3, что достаточно просто обнаружится при подаче входного символа *Data*.

В общем случае предикатная абстракция является недетерминированным автоматом, в котором степень недетерминизма существенно зависит от выбора множества предикатов и множества параметризированных входных символов. Упрощая утверждение из работы [16] для случая различимости начальных конфигураций двух расширенных автоматов, получаем, что два расширенных автомата различимы (адаптивной) входной последовательностью α, если начальные состояния соответствующих предикатных абстракций разделимы последовательностью α (адаптивно различимы), т.е. множества выходных реакций на α в этих состояниях не пересекаются (существует адаптивный различающий тестовый пример [15]). В работе [16] обсуждаются возможности выбора предикатов (для построения предикатной абстракции) с целью уменьшения недетерминизма, чтобы увеличить вероятность существования разделяющей последовательности. Кроме того, можно строить адаптивную различающую последовательность, т.е. говорить об адаптивной различимости двух расширенных автоматов. Насколько нам известно, адаптивные различающие последовательности для расширенных автоматов нигде не исследовались.

Отметим, что процесс построения предикатной абстракции достаточно трудоемкий, поскольку исследуются конфигурации расширенного автомата. Однако при наличии ошибки только на одном переходе (как получается при использовании инструмента μJava), можно просто отметить переход с ошибкой в предикатной абстракции спецификации, т.е. не строить заново предикатную абстракцию для автомата-мутанта.

Построение различающей последовательности на основе различных срезов двух расширенных автоматов достаточно подробно обсуждается в работах [13,17]. Как показывают проведенные авторами эксперименты, тесты, построенные как обход графа переходов таких срезов, достаточно хорошо обнаруживают одиночные функциональные ошибки, такие как ошибки переходов и выходов,

а также ошибки типа присвоения неправильного значения контекстной переменной/выходному параметру или замены некоторой арифметической/логической операции.

3. Метод построения проверяющего теста с использованием инструмента μJava

Проверяющие тесты, построенные по расширенному автомату одним из известных (достаточно простых) методов, дополняются различающими последовательностями для соответствующих мутантов, которые строятся с использованием специального инструмента μJava [7]. Инструмент μJava имеет достаточно обширные функциональные возможности и согласно документации способен генерировать 34 вида мутаций программного кода, среди которых выделяют традиционные ошибки (замена математических, логических операторов, операторов сравнения и т.д.) и ошибки объектно-ориентированного программирования (ошибки наследования, полиморфизма, и т.д.), достаточно хорошо соответствующие функциональным ошибкам программного обеспечения. Последнее и является основной причиной выбора такого подхода для повышения полноты тестов, построенных по расширенному автомату. Поскольку известно, что наиболее трудно обнаружимыми являются одиночные ошибки, то именно такие мутанты программных реализаций и рассматриваются. Для генерации мутаций необходимо запустить графическую оболочку μJava, выбрать проект программы и типы мутаций для генерации. В результате в директории Results появятся все сгенерированные мутанты в поддиректориях с именами, соответствующими типу мутации и её порядковому номеру. С использованием библиотеки для модульного тестирования JUnit [18] тест может быть подан сразу на все мутанты с целью определения, какие из мутантов не обнаруживаются тестом. Таким образом, построение теста по расширенному автомату с использованием инструмента μJava содержит следующие шаги.

Шаг 1. Первоначальный проверяющий тест TS строится одним из известных методов по расширенному автомату-спецификации M. Можно использовать обход графа переходов соответствующего расширенного автомата, различные методы покрытия путей, переменных, условий и т.п., а также случайно сгенерированные тесты определенной длины.

Шаг 2. По заранее определенному шаблону строится программная реализация расширенного автомата-спецификации таким образом, что ошибки программной реализации жестко связаны с ошибками в расширенном автомате. В частности, в программной реализации состояния расширенного автомата используются как метки для описания соответствующего режима работы. Контекстные переменные, входные и выходные параметры соответствуют таковым в программной реализации; предикаты описывают условия выполнения инструкций.

Шаг 3. Проверяющий тест TS проверяется на полноту для построенной программной реализации с использованием ошибок, вносимых генератором инструмента μJava. Те ошибки, которые не были обнаружены, вносятся в расширенный автомат, и строится множество Mut расширенных автоматов-мутантов, не обнаружимых тестом TS.

Шаг 4. Для каждого автомата *Imp* из множества *Mut* строится подходящая конечно-автоматная абстракция и определяется последовательность, различающая конечно-автоматные абстракции двух расширенных автоматов, автомата-спецификации и автомата *Imp*. Если такая последовательность существует, то она добавляется в тест *TS*; при отсутствии такой последовательности делается вывод о неразличимости текущего мутанта и спецификации.

Утверждение 1. Если существуют детерминированные конечные автоматы, моделирующие поведение расширенных автомата-спецификации и каждого построенного мутанта, то алгоритм, содержащий вышеописанные шаги, возвращает полный проверяющий тест относительно квазиэквивалентности, т.е. построенный тест обнаруживает всякий мутант, поведение которого отличается от спецификации на некоторой определенной в спецификации (параметризированной) входной последовательности.

В ряде случаев для расширенного автомата-спецификации или некоторого мутанта такой конечный автомат нельзя построить в силу вычислительных проблем или, например, бесконечного множества допустимых значений для некоторой контекстной переменной или входного параметра. Соответственно мы не можем гарантировать различимость мутанта и спецификации в случае ненахождения различающей последовательности, но, как показывают эксперименты, проведенные с рядом расширенных автоматов, описывающих поведение протоколов и технических систем, такие ситуации встречаются достаточно редко. Если построенные автоматы являются недетерминированными, то полнота построенного теста определяется не относительно эквивалентности, а относительно отношений неразделимости или адаптивной неразличимости (при использовании адаптивной различающей последовательности). Мы подчеркиваем, что для произвольных расширенных автоматов отсутствуют необходимые и достаточные условия для проверки отношений эквивалентности, редукции, разделимости и адаптивной различимости двух автоматов. В наших экспериментах рассматривались достаточно простые расширенные автоматы; поэтому для проверки этих отношений мы использовали только достаточные условия. Одним из таких условий является внесение ошибки, при которой появляется дополнительная компонента связности в эталонном расширенном автомате, причем в исходной компоненте связности не происходит никаких изменений.

4. Анализ проведенных экспериментов на примере SCP

Эксперименты проводились с расширенными автоматами, описывающими такие протоколы как SCP, «Time» protocol, SMTP, POP3, TFTP, калькулятор, Audio CD плеер. Практически во всех случаях, за исключением самых простых протоколов, исходный обход графа переходов расширенного автомата приходилось доопределять различающими последовательностями. Более детально мы иллюстрируем процесс для протокола SCP (Simple Connection Protocol), для которого расширенный автомат был реализован программно, и в качестве начального теста использовался обход графа переходов. В результате запуска инструмента μJava на программной

реализации было сгенерировано 245 традиционных (арифметических) мутантов и 7 мутантов объектного типа (см. таблицу 1).

Таблица 1. Сгенерированные мутанты

Table 1. Generated mutants

Наименование Name	Описание мутантов Mutant description	Количество мутантов Number of mutants
AOIS	Инкремент/декремент случайной переменной	96
AOIU	Отрицание переменной	5
LOI	Побитовое отрицание	24
ROR	Замена знаков операторов сравнения >, <, =, <=, >=, ==	91
COR	Замена знаков логических операторов ^, ||, &&, &, |&	4
COI	Внедрение в код логического отрицания условий !(true), !(false)	17
ASRS	Модификация операций составного присваивания +=, /=, -=, %=	8
JSI	Добавление служебного слова Static в объявлении член-данных класса	7
Всего		252

Запуск начального теста *TS* на данных мутантах показал, что 62 мутанта (24,6%) сработали идентично исходной программе. С использованием конечно-автоматных абстракций выяснилось, что 9 (3,6%) из них не эквивалентны спецификации. Остальные 53 (21%) мутантов не вносили изменений в поведение расширенного автомата-спецификации. Далее, путем возврата к расширенному автомату были выделены переходы, на которых произошли неэквивалентные мутации. Благодаря этому тест был достроен путем добавления трех различающих последовательностей параметризированных входных символов суммарной длины 11. Таким образом, длина теста увеличилась с 18 до 29 входных символов. Повторный запуск теста различил 9 неэквивалентных мутантов от спецификации. Таким образом, неотличимыми остались только 53 эквивалентных мутанта. В результате полнота теста выросла с 75,4% до 100 % (относительно мутантов, сгенерированных с использованием инструмента μJava).

5. Заключение

В данной работе мы предложили метод повышения полноты тестов, построенных по расширенному автомату, который описывает поведение программной реализации, за счет использования мутационного тестирования для программной реализации,

выполненной по специальному шаблону. Тесты, построенные как обход графа переходов спецификации, оказались неполными относительно ошибок, вносимых инструментом μJava, т.е. были функциональные ошибки, которые не обнаруживались таким тестом. Причина этого, в первую очередь, заключается в том, что расширенный автомат-спецификация чаще всего строится не непосредственно по тестируемой программной реализации, а на основе некоторых неформально описанных требований к такой реализации. Для полного обнаружения ошибок, вносимых инструментом μJava, тест дополнялся различающими последовательностями для конечно-автоматных абстракций спецификации и ее мутанта. Как показывают проведенные эксперименты, такой подход с использованием конечно-автоматных абстракций позволил выявить мутанты, эквивалентные спецификации, а также достроить проверяющий тест различающими последовательностями для неэквивалентных мутантов. Интересным вопросом является исследование возможности введения наблюдаемости для контекстных переменных на основе такого подхода. Если некоторые мутации контекстных переменных не обнаруживаются при использовании конечно-автоматных абстракций, то, возможно, имеет смысл сделать эти переменные наблюдаемыми при отладке программы, т.е. объявить их выходными параметрами. Построение различающих последовательностей дает возможность минимизировать список таких наблюдаемых переменных. Особый интерес представляет построение различающих последовательностей для автоматов специальных классов, например, древовидных и временных автоматов. Еще одно направление нашей дальнейшей работы относится к использованию полученных результатов не только для обнаружения, но и для локализации ошибок в программных реализациях.

Список литературы / References

[1] Kaur M., Singh R., “A Review of Software Testing Techniques”, *International Journal of Electronic and Electrical Engineering*, **7**:5 (2014), 463–474.

[2] Jorgensen P.C., *Software Testing: A Craftsman’s Approach, Third Edition*, Auerbach Publications, 2008.

[3] Nica M., Nica S., Wotawa F., “On the use of mutations and testing for debugging”, *Software – practice and experience*, **43**:9 (2013), 1121–1142.

[4] Nica S., *On the Use of Constraints in Program Mutations and its Applicability to Testing*, PhD thesis, Graz Technical University, 2013.

[5] Petrenko A., Boroday S., Groz R., “Confirming Configurations in EFSM Testing”, *IEEE Trans. Software Eng*, **30**:1 (2004), 29–42.

[6] El-Fakih K., Salameh T., Yevtushenko N., “On Code Coverage of Extended FSM Based Test Suites: An Initial Assessment”, *LNCS*, **8763** (2014), 198–204.

[7] “μJava documentation. μJava home page. URL: http://cs.gmu.edu/~offutt/mujava/”, 2014, (access date: 10.04.2016).

[8] Villa, T. et al., *The Unknown Component Problem: Theory and Applications*, Springer, 2012, 313 pp.

[9] Ermakov A., Yevtushenko N., “Increasing the fault coverage of tests derived against Extended Finite State Machines”, *System informatics*, **7** (2016), 23–32.

[10] Alcalde B. et al., “Network Protocol System Passive Testing for Fault Management: A Backward Checking Approach”, *Proc. of the 24th IFIP WG 6.1 Intern. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE’2004*, 150–166.

[11] Kushik N. et al., "Optimizing Protocol Passive Testing through 'Gedanken' Experiments with Finite State Machines", *Proc. of the Intern conference on Sofware Quality and Reliability, QSR'2016*, August, 2016.

[12] Kushik N., Yenigün H., "Heuristics for Deriving Adaptive Homing and Distinguishing Sequences for Nondeterministic Finite State Machines", *Lecture Notes in Computer Science*, **9447** (2015), 243–248.

[13] Коломеец А. В., *Алгоритмы синтеза проверяющих тестов для управляющих систем на основе расширенных автоматов*, дис. ... канд. техн. наук, Томский государственный университет, 2010; [Kolomeec A. V., *Algoritmy sinteza proverjajushhih testov dlja upravljajushhih sistem na osnove rasshirennyh avtomatov*, dis. ... kand. tekhn. nauk, Tomsk State University, 2010, (in Russian).]

[14] Kushik N., Yevtushenko N., Cavalli A., "On Testing against Partial Non-observable Specifications", *Proc. of the Intern. Conf. on the Quality of Information and Communications Technology*, 2014, 230–233.

[15] Kushik N. et al., "On adaptive experiments for nondeterministic finite state machines", *The Intern. Journal on Software Tools for Technology Transfer*, **18**:3 (2016), 251–264.

[16] El-Fakih K. et al., "Distinguishing extended finite state machine configurations using predicate abstractions", *Journal on Software Research and Development (published online)*, 2016.

[17] Михайлов Ю. В., Коломеец А. В., "Проверка переходов в расширенном автомате на основе срезов", *Вестник ТГУ. Серия: Управление, вычислительная техника и информатика*, **3**:4 (2008); [Mikhaylov Yu. V., Kolomeets A. V., "Proverka perekhodov v rasshirennom avtomate na osnove srezov", *Vestnik TGU. Seriya: Upravlenie, vychislitel'naya tekhnika i informatika*, **3**:4 (2008), (in Russian).]

[18] "JUnit 4, documentation. URL: http://cs.gmu.edu/~offutt/mujava/", (access date: 10.04.2016).

Ermakov A. D., Yevtushenko N. V., "Deriving Test Suites with the Guaranteed Fault Coverage for Extended Finite State Machines", *Modeling and Analysis of Information Systems*, **23**:6 (2016), 729–740.

DOI: 10.18255/1818-1015-2016-6-729-740

Abstract. Extended Finite State Machines (EFSMs) are widely used when deriving tests for checking functional requirements for software implementations. However, the fault coverage of tests covering appropriate paths, variables, etc. of the specification EFSM, remains rather obscure and such tests do not detect many functional faults in EFSM implementations. In this paper, an approach is proposed for deriving complete tests with respect to functional faults of a proper Java EFSM implementation. First, an initial test suite derived against the specification EFSM is checked with respect to faults generated by a μJava tool. Since the EFSM software implementation is template based, each undetected fault can be easily mapped into a mutant EFSM of the specification machine. Thus, a distinguishing sequence is derived for two Finite State Machines modeling two EFSMs instead of deriving such a sequence for two programs. If the corresponding FSMs are too complex or cannot be completely derived, a test suite can be incomplete. However, the performed experiments clearly show that a test suite extended by such distinguishing sequences detects much more functional faults in software implementations of a system whose behaviour is described by the given EFSM.

Keywords: Mutation testing, Extended Finite State Machine (EFSM), finite automata, μJava

About the authors:
Anton D. Ermakov, orcid.org/0002-0007-1896-0951, researcher, Tomsk State University,
36 Lenina ave., Tomsk 634050, Russia, e-mail: antonermak@inbox.ru

Nina V. Yevtushenko, orcid.org/0000-0002-4006-1161, Professor, Doctor of Technical Sciences, Tomsk State University,
36 Lenina ave., Tomsk 634050, Russia, e-mail: nyevtush@gmail.com

Acknowledgments:
This work was supported by the project of RSF № 16-49-03012.