Министерство образования и науки Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Радиофизический факультет
Кафедра информационных технологий в исследовании дискретных структур (ИТИДиС)

ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК
Руководитель ООП
д-р физ.-мат. наук, профессор
_____ Н.В. Евтушенко
«13» 06 2016 г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ КОНЕЧНО АВТОМАТНЫХ МЕТОДОВ ТЕСТИРОВАНИЯ В КОНТЕКСТЕ ДЛЯ ТЕЛЕКОММУНИКАЦИОННЫХ ПРОТОКОЛОВ

по основной образовательной программе подготовки магистров
направление подготовки 03.04.03 – Радиофизика

Щипачев Никита Михайлович

Научный руководитель ВКР
канд. физ.-мат. наук, доцент
_____ Н. Г. Кушик
подпись
«13» 06 2016 г.

Автор работы
студент группы № 07700
_____ Н. М. Щипачев
подпись

Томск-2016

РЕФЕРАТ

Магистерская диссертация 58 с., 5 гл., 20 рис., 1 табл., 43 источника.

ТЕЛЕКОММУНИКАЦИОННЫЙ ПРОТОКОЛ, TCP, ТЕСТИРОВАНИЕ В КОНТЕКСТЕ, КОНЕЧНЫЙ АВТОМАТ.

Диссертация посвящена экспериментальному исследованию конечно автоматных методов тестирования в контексте для телекоммуникационных протоколов. В работе проведен обзор методов и средств тестирования программного обеспечения, а также рассмотрены возможные топологии тестирования в контексте по критерию наблюдаемости и управляемости проверяемой компоненты. Конечно автоматные методы синтеза проверяющих последовательностей позволяют гарантировать полноту тестирования, тем не менее конечно автоматные методы тестирования в контексте для телекоммуникационных протоколов требуют новых исследований.

В работе предлагается эвристика для адаптации известных конечно автоматных методов для тестирования телекоммуникационных протоколов в контексте, которая основана на синтезе усеченного дерева преемников, представляющего контекст, и может быть использована при тестировании других технических систем на основе конечно автоматных моделей.

Компьютерные эксперименты проводились с реализацией телекоммуникационного протокола и подтвердили эффективность использования предложенной эвристики.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность проблемы. В современном мире программное обеспечение (далее ПО) имеет все большое значение для человека, однако с ростом развития ПО растет и его сложность, а значит и количество ошибок. Существует множество сфер с критическими требованиями к надежности программного обеспечения, такие как медицина, банковское дело или воздушный флот. Наличие ошибки в управляющей части таких систем зачастую недопустимо, соответственно, актуальным является как можно более полное тестирование этих подсистем.

Известны следующие исторические факты фатального влияния ошибки ПО на человеческую жизнь.

а) В 1980-х годах пять пациентов умерли после получения большой дозы рентгеновских лучей в результате ошибки в программировании машины для лучевой терапии Therac-25. Авария, как полагают, была вызвана сбоем, который повлиял на производительность системы [1].

б) Четвертого июня 1996 года Европейское космическое агентство произвело испытательный пуск ракеты Ariane 5. Ошибка в контролирующем программном обеспечении, написанном на языке программирования Ada, вызвала самоликвидацию ракеты через 37 секунд после взлета [2].

в) В 2007 году счета более 3 миллионов клиентов одного из крупнейших банков Великобритании оказались под угрозой взлома из-за ошибки в системе безопасности [3].

г) Расследование NASA в 2007 году установило, что космический зонд Mars Global Surveyor перестал работать из-за ошибки в программном обеспечении, допущенной за пять месяцев до его гибели [4].

д) В 2009 году 206 пациентов клиники Cedars-Sinai Medical Center в Лос-Анджелесе были облучены недопустимо высокой дозой радиации в результате ошибки в программе спирального томографа [5].

е) В 2012 году сбой компьютера чуть не привел инвестиционную компанию Knight Capital к банкротству. Фирма потеряла полмиллиарда долларов через полчаса после того, как ошибка программного обеспечения позволила в автоматическом режиме системе самой покупать и продавать акции без надзора человека. Цена на акции компании упала на 75 процентов в течение двух дней [5].

Следует отметить, что в ходе разработки ПО изначально формируются требования к итоговому продукту, это означает, что задается некоторая спецификация. Спецификация содержит в себе список требований и характеристик, которым должна соответствовать

разрабатываемая программная реализация. В спецификации описывается вся необходимая информация о поведении системы и о разнообразии ситуаций, которые могут возникать при ее взаимодействии с другими системами или людьми. Таким образом, наиболее распространенные методы тестирования предназначены именно для проверки, соответствует ли разработанная программа исходной спецификации (тестирование соответствия или conformance testing). При этом известно, что гарантированная полнота тестирования возможна только при условии привлечения формальных моделей и, в частности, конечно автоматных моделей. Тестирование на основе формальных моделей заключается в следующем. По спецификации, которая является неформальным описанием с набором требований к программному обеспечению, строится формальная модель; на основе полученной модели строится множество тестовых последовательностей, которые в дальнейшем подаются на соответствующую данной спецификации проверяемую программную реализацию. На основании наблюдаемых выходных реакций делается заключение о соответствии проверяемой программной реализации ее спецификации.

С другой стороны, время тестирования, как правило, является ограниченным, соответственно, с учетом роста количества ошибок возрастает необходимость в эффективном и быстром тестировании. Однако, по причине встраиваемости тестируемой критической части в более крупную реализацию, подача тестов на нее может быть затруднено. Так, например, множество телекоммуникационных протоколов инкапсулируется в телекоммуникационный протокол, который находится выше уровнем по сетевой модели OSI, и поэтому программные реализации исследуемых телекоммуникационных протоколов часто являются внутренними частями более сложного ПО. Таким образом, необходимо не только синтезировать качественные тесты для реализаций телекоммуникационных протоколов, но и принимать во внимание невозможность прямой подачи данных последовательностей на проверяемую реализацию. Иными словами, требуется также решать задачу трансляции синтезируемых тестовых последовательностей через некоторый контекст. В данной работе экспериментально исследуются методы тестирования в контексте для телекоммуникационных протоколов. Тестирование проводится на основе дискетных моделей, а именно, конечных автоматов.

Целью работы является экспериментальное исследование конечно автоматных методов тестирования в контексте для программных реализаций телекоммуникационных протоколов.

Методы исследования. Для достижения поставленной цели в работе используется аппарат дискретной математики, в частности, для синтеза тестов и их последующей трансляции привлекаются методы классической теории автоматов, комбинаторики и

математической логики. Эффективность предлагаемых подходов оценивается посредством компьютерных экспериментов с существующими или разработанными автором программными реализациями телекоммуникационных протоколов.

Научная новизна.

Научная новизна работы заключается в следующем.

а) В работе предложена эвристика для адаптации известных конечно автоматных методов тестирования в контексте для телекоммуникационных протоколов.

б) Компьютерные эксперименты с реализациями телекоммуникационных протоколов (в частности, с реализацией протокола TCP, встроенной в программное обеспечение Psi) подтверждают эффективность предложенной эвристики.

в) Проведенные компьютерные эксперименты позволили выявить несоответствие реализации протокола TCP, встроенной в ПО Psi [6] версии 0.15-2 спецификации протокола TCP, представленной в RFC 793 [7].

Основные положения, выносимые на защиту.

а) Предложена эвристика для адаптации известных конечно автоматных методов для тестирования телекоммуникационных протоколов в контексте, которая основана на синтезе усеченного дерева приемников, представляющего контекст, и позволяет проверять функционирование других встроенных технических систем на основе конечно автоматных моделей.

б) Результаты экспериментального исследования по тестированию телекоммуникационных протоколов в контексте на основе конечно автоматных моделей, которые, в частности, позволили выявить несоответствие реализации протокола TCP, встроенной в ПО Psi (версия 0.15-2) спецификации протокола TCP, представленной в RFC 793.

Достоверность результатов. Эффективность предложенного подхода подтверждается компьютерными экспериментами с реализациями телекоммуникационных протоколов и, в частности, с реализацией телекоммуникационного протокола TCP, встроенной в ПО Psi версии 0.15-2. При этом дерево преемников контекста Psi было усечено при максимальной длине наблюдаемых последовательностей, равной шести.

Практическая ценность. Результаты работы могут быть использованы при тестировании реальных систем, поведение которых может быть описано посредством конечно автоматных моделей. Разработанные программные реализации инструментов автоматизации подачи тестовых последовательностей и сравнения результатов могут быть использованы при тестировании программного обеспечения различного назначения и на основе различных

подходов к синтезу тестов. Разработанные программные реализации инструментов трансляции различных представлений конечного автомата могут быть использованы при синтезе тестов для технических систем на основе конечно автоматных моделей с использованием различных автоматизированных средств, например, FSM test 1.0. Построенная формальная спецификация для протокола TCP в виде конечного детерминированного автомата может быть использована при синтезе тестов для реализации этого протокола, встроенного в иное программное обеспечение (отличное от Psi).

Апробация работы. Основные положения и результаты, представленные в работе, были представлены на двух Российских конференциях с международным участием «Новые информационные технологии в исследовании сложных структур» на Алтае (2014 г.) и Екатеринбурге (2016 г.), а также на международной Летней Школе IT CoSAS в Анапе (2015 г.). По результатам работы опубликована одна статья в журнале, входящем в Перечень ВАК.

Структура и объем работы. Магистерская диссертация включает введение, 5 глав, заключение и список используемой литературы. Диссертация содержит 20 рисунков и 1 таблицу. Объем работы составляет 58 страниц, в том числе: титульный лист – одна страница, оглавление – 2 страницы, основной текст – 52 страницы, список использованных литературных источников из 43 наименований – 3 страницы.

Во введении обсуждается актуальность и цель работы, формулируются положения, выносимые на защиту, а также отмечается апробация работы.

В первой главе приводятся основные определения и обозначения, используемые в работе; представляется краткий обзор видов и способов тестирования программного обеспечения, в том числе, методов синтеза тестов для моделей черного и белого ящика для тестирования в контексте (на основе формальных моделей).

Работа посвящена исследованию актуальной задачи, а именно, задачи тестирования реализаций телекоммуникационных протоколов. Телекоммуникационный протокол регламентирует процедуру и формат взаимодействия между некоторым количеством независимых компонент, которые могут являться устройствами, компьютерами, программами или процессами [8].

Тестирование программных реализаций телекоммуникационных протоколов является серьезной задачей при текущем большом распространении сетей и распределенных служб. Богатство разнообразия сетей и быстро увеличивающееся число соединений между ними объясняется как растущим числом взаимодействующих компонент, так и созданием множества принципиально новых телекоммуникационных протоколов, для которых требуется взаимодействие с существующими техническими системами. Гарантированная полнота

тестирования возможна только с применением формальных моделей, и среди формальных моделей, на основе которых могут синтезироваться тесты, для таких систем и, в частности, для программного обеспечения выделяют модели с конечным числом переходов. В данной работе в качестве такой модели рассматривается конечный автомат. Под конечным автоматом S понимается пятерка $S = (S, I, O, Ts, s_0)$, где S – множество состояний с выделенным начальным состоянием s_0, I – входной алфавит, O – выходной алфавит, T – отношение переходов, $T \subseteq S \times I \times O \times S$.

Задачи анализа конечные автоматы являются хорошо изученными, и в настоящий момент известны методы построения проверяющих тестов для конечных автоматов. Тем не менее, проверяющие тесты, построенные для автоматов, которые описывают поведение реальной системы целиком, могут иметь слишком большие размеры для проведения эксперимента за некоторое отведенное время. Поэтому для тестирования часто выбирают только критическую компоненту технической системы, полнота тестирования которой должны быть гарантирована. Однако такая компонента может быть недоступна для прямой подачи тестовой последовательности. Другими словами, тестируемая реализация находится в контексте.

Данная диссертация посвящена исследованию тестирования в контексте для телекоммуникационных протоколов. Соответственно, в первой главе отдельный раздел посвящен методам синтеза тестов на основе формальных моделей (в том числе, в контексте), а также методам синтеза тестов для телекоммуникационных протоколов.

Во второй главе описывается постановка задачи, а именно, обсуждается проблема тестирования в контексте для реализаций телекоммуникационных протоколов. Обсуждаются различные топологии тестирования в контексте при исследовании телекоммуникационных протоколов, рассматриваются особенности технических реализаций предложенных топологий. Предлагается эвристика для адаптации известных конечно автоматных методов тестирования в контексте телекоммуникационных протоколов.

Основная идея тестирования в контексте заключается в том, что тестер не имеет прямого доступа к тестируемой реализации и вынужден взаимодействовать с ней через контекст некоторой иной реализации. Следовательно, тестовая последовательность должна быть транслирована через контекст, что влияет на полноту и сложность метода синтеза тестов.

Упоминая синтез тестов в контексте для автоматных моделей, мы предполагаем композицию двух (или более) автоматных моделей: автоматной модели, описывающей тестируемую реализацию, а также автоматной модели, описывающий окружающую среду

(контекст), который определяет условия и правила трансляции тестовой последовательности. Однако на практике возможны более сложные (автоматные) взаимодействия.

Следует отметить, что в данной работе используется параллельная композиция автоматов, которая предполагает, что в один момент времени может передаваться слово только по одному каналу связи.

По причине того, что не всегда существует возможность подать проверяющую последовательность непосредственно на тестируемую реализацию телекоммуникационного протокола, можно выделить ряд топологий в зависимости от ограничений на управляемость и наблюдаемость критической компоненты технической сложной системы, что в свою очередь, может по-разному влиять на процесс тестирования. Различные топологии зачастую присущи определенным телекоммуникационным протоколам за счет особенностей их взаимодействия с протоколами других уровней. Соответственно, иногда для возможности исследования определенной топологии тестирования в контексте необходимо обратиться к техническим средствам, которые описаны в предпоследнем подразделе данной главы. Примерами таких программных средств могут служить tcpdump, netcat, wireshark.

Третья глава посвящена описанию алгоритмов и программных реализаций инструментов, разработанных в рамках данной работы для решения различных подзадач в процессе проведения экспериментального исследования конечно автоматных методов тестирования в контексте для телекоммуникационных протоколов.

Первый раздел посвящен основным алгоритмам и программной реализации инструмента автоматизации процесса множественной подачи тестовых последовательностей на программную реализацию контекста, в которой находится реализация тестируемого телекоммуникационного протокола. Реализация инструмента для автоматического тестирования представляет собой два самостоятельных модуля, предназначенных для узконаправленных задач. Комбинация этих модулей позволяет приспосабливать инструмент под конкретные виды тестирования. Первый модуль *wrap* облегчает работу испытателя (тестера) и исключает человеческий фактор из процесса подачи и извлечения данных с интерфейса тестируемой программы. Второй модуль *comp* облегчает работу испытателя (тестера) и исключает человеческий фактор из процесса сравнения данных тестирования. Работа с модулями происходит через добавление аргументов командной строки при исполнении модуля. Порядок аргументов не важен, при использовании недопустимых аргументов или нехватке обязательных выводится соответствующее сообщение об ошибке. В разделе подробно описан алгоритм работы каждого модуля, кроме того, перечислены основные функции, написанные при реализации данного инструмента.

Для промежуточной обработки формальных представлений детерминированного полностью определенного конечного автомата и тестовых последовательностей, синтезированных на его основе (для дальнейшего использования программы FSM Test 1.0 [9]), были написаны следующие программы на языке bash.
а) table_hum2fsm.sh – программа для перевода формата определения конечного автомата из таблиц переходов и выходов в его представление посредством списка переходов;
б) fsm_hum2toolkit.sh – программа перевода формата определения конечного автомата из произвольного списка переходов в список переходов в виде хеш-значений соответствующих входных/выходных символов и состояний, а также заголовочным объявлением мощностей требуемых алфавитов для последующего использования программы FSM Test 1.0;
в) tests_toolkit2hum.sh – программа перевода списка тестовых последовательностей из вида хеш-значений соответствующих входных/выходных символов в вид соответствующих им текстовых значений для конкретного автомата, заданного списком переходов.

В разделе кратко приводятся особенности использования каждой из программ, сопровождаемые иллюстративными примерами.

В четвертой главе приведено экспериментальное исследование конечно автоматных методов тестирования в контексте для телекоммуникационного протокола TCP.

Как известно, тесты будут полными и могут быть полиномиальной длины только для класса детерминированных полностью определенных автоматов. Поскольку задача тестирования, как правило, ставит целью наибольшую полноту, исходную автоматную модель, которая расположена на сайте School of Computer Science University of St Andrews [10] следует доопределить до детерминированного, полностью определенного конечного автомата. Доопределение автоматной модели протокола TCP до полностью определенного детерминированного конечного автомата проводилось по спецификации RFC793 [7].

На первом этапе преобразования автоматной модели из соответствующей спецификации были исключены переходы по временной задержке (таймауту); для этого был увеличен соответствующий уровень абстракции модели посредством принятия во внимание предположения «о мгновенной передаче событий». Полученный детерминированный полностью определенный приведенный конечный автомат, описывающий поведение клиентской реализации TCP представляется в работе таблицей переходов.

В качестве контекста выбирается остальная часть ПО Psi Client (исключая TCP), а также реализация XMPP. Следовательно, для построения тестовых последовательностей, подаваемых через контекст, требуется определение конечных автоматов, описывающих реализации контекста. В разделе приводятся построенные детерминированные, полностью

определенные конечные автоматы для компонент контекста. Для данных автоматов построены деревья преемников, усеченные на уже определенных состояниях. Возможные последовательности выходных реакций представлены регулярными выражениями.

Экспериментально исследуется полнота тестов, синтезированных классическими конечно автоматными методами, при их дальнейшей трансляции через соответствующий контекст. Рассматриваются метод Василевского (W метод). Для автомата TCP Client методом Василевского с помощью программы FSM Test 1.0 и прикладных инструментов построены наборы тестовых последовательностей. Вручную выделено подмножество тестовых последовательностей, трансляция которых возможна через контекст. Тестовые последовательности были поданы на тестируемую реализацию TCP Client через контекст. На основе анализа передаваемых по сети пакетов, был сделан вывод о несоответствии реализации TCP в ПО Psi ее спецификации, заданной конечным автоматом. А именно, проверяемая реализация передала по сети пакет с флагами FIN и ACK, находясь в состоянии, в котором спецификация не допускает передачу пакета с данными флагами.

В пятой главе обсуждаются приложения результатов, полученных в работе. В частности, приводится дискуссия о том, каким образом полученные результаты могут быть использованы при тестировании телекоммуникационных протоколов различных уровней. Обсуждается влияние выбора границы между проверяемой реализацией и ее контекстом на полноту транслированного теста на примере тестирования реализации протокола TFTP. Рассмотрены возможные проблемы при тестировании других телекоммуникационных протоколов различных уровней при использовании разных топологий доступности встраиваемой компоненты для управления и наблюдения. С другой стороны, во втором части главы приводится дискуссия, затрагивающая возможности управляемости и наблюдаемости не только проверяемой реализации, но и ее контекста. Обсуждаются ситуации возникновения ограничений на управление контекстом проверяемой компоненты. Показывается, каким образом конечно автоматные модели могут быть использованы при мониторинге проверяемой протокольной реализации с целью проверки ее функциональных и нефункциональных свойств.

В **заключении** кратко перечисляются полученные в диссертации результаты, и обсуждаются перспективы дальнейших научных исследований.

1 Основные определения и обозначения. Методы синтеза тестов для программного обеспечения

1.1 Основные определения и обозначения

Работа посвящена тестированию технических систем, а именно, тестированию системного программного обеспечения, в частности, реализаций телекоммуникационных протоколов. Поскольку всякую программное обеспечение не лишено ошибок, необходимо качественно тестировать любые программные продукты.

Следуя [11], *ошибкой* будем считать функцию ожидания человека и реакции системы, а проявлением ошибки — несоответствие ожиданий и реакции, или отказ системы от работы. В данной работе считаем, что ожидания человека опираются на разумность и, в первую очередь, на корректно сформированные требования к программному продукту. Здесь под системой в первую очередь будем понимать программное обеспечение, далее ПО.

Под *проверкой* системы на ошибку, как правило, понимается выявление истинности существования ошибки в системе [12]. Такая проверка может быть выполнена пассивно или активно. Проверка программного обеспечения на наличие ошибок, которое требует обязательного исполнения кода, называется *динамической (активной)*. Вместе с тем, проверка программного обеспечения, не требующая исполнения приложения, называется *статической (пассивной)*. Таким образом, *тестирование* есть процесс проверки системы на наличие ошибок, который заключается в подаче входных данных, наблюдении выходной реакции и последующем сравнении выходных данных с эталонными (ожидаемыми), а также выводе заключения о наличии или отсутствии ошибки в программном обеспечении [11].

Как было отмечено выше, проверка наличия ошибок в программном обеспечении возможна также без непосредственного исполнения программного кода, в частности, возможно проведение *верификации* системы, а именно, анализа свойств системы или ее модели. Например, без непосредственного исполнения программы можно установить возможность превышения границы типа для значения некоторой переменной. Кроме этого, проверка наличия ошибок возможна без прямого исполнения программного кода испытателем при *мониторинге*, когда испытатель имеет возможность только наблюдения за внешними воздействиями над проверяемой реализацией и ее реакциями.

Тестирование или верификация считаются *успешными*, если они выявили хотя бы одну новую ошибку. Вместе с тем, определение ошибки не является достаточным; после того, как ошибка обнаружена ее необходимо локализовать. Иными словами, необходимо уточнить, в каком именно месте имеется ошибка, и для этого соответствующее программное

обеспечение проходит процесс отладки. *Отладка* – это процесс непосредственного обнаружения ошибки. И, следовательно, отладка считается успешной, если она позволила приблизиться к исправлению ошибки.

Различные дискретные системы и методы их анализа можно разделить по знаниям об их структуре на три группы.

а) *черный ящик* – система, структура которой неизвестна, а именно, известны только порты (интерфейсы) системы;

б) *белый ящик* – система, структура которой известна полностью;

в) *серый ящик* – система, полных знаний о структуре которой нет, но, как правило, известны отношения ее частей (модулей) и их значения.

Здесь и далее под *модулем* системы понимается выделенный фрагмент системы, который оформлен как отдельный блок и обычно способен к самостоятельной работе или к работе в рамках другой технической системы. Функция или набор функций может служить примером модулем некоторого программного инструмента.

Критерием оценки качества проверки системы на наличие ошибок, как правило, является *полнота*. Под *полнотой* проверки системы на ошибку будем понимать способность охватывать (покрывать) проверкой долю возможных ошибок в систем. Говорят также, что проверка может быть полной относительно заданного класса ошибок; в этом случае предполагается, что каждая ошибка соответствующего типа обнаруживается данной проверкой.

Проверка с максимальной полнотой (покрытием всех возможных ошибок) называется *исчерпывающей*. Соответственно, для исчерпывающего тестирования на основе модели черного ящика необходимо покрытие всех возможных:

а) комбинаций наборов входных данных для систем без памяти;

б) последовательностей комбинаций наборов входных данных и значений состояний системы для систем с памятью.

Вместе с тем, для исчерпывающего тестирования системы на основе модели белого ящика необходимо покрытие всех возможных маршрутов передачи управления. В любом случае для исчерпывающего тестирования, как правило, требуются колоссальные средства (произведение времени и мощностей), поскольку для большинства систем количество комбинаций входных данных и возможных маршрутов огромно, и резко возрастает с увеличением сложности системы. Потому в реальных условиях, в основном, исчерпывающее тестирование не производится, и ставится упор на компромисс между затраченными средствами и степенью полноты проверки системы. Отметим, что для увеличения

практической пользы и полноты проверки без увеличения средств, затраченных на тестирование, применяются различные методы синтеза тестов.

Несомненно, что методы синтеза тестов, как и сам процесс тестирования, должны быть автоматизированы. В частности, в данной работе мы обращаемся к средствам автоматизации процесса тестирования программного обеспечения.

Средства автоматизации — системы, позволяющие облегчить и ускорить специализированную часть работы, в данном случае процесса тестирования.

Договоримся также, что под *надежностью* системы будем понимать вероятность ее работы без отказов в течение определенного периода времени; отметим, что надежность, как правило, рассчитывается с учетом стоимости каждого отказа [11]. *Совместимостью* системы назовем способность ее воспроизведения в различных условиях окружения [13].

1.2 Виды и способы тестирования программного обеспечения.

Тестирование можно классифицировать по следующим признакам.

По способу взаимодействия с внутренней структурой (применимо только к моделям белого и серого ящиков) различают [14] следующие виды тестирования.

Рассматривается *пошаговое* тестирование, при котором тесты проходят последовательно на каждом этапе сборки программы из ее модулей. Здесь же можно выделить *восходящее* (начинается с одиночного модуля, заканчивается готовым комплексом), и *нисходящее* (начинается со собранного комплекса, разбирается до отдельного модуля) тестирование.

Выделяют также *монолитное* тестирование, при котором вначале тестируются модули по отдельности, а затем полностью собранная программа.

Отметим, что выбор способа тестирования отражается на продуктивности будущего исправления ошибок. Таким образом, монолитное тестирование позволяет прозрачно пронаблюдать ошибки отдельных модулей, но при ошибке связей модулей тестирование дает мало информации о конкретной связи, на которой может находиться ошибка. Пошаговое тестирование лучше отслеживает межмодульные ошибки, в то время как нисходящее – быстрее показывает работоспособность готовой системы. Тем не менее, внутренние ошибки модулей наглядны только в состоянии максимально «разобранной» системы.

По степени автоматизации различают [15]:

а) *ручное* тестирование – наблюдается при проведении его человеком без использования средств автоматизации;

б) *автоматизированное* тестирование – справедливо при его проведении без явного участия испытателя;

в) *полуавтоматизированное* тестирование – справедливо для тестов, где испытатель использовал средства автоматизации на некотором этапе тестирования.

Если при тестировании ожидания человека опираются главным образом на спецификацию, то такое тестирование называется функциональным. Если же ожидания опираются на документацию, то говорят что это тестирование системы по целям. К такому тестированию, в частности, относится тестирование удобства использования, удобства эксплуатации, удобства обслуживания и др.

Мы не останавливаемся на этих способах тестирования, поскольку в данной работе нас интересует именно функциональное тестирование или тестирование конформности (соответствия).

1.3 Краткий обзор методов тестирования программного обеспечения

В данном разделе кратко описаны методы синтеза тестов для программного обеспечения. Рассматриваются различные возможности управления и наблюдения проверяемой системы.

1.3.1 Методы синтеза тестов для программного обеспечения для черного ящика

Методы синтеза тестов для модели **черного ящика** опираются на анализ структуры спецификации данного программного продукта. Вместе с тем, могут быть также использованы личный опыт и чутье испытателя. В частности, для модели черного ящика в литературе [12] рассматривается метод синтеза тестов на основе *эквивалентных разбиений*, при котором исходные данные программы разбиваются на конечное число классов эквивалентности. Так, каждый тест, входящий в некоторый класс эквивалентен (в смысле выходной реакции программы) любому другому тесту из этого класса. Тесты подбираются таким образом, чтобы участвовать в максимальном количестве классов эквивалентности. Обычно этот метод делится на два соответствующих практических этапа — выделение классов эквивалентности и непосредственное построение тестов.

Достаточно широко распространен метод синтеза тестов на основе *анализа граничных значений*, который заключается в генерации тестовых последовательностей, близких к некоторой границе. Это могут быть максимальные и минимальные значения некоторого проверяемого параметра.

Кроме того, широко распространен метод синтеза тестов на основе *анализа причинно-следственных связей* [11]. Этот метод можно интерпретировать как некоторые ЕСЛИ-ТО высказывания, когда определяется ряд причинно-следственных связей, которые, как правило, сформированы в виде таблиц. Тест синтезируется таким образом, чтобы покрыть все возможные связи.

В спецификации определяются множество причин и множество следствий. Строится таблица истинности, в которой последовательно перебираются все возможные комбинации причин и определяются следствия каждой комбинации причин. Таблица снабжается примечаниями, задающими ограничения и описывающими комбинации причин и/или следствий, которые являются невозможными из-за синтаксических или внешних ограничений. Каждая строка таблицы истинности преобразуется в тест.

Самостоятельным методом считается метод *предположения об ошибке*. В данном случае опытный испытатель выдвигает предположение о существующей ошибке на основе своего опыта и\или интуиции, и строит соответствующий тест.

Дымовым называется метод синтеза тестов, в котором учитываются явные и грубые поверхностные ошибки, которые часто наблюдаются в подобных случаях. Этот метод так же основан на опыте и интуиции испытателя.

1.3.2 Методы синтеза тестов для программного обеспечения для белого ящика.

Методы синтеза тестов для модели **белого ящика** опираются на знания о структуре программного обеспечения, и потому выражают различную степень полноты покрытия кода продукта. Необходимым, но не достаточным условием тестирования является *покрытие операторов*. В этом случае каждый из операторов должен выполниться хотя бы один раз. В литературе [11] рассматривается также критерий *покрытия решений* (переходов), который подразумевает такой набор тестов, что на каждое решение будет получено значение «истина» или «ложь» хотя бы однажды. Вместе с тем, при использовании модели белого ящика более сильным требованием может быть *покрытие условий*, при котором каждый условный оператор с выходом ИСТИНА и ЛОЖЬ должен быть покрыт, по крайней мере, один раз. Для того чтобы этот метод удовлетворял критерию покрытию операторов, необходимо чтобы он передавал каждой точке входа управление, по крайней мере, один раз.

Здесь и далее *точка входа* — адрес в оперативной памяти, с которого начинается выполнение программы, а *оператор* — инструкция изменения каких-либо внутренних данных.

Отметим, что перечисленные методы покрытия кода или его частей можно совмещать и расширять, в частности, в литературе [11] рассматривается метод *покрытия решений/условий*, а также метод *комбинаторного покрытия условий*.

При *мутационном* тестировании в систему намеренно вносится ошибка (мутация). Обнаружение внесенной ошибки некоторым тестом говорит о его качестве. Иными словами, при мутационном тестировании в программу вносятся ошибки различных типов, и синтезируется тестовые последовательности, обнаруживающие эти ошибки.

1.3.2.1 Методы синтезов тестов на основе формальных моделей

Под «Белым ящиком» можно в некоторых случаях понимать не только исходный код программного обеспечения, но и его формальное описание. Среди формальных описаний широко распространены формальные модели с графическим представлением.

Говоря о графическом представлении систем коммуникации можно выделить два вида представлений, используемых в практике программных разработок. Это графическое представление структуры системы и графическое представление ее поведенческих аспектов. Широко известны и чаще употребляемы графические представления поведенческих аспектов технологическим схемами, диаграммами Насси-Шнейдермана, схемами переходов, сетями Петри, диаграммами состояний, SDL и диаграммами последовательностей.

Преимущества графических представлений в простоте изучения и отсутствии необходимости математической подготовки для понимания. Недостатком является неоднозначность трактовки обозначений разными пользователями [16].

Всякае программная реализация телекоммуникационного протокола представляет собой некоторый алгоритм. Одно из возможных определений алгоритма, как математического объекта, является Машина Тьюринга (далее МТ). Следуя [15] можем сказать, что МТ достаточно для формального определения алгоритма. В данной работе под алгоритмом, способным решить определенную задачу, далее мы будем понимать именно эквивалентную ему МТ, эту задачу решающую.

МТ является расширением модели конечного автомата, расширением, включающим потенциально бесконечную память с возможностью перехода от обозреваемой в данный момент ячейки к ее левому или правому соседу [16]. Здесь и далее в работе строятся тесты для проверки корректности функционирования протокольных реализаций на основе конечных автоматов.

Под *конечным автоматом* с входным алфавитом I и выходным алфавитом O будем понимать пятёрку $S = (S, I, O, Ts, s_0)$, где S — непустое конечное множество состояний с

выделенным начальным состоянием s_0, I и O — непересекающиеся конечные алфавиты входных и выходных символов соответственно. Отношение $Ts \subseteq S \times I \times O \times S$ есть отношение перехода из состояния s в s' по входному символу i с реакцией o [17].

1.3.2.2 Особенности тестирования в контексте

При тестировании программного обеспечения в контексте некоторой другой программы, в частности, телекоммуникационных протоколов более высоких уровней, можно рассмотреть два общих случая.

В первом случае синтез тестовых последовательностей происходит для всей системы в целом. Методы этого класса строят тестовые последовательности для формального или неформального описания общего поведения и проверяемой системы и ее окружения (внешней среды). Такие тесты могут обнаруживать неисправности не только критической части, но и ошибки контекста, что часто приводит к большой длине проверяющего теста. Кроме этого, если таким формальным описанием будет выступает автоматная модель, то она может иметь неразумно большое количество состояний, что, в свою очередь, приводит к невозможности использования методов тестирования с гарантированной полнотой за счет слишком большого времени, необходимого для синтеза и подачи такого теста.

Во втором случае синтез тестовых последовательностей происходит только для конкретной части системы, далее решается соответствующая задача трансляции. В этом классе тестовые последовательности строятся исключительно для проверяемой подсистемы, возможно, заданной своим формальным описанием; затем тестовая последовательность транслируется через контекст с целью получения реальных входных воздействий, которые должны быть поданы на внешнею среду проверяемой системы.

1.4 Методы синтеза тестов для проверки реализаций телекоммуникационных протоколов на наличие ошибок

В данном разделе рассматриваются особенности тестирования реализаций телекоммуникационных протоколов как специализированного системного ПО. Телекоммуникационный протокол представляет собой комплекс правил логического уровня, регламентирующий единообразный способ поведения взаимодействующих частей системы для осуществления корректной передачи, обработки или сохранения информации. Телекоммуникационный протокол выражается спецификацией – набором требований к

реализации алгоритма, исполняющего заданные протоколом правила поведения. Автору работы известны следующие методы тестирования телекоммуникационных протоколов:

а) тестирование соответствия (аттестационное);

б) тестирование производительности;

в) тестирование совместного функционирования;

г) тестирование совместимости;

д) мониторинг.

Аттестационное тестирование предполагает проверку соответствия реализации телекоммуникационного протокола своей спецификации и, в соответствии со стандартом ISO 9646 [18], основывается на наличии в спецификации готовых сценариев проверки реализованной системы на соответствие заданной спецификации. Однако, вследствие специфики конкретных протоколов такие сценарии свободно не распространяются, или же наборы требований задаются неформально и не сопровождаются такими сценариями. В применении такого метода в качестве формы задания спецификации могут быть полезны конечно автоматные модели, для которых известны методы синтеза тестов с гарантированной полнотой.

Тестирование производительности известно также как нагрузочное тестирование и опирается на тестирование реализации телекоммуникационного протокола за пределами условий, заданных в его спецификации. В ходе такой проверки измеряются параметры системы, которые зависят от поступающей нагрузки, и значения этих параметров сравниваются с допустимыми значениями [19].

Тестирование совместного функционирования учитывает факт наличия в спецификации допуска различной интерпретации некоторых правил, что, следовательно, приводит к различным вариантам реализации телекоммуникационного протокола. Такие реализации могут не работать совместно, хотя каждая часть в отдельности соответствуют спецификации протокола. При использовании этого метода проверяется, при каких условиях и в какой степени разные реализации одного протокола могут производить корректное взаимодействие и выдавать ожидаемый результат [19].

Тестирование совместимости (тестирование функционирования) ставит упор на проверку реализации телекоммуникационного протокола в условиях некорректной работы взаимодействующей стороны. Метод является распространенным, благодаря нарастающей сложности протоколов с течением их развития. Также метод известен как регрессивное тестирование. Суть метода заключается в проверке новой системы сперва на правильное исполнение функций, которые успешно исполняла старая версия реализации устаревшего

протокола. После успешного прохождения такого теста далее синтезируются тесты для проверки новых функций, заданных в протоколе новыми правилами [19].

Метод мониторинга также известен как пассивное тестирование. Смыслом такого метода является отсутствие осознанных действий над реализацией телекоммуникационного протокола с целью проверки его по каким-либо свойствам. Тестирование в данном случае заключается в наблюдении за различными неуправляемыми испытателем входными действиями над проверяемой системой и ее выходными реакциями [20].

2 Тестирование реализаций телекоммуникационных протоколов в контексте

В данном разделе ставится задача тестирования в контексте для реализаций телекоммуникационных протоколов, обсуждаются возможные типы топологий при тестировании в контексте, а также предлагается эвристика для адаптации конечно автоматных методов тестирования в контексте для реализаций телекоммуникационных протоколов.

2.1 Постановка задачи

Как отмечалось ранее, особым случаем тестирования можно выделить так называемое тестирование в контексте. Суть такого тестирования в том, что тестер не имеет прямого доступа к проверяемой реализации и вынужден взаимодействовать с ней через контекст некоторой иной программы. Соответственно, тестовая последовательность должна быть транслирована через контекст, что непосредственно влияет на полноту и сложность метода синтеза тестов. Схематично этот факт отражен на рисунке 1.

2.2 Обсуждение различных топологий при тестировании в контексте и их технических реализаций

При исследовании синтеза тестов в контексте для автоматных моделей, предполагается композиция двух (или более) автоматных моделей: автоматной модели, описывающей тестируемую реализацию, а также автоматной модели, описывающий окружающую среду (контекст), который создает условия и правила трансляции тестовой последовательности. Однако на практике возможны более сложные автоматные взаимодействия, особенности которых могут сказаться на сложности и полноте синтезируемых тестов.

Стоит отметить, что в данной работе используется параллельная композиция автоматов, которая предполагает, что в один момент времени может передаваться слово только по одному каналу связи (рисунок 1).

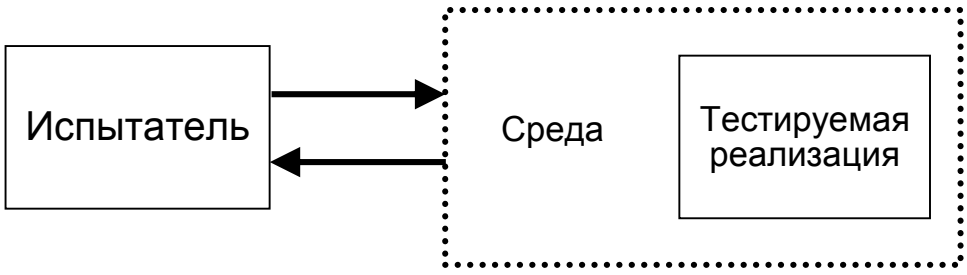


Рисунок 1 — Тестирование в контексте

Различные возможные топологии тестирования в контексте можно условно разделить по уровню доступа к тестируемой реализации. Частичная управляемость и наблюдаемость компонентов влияет на сложность соответствующей задачи анализа.

Так простейшей (в смысле наименьшего количества каналов связей) топологией тестирования в контексте будет являться, приведенная на рисунке 2. В данном случае, тестируемая реализация оказывается ненаблюдаемой и неуправляемой. Это означает, что мы не имеем ни одного прямого выхода, по которому могли бы наблюдать работу тестируемой реализации, а также, что мы не имеем ни одного прямого входа, на которой могли бы подавать последовательности непосредственно на тестируемую реализацию.

Такая топология типична для большинства встроенных несетевых программных реализаций, используемых более крупными программными продуктами. Например, таковыми являются библиотеки математических функций внутри сложных инженерных продуктов.

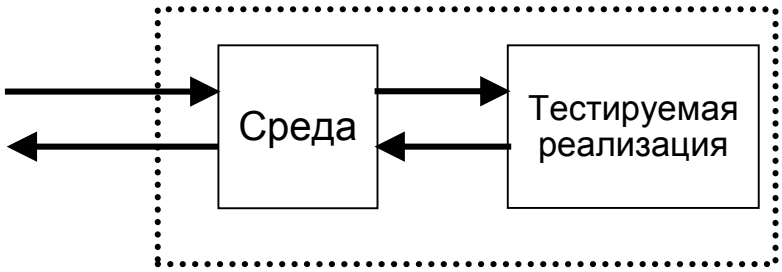


Рисунок 2 — Топология с ненаблюдаемой и неуправляемой тестируемой компонентой

Другой распространенной топологией тестирования в контексте может являться топология, приведенная на рисунке 3. Такая тестируемая реализация будет частично наблюдаемой, но неуправляемой. Другими словами, как и ранее, отсутствует возможность прямой подачи последовательности на тестируемую реализацию, однако появляется возможность наблюдать, какие последовательности передает тестируемая реализация программной реализации 2 и какой ответ получает от нее. Частичность заключается в том, что не все пути взаимодействия с тестируемой реализацией доступны для наблюдения, однако даже при такой топологии появляется возможность для некоторых случаев по наблюдаемым символам строить выводы, например, о конкретном состоянии, в котором находится тестируемая реализация. Данная топология распространена при тестировании сетевых модулей программного обеспечения, таких, как реализации сетевых протоколов TCP, UDP и более высоких уровней, в том числе TFTP, NTP и прочих, если они являются закрытой частью сложного ПО. Данная ситуация объясняется тем, что взаимодействие на различных уровнях в сети является в своем большинстве наблюдаемым, т. к. является взаимодействием по

открытому (публичному) общему каналу связи. Однако такая топология не совсем "работает", например, если искомые пакеты зашифрованы внутри других пакетов.

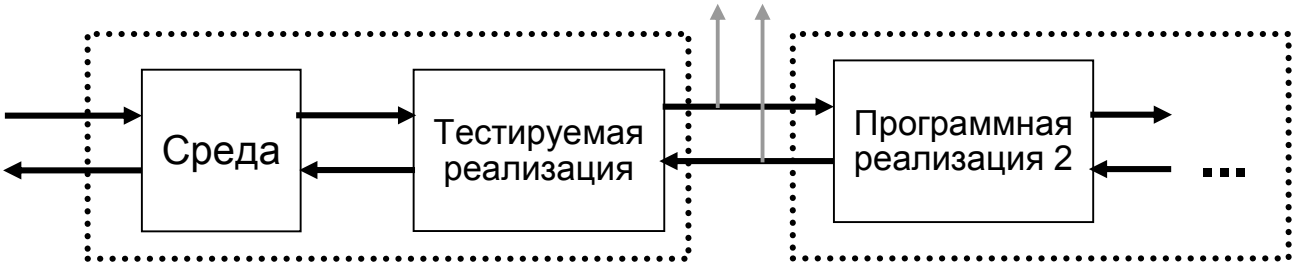


Рисунок 3 — Топология с частично наблюдаемой и неуправляемой тестируемой компонентой

Развитием предыдущей топологии может являться топология, приведенная на рисунке 4. Такая топология является частично управляемой и частично наблюдаемой. Это возможно, если часть каналов подачи последовательности на тестируемую реализацию и наблюдения ответных реакций проходит через контекст, а часть – непосредственно с тестируемой реализацией.

Примером такой топологии может являться эксперимент, когда взаимодействие с тестируемой сетевой компонентой моделирует сам тестер, как если бы взаимодействие происходило с другой программной реализацией, как на предыдущей топологии. Другими словами, тестер моделирует программную реализацию 2 (рисунок 3). Другим примером аналогичной топологии может являться случай, когда в некоторых моментах ПО, являющееся контекстом для тестируемой реализации, передает полный доступ к управлению тестируемой реализацией пользователю без своего контроля и влияния на себя.

Такая топология позволяет в некоторых случаях привести с помощью внешнего прямого воздействия автомат в необходимое состояние, если такое не было возможно для топологии с частично наблюдаемой, но неуправляемой тестируемой реализацией.

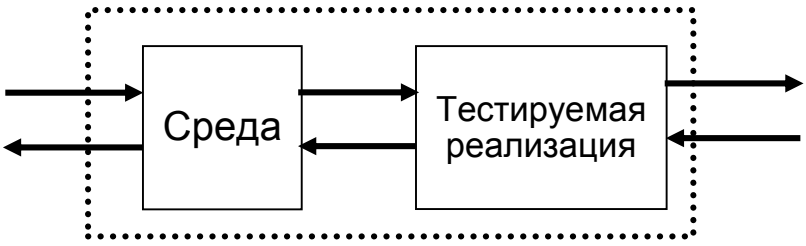


Рисунок 4 — Топология с частично наблюдаемой и частично управляемой тестируемой компонентой

Также вариантом топологии по уровню доступа может являться топология, приведенная на рисунке 5. В данной топологии тестирования в контексте проверяемая компонента является частично управляемой, но не наблюдаемой. Это означает, что к тестируемой реализации у нас есть возможность подавать последовательности, как через контекст, так и напрямую, однако наблюдать выходную реакцию тестируемой реализации мы можем только через контекст.

Такая топология справедлива для тех случаев, когда ПО, являющееся контекстом, в некоторые моменты времени позволяет подавать тестовые последовательности напрямую к тестируемой реализации.

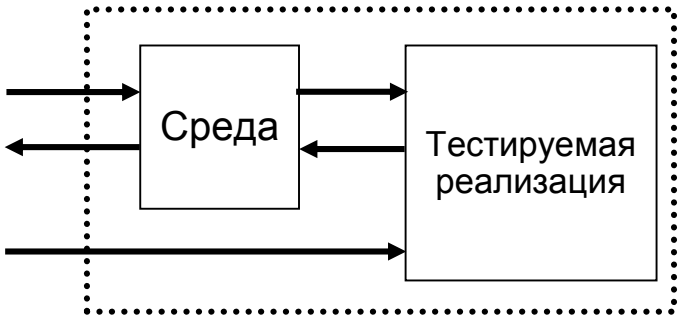


Рисунок 5 — Топология с ненаблюдаемой и частично управляемой тестируемой компонентой

Для топологий тестирования в контексте с частичной наблюдаемостью проверяемой компоненты можно выделить два типа взаимодействия наблюдаемого выходного канала связи и тестируемой компоненты.

Если топология, приведенная на рисунке, 3 имеет место быть, то тестер наблюдает посылаемое сообщение от тестируемой реализации к другому автомату, но своим наблюдением не приостанавливает действие параллельной композиции и далее, без ожидания наблюдателя, общение (диалог) внутри композиции продолжается.

Другим вариантом может использоваться топология, представленная на рисунке 6. В таком случае тестируемая компонента по уровню доступа остается аналогичной предыдущему варианту (частично наблюдаемой и неуправляемой), однако при получении сообщения по прямому каналу связи от тестируемой реализации параллельная композиция приостанавливается в ожидании передачи следующего сообщения из тестовой последовательности на среду.

Примером последнего случая может быть тестирование ПО, которое в некоторые моменты времени пропускает сообщение от тестируемой внутренней компоненты напрямую к пользователю в его стандартный интерфейс вывода.

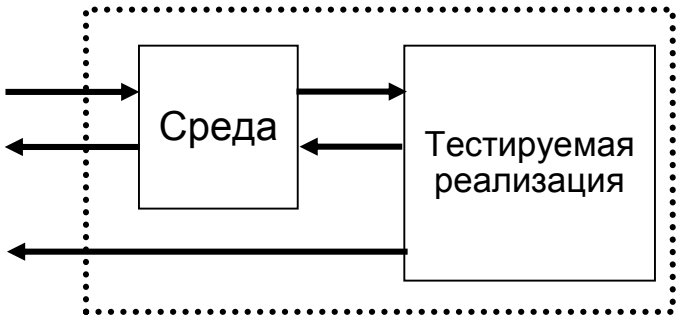


Рисунок 6 — Топология с частично наблюдаемой и неуправляемой тестируемой компонентой

Далее рассмотрим возможные технические реализации некоторых топологий.

Отметим, что если в ПО тестируемая компонента использует физический или логический сегмент сети, а также ее сетевые пакеты не инкапсулированы в зашифрованные пакеты, то топологии с частичной наблюдаемостью можно технически реализовать, перехватывая пакеты в соответствующем сегменте сети с помощью ПО tcpdump [21], netcat [22], wireshark [23] и прочего ПО для диагностики сетей.

Моделировать поведение приложения по сети для реализации топологии с частично управляемой и частично наблюдаемой тестируемой компонентой возможно через использование библиотек libpcap [21], а также через системный интерфейс программирования приложений raw socket.

2.3 Эвристика для адаптации известных конечно автоматных методов для тестирования в контексте телекоммуникационных протоколов

В данной работе предлагается эвристика для адаптации известных конечно автоматных методов для тестирования в контексте телекоммуникационных протоколов. Для применения эвристики спецификация проверяемого протокола, а также описание окружающего его контекста должны быть представлены в виде конечного автомата. Эвристика состоит из двух шагов.

На первом шаге для адаптации конечно автоматных методов тестирования в контексте необходимо выбирать максимальную глубину l для усеченного дерева преемников конечного автомата, описывающего поведение контекста. Как известно, *дерево преемников* представляет поведение автомата на всех входных последовательностях и, соответственно, является бесконечным. Вершины дерева преемников помечаются подмножествами состояний из множества S, а дуги – входными символами или входо-выходными парами. Корню дерева, как правило, приписывается начальное состояние автомата. Из вершины, помеченной

подмножеством состояний R, определена исходящая дуга, помеченная входо-выходной парой i/o, в вершину, помеченную io-преемником множества состояний R [24]. Таким образом, последовательности, помечающие ветви дерева, описывают все входо-выходные последовательности, определенные в начальном состоянии автомата. Очевидно, что дерево преемников является бесконечным, поскольку определено для входо-выходных последовательностей произвольной длины. Соответственно, для синтеза специальных входо-выходных последовательностей конечной длины по данному дереву определяются правила его усечения. В данном случае одним из правил усечения дерева преемников является ограничение по длине l входо-выходной последовательности, определенной в нем.

Максимальная длина l выбирается из соображений времени, которое может быть затрачено на процесс тестирования проверяемой реализации. Отметим, что построение дерева преемников необходимо для получения возможных выходных реакций автомата-контекста, которые позволят строить выводы о возможности транслирования тестовых последовательностей через контекст на проверяемую реализацию. Соответственно, нам может быть важно наличие в дереве приемников определенного состояния, которое имеет переходы, критически определяющие возможность или невозможность транслирования тестовых последовательностей. Тем не менее, чтобы достичь определенного состояния конечного автомата в худшем случае необходима последовательность длиной, на единицу меньше количества состояний в автомате. Даже для небольших автоматных моделей, имеющих порядка сотни состояний, наличие таких последовательностей остро скажется на времени, которое понадобится для подачи транслируемых проверяющих последовательностей.

Например, символ a, участвующий в тестовой последовательности для проверяемой реализации, будет выдан автоматом, описывающим поведение контекста, при переходе последнего из состояния N. При этом, чтобы достичь состояния контекста N совершить переход из него, нужно подать на него последовательность длины x. Это означает, что при трансляции проверяющей последовательности, длина будет увеличиваться на x при каждом вхождении символа a. Транслированный тест может иметь такую длину, что подача такого теста займет время большее, чем допустимо для тестирования. Поэтому разумно ограничить длину достижения состояния при моделировании поведения автомата контекста, что уменьшит полноту транслированного теста, но сэкономит время на трансляцию и подачу слишком длинных тестовых последовательностей.

Исходя из того, что в худшем случае каждому символу проверяющей последовательности при трансляции будет соответствовать последовательность длины l, работая в предположении, что время на подачу любого символа является одинаковым, выбор

максимальной глубины дерева преемников можно выразить следующим образом. Если T – время, которое в общем может быть затрачено на подачу тестовой последовательности; t – время, необходимое на подачу одного символа; L – длина проверяющей последовательности, то $l \leq \frac{T}{t * L}$.

После выбора величины l необходимо построить дерево преемников по конечному автомату, описывающего поведение контекста, применяя два нижеописанных правила. Первое правило формулируется следующим образом: если по выбранному переходу в вершине получаем состояние, которое присутствует на данном ярусе дерева или выше, то такая вершина дерева объявляется терминальной. Второе правило, как отмечалось ранее, определяется длиной l: если вершина дерева имеет высоту l, то данная вершина дерева объявляется терминальной.

На втором шаге эвристики предлагается на основе полученного дерева преемников построить регулярные выражения, описывающие возможные выходные реакции конечного автомата, описывающего поведение контекста. Следует иметь в виду, что если при построении дерева преемников были вершины, объявленные терминальными по причине достижения максимальной глубины обхода дерева, то нельзя гарантировать, что выводимые регулярные выражения будут описывать все возможные выходные реакции системы, описывающей контекст.

Для построения регулярных выражений, описывающих выходные реакции автомата, мы предлагаем осуществить моделирование полученного усеченного дерева преемников. При достижении терминального состояния, выписанная выходная последовательность ветви дерева, по которой произошел обход, группируется в скобки, и если текущее терминальное состояние было объявлено не из-за достижения максимальной глубины обхода дерева, данная группа обозначается символом "*", как обычно, означающим количество возможных повторений от нуля до бесконечности. Между группами устанавливается символ "|", означающий, что может быть встречена группа, находящаяся слева или справа. Если ни одно терминальное состояние не было объявлено таковым по причине достижения максимальной глубины обхода дерева приемников, то общее полученное выражение можно взять в группу и обозначить символом "*".

Полученные регулярные выражения описывают возможные выходные реакции автомата, описывающего поведение контекста. Сопоставляя эти выражения с набором последовательностей, которые нужно подать на реализацию телекоммуникационного протокола для проверки соответствия этой реализации своей спецификации, заданной конечным автоматом, можно выделить те последовательности, которые возможно

транслировать через контекст при выбранной длине *l*. Полученное дерево преемников можно использовать для трансляции проверяющей последовательности в последовательность, которую можно подать на контекст, для проверки свойств внутренней тестируемой реализации.

3 Описание алгоритмов и программных реализаций

В ходе экспериментального исследования конечно автоматных методов тестирования в контексте протокольных реализаций был разработан ряд инструментов, упрощающих процесс данного исследования и, следовательно, уменьшающие затраченное на него время. В данном разделе описываются алгоритмы и программные реализации предложенных инструментов.

3.1 Описание инструмента для автоматического тестирования программ и его алгоритмов

Данный подраздел посвящен основным алгоритмам и программной реализации инструмента автоматизации процесса множественной подачи тестовых последовательностей на программную реализацию контекста, в которой находится реализация тестируемого телекоммуникационного протокола.

Реализация инструмента для автоматического тестирования представляет собой два самостоятельных модуля, предназначенных для узконаправленных задач. Комбинация этих модулей позволяет приспосабливать инструмент под конкретные виды тестирования [25].

Первый модуль *wrap* облегчает работу испытателя и человеческий фактор из процесса подачи и извлечения данных с интерфейса тестируемой программы. По шагам его работу можно описать таким образом.

Вход: множество тестовых последовательностей $S=\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, тестируемая программа P, множество флагов с аргументами $F=\{\beta_1, \beta_2, \ldots, \beta_m\}$

Выход: множество выходных реакций P на S.

Выход: множество выходных реакций P на S.

1. Если F содержит более 20 элементов, то вывести сообщение о соответствующей ошибке и выйти.

2. Если во время преобразования F в структуру *arg* были найдены неожиданные флаги, то вывести сообщение о соответствующей ошибке и выйти.

3. Присвоить переменной *arg.type,* ответственной за способ восприятия S, 0 если был флаг *a*, иначе 2 если был флаг *w*, иначе 1.

4. Если был флаг *d* — вывести в подробной форме текущие основные переменные программы и среды.

5. Если P не была передана в виде аргумента, то вывести сообщение о соответствующей ошибке и выйти.

6. Пока истина выполнять следующие шаги.

6.1 Очистить переменную *arg.main_string*, отвечающую за элемент из S.

6.2 Пока истина выполнять следующие шаги.

6.2.1 Если пуста переменная *arg.fin*, отвечающая за файл содержащий S, то считать символ из стандартного входа, иначе считать из этого файла. Далее считанный символ обозначим через *c*.

6.2.2 Если *arg.type* равен 2 и *c* является пробелом, или *arg.type* не равен 0 и *c* является разрывом строки, или *c* является константой *EOF*, то выйти из цикла, иначе конкатенировать *c* к *arg.main_string*.

6.3 Если *arg.main_string* пуста и *c* равна константе *EOF*, то выйти из цикла.

6.4 Запустить *P* и перехватить ее стандартные вход и выход.

6.5 На вход *P* подать *arg.main_string*.

6.6 Очистить *arg.main_string*.

6.7 Пока текущий символ с выхода *P* не равен константе *EOF* выполнять следующие шаги.

6.7.1 Конкатенировать текущий символ с выхода целевой программы к *arg.main_string.*

6.8 Если пуста переменная *arg.fout*, отвечающая за выходной файл, то отправить *arg.main_string* на поток стандартного выхода, иначе отправить на поток записи в этот файл.

6.9 Если пуста переменная *arg.fout* и *arg.type* не равен 0, то отправить на поток стандартного выхода переход на новую строку, иначе отправить на поток записи в файл *arg.fout*.

6.10 Если *c* равна константе *EOF*, то выйти из цикла.

7. Выход.

Второй модуль *comp* облегчает работу с испытателя и человеческий фактор из процесса сравнения данных тестирования. По шагам его работу можно описать таким образом.

Вход: множество данных $S1=\{\alpha 1_1, \alpha 1_2, \ldots, \alpha 1_n\}$, множество данных $S2=\{\alpha 2_1, \alpha 2_2, \ldots, \alpha 2_n\}$, множество флагов с аргументами $F=\{\beta_1, \beta_2, \ldots, \beta_m\}$.

Выход: множество различий между соответствующими элементами множеств *S1* и *S2*.

1. Если *F* содержит более 20 элементов, то вывести сообщение о соответствующей ошибке и выйти.

2. Если во время преобразования *F* в структуру *arg* были найдены неожиданные флаги, то вывести сообщение о соответствующей ошибке и выйти.

3. Присвоить переменной *arg.type*, ответственной за способ восприятия *S1*, значение 0 если был флаг *a*, иначе – 2, если был флаг *w*, в противном случае – 1.

4. Присвоить переменной *arg.type2*, ответственной за способ восприятия *S2*, значение 0 если был флаг *A*, иначе – 2 если был флаг *W*, в противном случае – 1.

5. Если был установлен флаг *d* — вывести в подробной форме текущие основные переменные программы и среды.

6. Если *S2* не было передано в виде аргумента, то вывести сообщение о соответствующей ошибке и выйти.

7. Пока истина выполнять следующие действия.

7.1 Очистить переменную *arg.main_string*, отвечающую за элемент из *S1*.

7.2 Очистить переменную *arg.main_string2*, отвечающую за элемент из *S2*.

7.3 Пока истина выполнять следующие действия.

7.3.1 Если пуста переменная *arg.fin*, отвечающая за файл содержащий *S1*, то считать символ из стандартного входа, иначе – считать из этого файла. Далее считанный символ обозначим как *c*.

7.3.2 Если *arg.type* равен 2 и *c* является пробелом, или *arg.type* не равен 0 и *c* является разрывом строки, или *c* является константой *EOF*, то выйти из цикла, иначе конкатенировать *c* к *arg.main_string*.

7.4 Пока истина выполнять следующие действия.

7.4.1 Считать символ из файла с *S2*. Далее считанный символ обозначим как *c2*.

7.4.2 Если *arg.type2* равен 2 и *c2* является пробелом, или *arg.type2* не равен 0 и *c2* является разрывом строки, или *c2* является константой *EOF*, то выйти из цикла, иначе конкатенировать *c2* к *arg.main_string2*.

7.5 Если *arg.main_string* и *arg.main_string2* пусты, и *c* и *c2* равны константе *EOF*, то выйти из цикла.

7.6 Если *arg.main_string* и *arg.main_string2* не равны, то поместить в *arg.main_string* сообщение о различиях между *arg.main_string* и *arg.main_string2*, иначе – переход на шаг 7.9.

7.7 Если пуста переменная *arg.fout*, отвечающая за выходной файл, то отправить *arg.main_string* на поток стандартного выхода, иначе отправить на поток записи в этот файл.

7.8 Если пуста переменная *arg.fout*, то отправить на поток стандартного выхода переход на новую строку, иначе отправить на поток записи в файл *arg.fout*.

7.9 Если обе переменные *c* и *c2* равны константе *EOF*, то выйти из цикла.

8. Выход.

Таким образом можно сочетать модули для задач следующего типа. При заранее сформированном наборе входных данных для тестирования и наборе ожидаемых выходных реакций провести тестирование программы и получить результат о том, наблюдается ли ошибка на этих данных. Другим примером приложения может служить следующая задача. При сформированных входных данных для тестирования и наличии «программы-эталона», работу которой в конкретном локализованном случае мы считаем безошибочной, провести тестирование программы и получить результат о том, наблюдается ли ошибка на этих данных.

Инструмент для автоматического тестирования реализован на языке C++ и скомпилирован с помощью G++ из набора GNU Compiler Collection на Ubuntu (Linux 3.8.0 x64). Работа модулей предполагается в консольном режиме. Соединение модулей предполагается стандартными средствами перенаправления ввода-вывода, например, конвейером.

В процессе реализации для модулей инструмента автоматического тестирования программ были написаны следующие функции.

Функция обработки аргументов модуля *int arguments(int argc, char* argv[12], struct arguments& arg)* преобразует параметры в общем виде в подготовленные для дальнейшей работы аргументы, где argc — количество действительных параметров, argv — массив параметров, arg — готовые после обработки аргументы модуля; возвращает 0 при успешном выполнении, иначе 1.

Функция поиска в сырых параметрах вызова справки *bool findhelp(int argc, char* argv[12], string help)*, где help — выводимый текст справочной информации; возвращает значение true при успешном нахождении такого вызова, иначе false.

Функция *string vec2string(unsigned char vec)* возвращает строку из нулей и единиц, соответствующую битовому вектору vec.

Функция *int debug(int argc, char* argv[12], struct arguments& arg)* вывода подробной информации о текущей работе модуля.

Функция *int iomodule(struct arguments& arg)* обработки потока входных и выходных данных исходя из их форм, указанных в аргументах.

Функция выполнения смысловой части модуля *int mainwork (struct arguments& arg)*, которая для модуля *comp* производит непосредственное сравнение данных и формирует различия для вывода, для модуля *wrap* запускает тестируемую программу и перехватывает ее стандартный вход и выход, через которые ведет работу.

Работа с модулем происходит через добавление аргументов командной строки при исполнении модуля. Порядок аргументов не важен, при использовании недопустимых

аргументов или нехватке обязательных будет выведено соответствующее сообщение об ошибке. Аргументы могут быть совмещены, так аргумент «-wW» эквивалентен паре аргументов «-w» и «-W» Для модуля «wrap» аргументы регулируют следующее.

а) Вызов справочной информации об использовании этих модулей и их аргументов для упрощения взаимодействия инструмента с пользователем. Для ее вызова необходимо запустить модуль с аргументом «-h» или «--help». При любой неудаче при запуске модуля, вызванной неправильным составлением аргументов модуля в параметрах, модуль предлагает запустить себя с этим аргументом.

б) Вывод более подробной информации о текущей работе модуля для облегчения поиска ошибок в этом модуле. Вывод запускается аргументом «-d».

в) Источник входных данных; им может выступать или стандартный вход, или указанный файл (вызывается аргументом «-i [файл]»).

г) Цель назначения выходных данных; ее может быть или стандартный вывод, или указанный файл (вызывается аргументом «-o [файл]»).

д) Форма объекта из набора данных тестирования; это могут быть строки, разделенные пробелом (код 32 таблицы ASCII, вызывается аргументом «-w»), или переводом на новую строку (коды 0 и 10 таблицы ASCI, режим работы модуля по умолчанию), или концом ввода (символьная константа eof из стандарта языка C, вызывается аргументом «-a»).

е) Файл целевой программы, над которой проводится тестирование. Вызывается аргументом «-t [файл]» и является обязательным аргументом.

ж) Для модуля *comp* аргументы аналогично регулируют вызов справочной информации, вывод более подробной информации о текущей работе модуля, источник первого набора входных данных, форму объекта из первого набора входных данных, цель назначения выходных данных, а так же следующие параметры.

з) Источник второго набора входных данных; им может являться только указанный файл (вызывается аргументом «-I [файл]»).

и) Форму объекта из второго набора входных данных; это могут быть строки, разделенные или пробелом (вызывается аргументом «-W»), или переводом на новую строку (режим работы модуля по умолчанию), или концом ввода (вызывается аргументом «-A»).

Модуль *comp* выводит различия в следующем виде: {номер сравнения}. Первый объект: "{различие}". Второй объект: "{различие}".

3.2 Описание инструмента трансляции представлений формальных моделей

Для промежуточной обработки формальных представлений детерминированного полностью определенного конечного автомата и тестовых последовательностей, синтезированных на его основе (для дальнейшего использования FSM Test 1.0), были написаны следующие программы на языке bash.

а) table_hum2fsm.sh – программа для перевода формата определения конечного автомата из таблиц переходов и выходов в его представление посредством списка переходов.

б) fsm_hum2toolkit.sh – программа перевода формата определения конечного автомата из списка переходов в список переходов в виде хеш-значений соответствующих входных/выходных символов и состояний, а также заголовочным объявлением мощностей требуемых алфавитов для последующего использования FSM Test 1.0.

в) tests_toolkit2hum.sh – программа перевода списка тестовых последовательностей из вида хеш-значений соответствующих входных/выходных символов в вид соответствующих им текстовых значений для конкретного автомата, заданного списком переходов.

Ниже кратко приводятся особенности использования каждой из программ, сопровождаемые иллюстративными примерами

Программа table_hum2fsm.sh

Использование: необходимо осуществить следующий вызов из командной строки

./table_hum2fsm.sh Таблица_состояний Таблица_выходных_символов

Программа принимает на вход два файла таблиц (переходов и выходов, задающих детерминированный конечный полностью определенный автомат, передает на стандартный поток вывода все переходы, соответствующие этому автомату в формате:

`состояние входной_символ следующее_состояние выходной_символ`

Таблица переходов описывает, в какое состояние перейдет автомат по каждому входному символу (из текущего состояния). Таблица задается в следующем формате:

```
              вх_символ_1 вх_символ_2  …  вх_символ_n
состояние_1  состояние_x состояние_x …  состояние_x
состояние_2  состояние_x состояние_x …  состояние_x
…
состояние_m  состояние_x состояние_x …  состояние_x
```

Предполагается, что текст разделен табуляцией.

Таблица выходов описывает, какой выходной символ будет наблюдаться при подаче соответствующего входного символа в заданном состоянии. Таблица представляется в следующем формате:

```
              вх_символ_1 вх_символ_2  …  вх_символ_n
состояние_1  вых_симв_x  вых_симв_x  …  вых_симв_x
```

```
состояние_2  вых_симв_х  вых_симв_х  …  вых_симв_х
…
состояние_m  вых_симв_х  вых_симв_х  …  вых_симв_х
```

Программа fsm_hum2toolkit.sh

Использование: необходимо осуществить следующий вызов из командной строки

./fsm_hum2toolkit.sh Список_переходов

На стандартный поток вывода программа передает конечный детерминированный полностью определенный автомат в формате для использования пакета FSM Test 1.0: `состояние входной_символ следующее_состояние выходной_символ`
на стандартный поток вывода отдает конечный детерминированный полностью определенный автомат в формате для использования пакета FSM Test 1.0:

```
F 0
s количество_состояний
i количество_входных_символов
o количество_выходных_символов
n0 0
p количество_переходов
хеш_состояния хеш_входного_сивола хеш_следующего_состояния
хеш_выходного
…
```

Начальное состояние будет тем, которое стоит первым в лексико-графическом порядке в списке переходов, переданном программе.

Программа tests_toolkit2hum.sh

Использование: необходимо осуществить следующий вызов из командной строки
`./tests_toolkit2hum.sh Список_переходов Тест`

Программа принимает на вход список переходов конечного автомата в следующем в формате:
`состояние входной_символ следующее_состояние выходной_символ`
На вход также передаются тестовые последовательности для этого автомата, сгенерированные FSM Test 1.0 в формате хеш-значений.
На стандартный выход программа выводит список тестовых последовательностей, соответствующих автомату, заданному списком переходов.

4 Тестирование в контексте для протокола TCP

Говоря о тестировании реальных протокольных реализаций, неминуемым образом приходится сталкиваться с тем, что большинство протокольных реализаций являются внутренней частью более сложного ПО. Этот факт зачастую исключает возможность прямой подачи на тестируемую реализацию входных последовательностей и возникает необходимость говорить о тестировании в контексте, подразумевая то ПО, внутри которого находится тестируемая реализация, некоторой внешней средой.

В главе описывается эксперимент над протоколом TCP (Transmission Control Protocol), который содержит набор правил для стабильной передачи данных между хостами компьютерной сети [7]. Учитывая, что TCP является протоколом транспортного уровня и отвечает за стабильность соединения, многие протоколы прикладного уровня используют его для инкапсуляции своих пакетов, поэтому зачастую реализации TCP находятся внутри других программных продуктов и не имеют к себе прямого доступа со стороны пользователя. Невзирая не это, испытатель, проверяющий этот протокол на наличие ошибок может наблюдать некоторые выходные реакция напрямую с тестируемой реализации, из-за того, что пакеты TCP можно свободно перехватить, находясь в физическом сегменте сети по которому проверяемый программным продукт передает свои данные. Это означает, что для тестирования этого протокола можно использовать топологию с неуправляемой, но частично наблюдаемой проверяемой встроенной компонентой.

Предложенная эвристика для адаптации известных конечно автоматных методов для тестирования телекоммуникационных протоколов в контексте, которая основана на синтезе усеченного дерева приемников, представляющего контекст, и позволяет проверять функционирование других встроенных технических систем на основе конечно автоматных моделей, используется в данной главе при проведении эксперимента над реализацией TCP.

4.1 Доопределение автоматной модели TCP

Как известно, тесты будут полными и могут быть полиномиальной длины только для детерминированных полностью определенных автоматов. Поэтому для тестирования в контексте реализации протокола TCP, была исследована некоторая существующая автоматная модель, которая была далее преобразована в полностью определенный детерминированный автомат.

Следует отметить, что доопределение автоматной модели протокола TCP до полностью определенного детерминированного конечного автомата проводилось по

спецификации RFC793 [7]. Исходная автоматная модель, которая была взята из [10], представлена на рисунке 7.

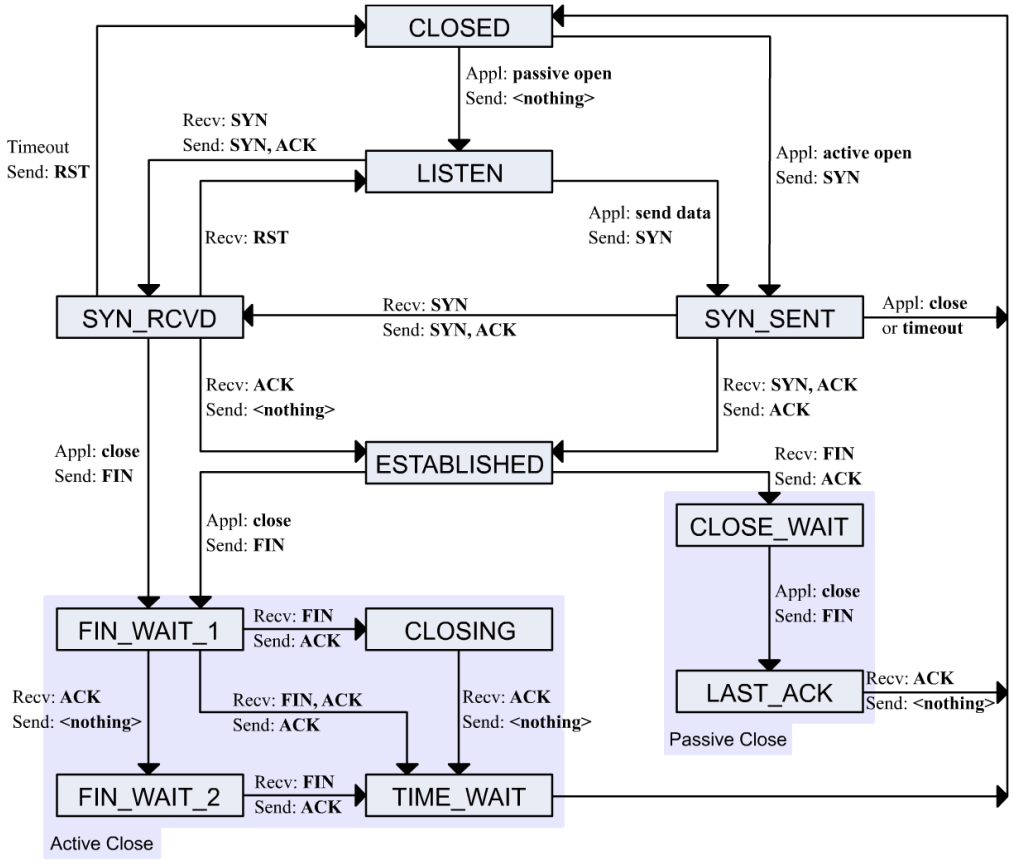


Рисунок 7 – Исходная автоматная модель TCP

Исключение переходов по таймауту.

На первом этапе преобразования автоматной модели из соответствующей спецификации были исключены переходы по временной задержке (таймауту); для этого был увеличен соответствующий уровень абстракции модели посредством принятия во внимание предположения «о мгновенной передаче событий».

Таким образом, исходя из того, что на рассматриваемом уровне абстракции сигналы по сети передаются всегда мгновенно, без ошибок и сохраняя свою очередность, то уровнями приоритета пакетов модель конечного автомата, описывающего TCP, пренебрегает. Вместе с тем, по причине того, что на данном уровне абстракции рассматривается взаимодействие только двух реализаций TCP в одном сегменте сети, игнорируется проверка поля SEQ и номер очереди пакета, оставляя в рассмотрении только его флаг как входной символ для автомата. Параметры, передаваемые TCP своему клиенту, также игнорируются, оставляя значимым только заголовок сообщения как выходной символ автомата.

Следующее состояние и выходная реакция для состояний Closed, Listen, SYN_rcvd, SYN_sent, Estab, FIN_wait_1, FIN_wait_2, Close_wait, Last_ACK по неопределенным входным сигналам может быть доопределено, руководствуясь следующим текстом перевода RFC793 [26] (рисунок 8).

Согласно главному правилу, сигнал перезагрузки (RST) должен посылаться всякий раз, когда приходит сегмент, который очевидным образом не предназначен для данного соединения. Если непонятно, имеет ли место данный случай, следует воздержаться от перезагрузки.

Можно выделить три группы состояний для соединения:

1. Если соединения не существует (CLOSED), то сигнал перезагрузки посылается в ответ на любой пришедший сегмент, за исключением встречного сигнала перезагрузки. В частности, сигналы SYN, адресованные на несуществующее соединение, ответгаются именно таким образом.
2. Если соединение находится в каком-либо несинхронизированном состоянии (LISTEN, SYN-SENT, SYN-RECEIVED), если какие-либо подтверждения пришедшего сегмента еще не отправлены (сегмент несет неприемлемое значение в поле ACK) или пришедший сегмент имеет уровень безопасности/закрытости не соответствующий уровню и защите данного соединения, то отправляется сигнал перезагрузки.
3. Если соединение находится в синхронизированном состоянии (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), то любой неприемлемый сегмент (не попадающий в окно номеров очереди, несущий неправильный номер подтверждения) должен приводить к появлению сегмента с пустым полем подтверждения, содержащего текущий номер в очереди на посылку, а также подтверждение, указывающее на следующий ожидаемый с этого соединения номер. Соединение остается в своем прежнем состоянии.

Рисунок 8 – Перевод RFC793 о реакциях на неопределенные входные символы

Согласно описанию поведения, приведенного выше, для всех состояний по входным символам SYN, FIN, ACK, SYN_ACK, FIN_ACK доопределен выходной символ RST и следующее состояние Closed, Listen, SYN_rcvd, SYN_sent, Estab, FIN_wait_1, FIN_wait_2, Close_wait или Last_ACK соответственно.

Поведение автомата для всех состояний по входным символам passive_open, active_open, send_data может быть доопределено согласно следующему тексту RFC793 (рисунок 9).

```
Запрос OPEN
  Состояние CLOSED (т.е. блок TCB отсутствует)
        Создать новый блок управления передачей (TCB) для хране-
     ния информации о состоянии соединения. Заполнить поля иден-
     тификатора местного сокета, чужого сокета, приоритета, за-
     крытости/безопасности, а также контрольного времени для кли-
     ента. Заметим, что некоторые параметры чужого сокета могут
     остаться неконкретизированными при пассивном открытии и со-
     ответствующие им поля должны быть заданы исходя из парамет-
     ров пришедшего SYN сигнала. Клиенту может быть предоставлена
     возможность проверять параметры безопасности и приоритета,
     если в ответ на такой запрос не будет получено сообщение
     "error: precedence not allowed" или "error: security/
     compartment not allowed". В случае пассивного открытия
     следует перейти в состояние LISTEN и вернуть управление дав-
     шему команду OPEN процессу. Если открытие является активным,
     а чужой сокет не конкретизирован, то вернуть сообщение
     "error: fireign socket unspecified". Если открытие является
     активным и указан чужой сокет, то послать сегмент с сигналом
     SYN. Выбирается начальный номер для очереди отправления.
  Состояние LISTEN
        Если происходит активизация и указан чужой сокет, то сме-
     нить состояние соединения с пассивного на активный, выбрать
     ISS. Послать сегмент с сигналом SYN, занести в SND.UNA зна-
     чение ISS, а в SND.NXT - ISS+1. Перейти в SYN-SEND состо-
     яние. Данные, указанные в команде SEND, могут быть посланы в
     том же сегменте с сигналом SYN, или же могут быть помещены в
     очередь на передачу, которая может быть осуществлена после
     перехода в ESTABLISHED состояние. Если в команде сделан за-
     прос на применение бита срочности, то в результате ее выпол-
     нения должны быть посланы сегменты данных. Если в очереди
     заказов на пересылку нет места, то в результате будет полу-
     чен ответ "error: insufficient resources". Если чужой сокет
     не указан, то вернуть сообщение "error: foreign socket
     unspecified"
```

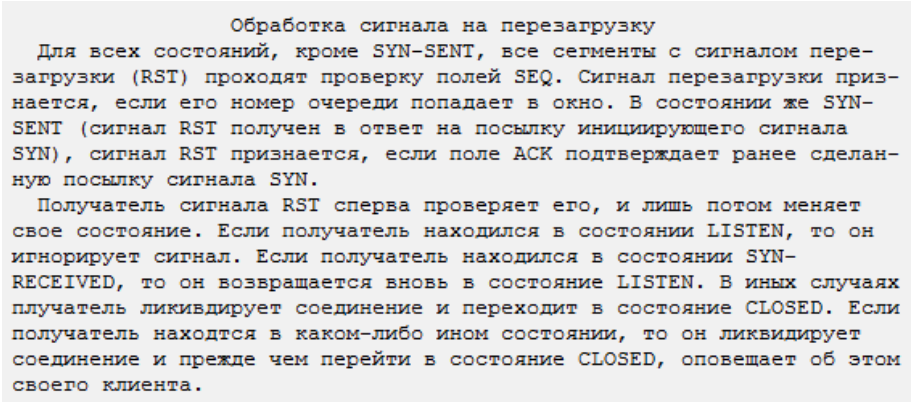
Рисунок 9 – Перевод RFC793 о реакциях на воздействия со стороны контекста

Поведение автомата для всех состояний по входным символу close доопределено согласно следующему тексту перевода RFC793 (рисунок 10).

```
Запрос CLOSE
  Состояние CLOSED (например, нет блока TCB)
        Если клиент не имеет доступа к такому соединению, вернуть
     сообщение "error: connection illegal for this process". В
     противном случае вернуть сообщение "error: connection does
     not exist".
  Состояние LISTEN
        Любые остающиеся неудовлетворенными запросы RECEIVE будут
     завершены с сообщением "error: closing". Стереть блок TCB,
     перейти в CLOSED состояние и вернуть управление клиенту.
  Состояние SYN-SENT
        Стереть блок TCB и вернуть сообщение "error closing" для
     любых еще остающихся в очередях запросов SEND или RECEIVE.
  Состояние SYN-RECEIVED
        Если не сделано каких-либо запросов SEND и нет данных,
     ожидающих отправки, то сформировать FIN сегмент и послать
     его, а затем перейти в FIN-WAIT-1 состояние. В противном
     случае поместить данные в очередь для рассмотрения после
     установления ESTABLISHED состояния.
```

Рисунок 10 – Перевод RFC793 о реакциях на входной символ close

Поведение по входному сигналу RST определяет следующий текст RFC793 (рисунок 11).



Обработка сигнала на перезагрузку

Для всех состояний, кроме SYN-SENT, все сегменты с сигналом перезагрузки (RST) проходят проверку полей SEQ. Сигнал перезагрузки признается, если его номер очереди попадает в окно. В состоянии же SYN-SENT (сигнал RST получен в ответ на посылку инициирующего сигнала SYN), сигнал RST признается, если поле ACK подтверждает ранее сделанную посылку сигнала SYN.

Получатель сигнала RST сперва проверяет его, и лишь потом меняет свое состояние. Если получатель находился в состоянии LISTEN, то он игнорирует сигнал. Если получатель находился в состоянии SYN-RECEIVED, то он возвращается вновь в состояние LISTEN. В иных случаях плучатель ликивдирует соединение и переходит в состояние CLOSED. Если получатель находтся в каком-либо ином состоянии, то он ликвидирует соединение и прежде чем перейти в состояние CLOSED, оповещает об этом своего клиента.

Рисунок 11 – Перевод RFC793 о реакциях на входной символ RST

Согласно описанию поведения, приведенного выше, для всех состояний по входному символу RST доопределен выходной символ и следующее состояние Closed или Listen соответственно.

В получившемся полностью определенном, детерминированном конечном автомате найдены эквивалентные состояния, а именно, состояния CLOSING и LAST-ACK. Для сохранения приведенности автомата-спецификации состояние CLOSING было исключено, все переходы в это состояние перенаправлены в состояние LAST-ACK.

Поскольку алфавиты входных и выходных символов исходной автоматной модели пересекались, в полученном конечном автомате предложено различать данные символы по принадлежности исходящих пакетов – сервер или клиент. Например, ACK_S – соответствует пакету, отправленному с серверной реализации TCP, а символ ACK_C – пакету, отправленному с клиентской реализации TCP.

Полученный детерминированный полностью определенный приведенный конечный автомат, описывающий поведение клиентской реализации TCP, можно задать следующей таблицей переходов/выходов – таблица 1.

Таблица 1 – Таблица переходов конечного автомата TCP

Входной символ \ Состояние	Closed	Listen	SYN_rcvd	SYN_sent	Estab	FIN_wait_1	FIN_wait_2	Close_wait	Last_ACK
FIN_ACK_S	Closed / RST_C	Listen / RST_C	SYN_rcvd / RST_C	SYN_sent / RST_C	Estab / RST_C	Closed / ACK_C	FIN_wait_2 / RST_C	Close_wait / RST_C	Last_ACK / RST_C
SYN_ACK_S	Closed / RST_C	Listen / RST_C	SYN_rcvd / RST_C	Estab / ACK_C	Estab / RST_C	FIN_wait_1 / RST_C	FIN_wait_2 / RST_C	Close_wait / RST_C	Last_ACK / RST_C
ACK_S	Closed / RST_C	Listen / RST_C	Estab / ok	SYN_sent / RST_C	Estab / RST_C	FIN_wait_1 / ok	FIN_wait_2 / RST_C	Close_wait / RST_C	Closed / ok
RST_S	Closed / quest	Listen / RST_C	Listen / ok	Closed / quest	Closed / quest	Closed / quest	Closed / quest	Closed / quest	Closed / quest
FIN_S	Closed / RST_C	Listen / RST_C	SYN_rcvd / RST_C	SYN_sent / RST_C	Close_wait / ACK_C	Last_ACK / ACK_C	Closed / ACK_C	Close_wait / RST_C	Last_ACK / RST_C
SYN_S	Closed / RST_C	SYN_rcvd / SYN_ACK_C	SYN_rcvd / RST_C	SYN_rcvd / SYN_ACK_C	Estab / RST_C	FIN_wait_1 / RST_C	FIN_wait_2 / RST_C	Close_wait / RST_C	Last_ACK / RST_C
close	Closed / ok	Closed / ok	FIN_wait_1/ FIN_C	Closed / ok	FIN_wait_1 / FIN_C	FIN_wait_1 / ignore	FIN_wait_2 / ignore	Last_ACK / FIN_C	Last_ACK / ignore
send_data	SYN_sent / SYN_C	SYN_sent / SYN_C	SYN_rcvd / ignore	SYN_sent / ignore	Estab / ignore	FIN_wait_1 / ignore	FIN_wait_2 / ignore	Close_wait / ignore	Last_ACK / ignore
active_open	SYN_sent / SYN_C	SYN_sent / SYN_C	SYN_rcvd / ignore	SYN_sent / ignore	Estab / ignore	FIN_wait_1 / ignore	FIN_wait_2 / ignore	Close_wait / ignore	Last_ACK / ignore
passive_open	Listen / ok	Listen / ignore	SYN_rcvd / ignore	SYN_sent / ignore	Estab / ignore	FIN_wait_1 / ignore	FIN_wait_2 / ignore	Close_wait / ignore	Last_ACK / ignore

4.2 Постановка эксперимента

При тестировании реализации TCP, входящей в ПО Psi, используется параллельная композиция автоматов, представленная на рисунке 12.

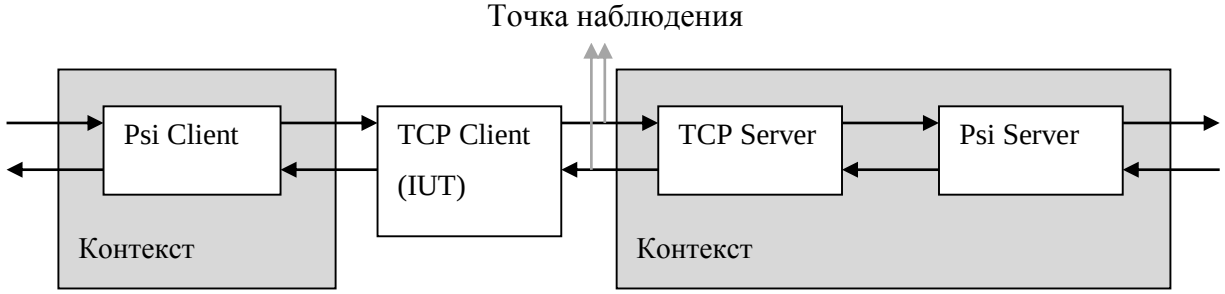


Рисунок 12 – Параллельная композиция автоматов
при тестировании реализации TCP из ПО Psi

В качестве контекста выбирается остальная часть ПО Psi Client (исключая TCP), а также реализация XMPP, который далее будем обозначать, как Server. Следовательно, для построения тестовых последовательностей, подаваемых через контекст, требуется определение конечных автоматов, описывающих реализации контекста – Psi Client, Psi Server, TCP Server. Автоматная модель для TCP Server была определена ранее.

Соответственно, в работе построены следующие детерминированные, полностью определенные конечные автоматы для Client и Server.

Автомат Client описывает поведение реализации XMPP клиента относительно реализации TCP клиента и пользователя. Учитывая, что реализация TCP клиента должна принимать от реализации XMPP клиента символы active_open, close, а действия пользователя, влияющие на это, могут быть связаны только с инициализацией соединения и закрытием его, то можно выделить следующие состояния:
Not_connect – XMPP клиент не предполагает передачу данных по сети, сигналы от реализации TCP (ok, ignore) не ожидаются, сигнал на закрытие соединения от пользователя (exit_c) не ожидается – такие сигналы будут обработаны сообщением об ошибке пользователю (error_c), при это смены состояния системы не произойдет. В этом состоянии ожидается команда от пользователя на открытие соединения (init_c) с посылом соответствующей команды для TCP реализации на активное подключение к серверу (active_open) с переходом в следующее состояние.

Мы далее более детально описываем возможные состояния автомата-клиента.

Estab – после отправки сигнала на открытие соединения клиент переходит в состояние, подразумевающее возможность передачи данных поверх уже установленного соединения TCP. В этом состоянии реализация XMPP клиента не ожидает сигналов от TCP реализации. Возможное общение с XMPP реализацией, инкапсулирующей свои данные через TCP, на данном уровне абстракции игнорируются. Сигнал от пользователя на установление соединения не ожидается. Все выше описанные неожидаемые сигналы не приводят к изменению состояния и сопровождаются сообщением об ошибке для пользователя. Пользователь может дать команду на закрытие соединения, что будет сопровождаться соответствующим сигналом для TCP реализации (close) и переходом в следующее состояние.

Start_close – после передачи сигнала о закрытии соединения, система ждет ответа от TCP реализации. При получении утвердительного сигнала, пользователю выдается сообщение об успехе завершения (successful_exit_c), и система переходит в начальное состояние Not_connect. При получении негативного ответа от TCP реализации, система передает сообщение об ошибке пользователю и так же переходит в начальное состояние. Действия пользователя в этом состоянии не ожидаются, не меняют текущего состояния и сопровождаются сообщением об ошибке. Соответствующий автомат приведен на рисунке 13.

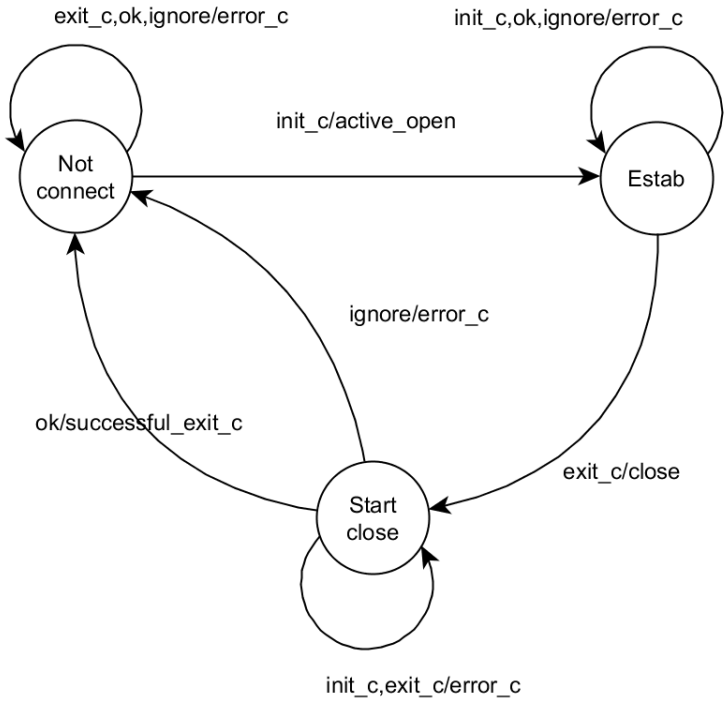


Рисунок 13 – Автомат Client

Автомат Server описывает поведение реализации XMPP сервера относительно реализации TCP сервера и пользователя. Учитывая, что реализация TCP сервера должна

принимать от реализации XMPP клиента символы passive_open, close, а действия пользователя, влияющие на это, могут быть связаны только с инициализацией соединения и закрытием его, то можно выделить следующие состояния:

Not_connect – XMPP сервер не предполагает передачу данных по сети, сигналы от реализации TCP (ok, ignore) не ожидаются, сигнал на закрытие соединения от пользователя (exit_s) не ожидается; такие сигналы будут обработаны сообщением об ошибке пользователю (error_s), и система не меняет своего состояния. Ожидается команда от пользователя на открытие соединения (init_s) с посылом соответствующей команды для TCP реализации на пассивное включение – начало прослушивания соответствующего порта на входящие подключения от клиентов (passive_open) с переходом в следующее состояние.

Start_listen – после передачи сигнала на пассивное открытие соединения, реализация сервера XMPP ожидает подтверждение от реализации TCP, при получении положительного сигнала, передается сообщение пользователю об успехе инициализации (successful_init_s), и система переходит в состояние Listen. При получении негативного сигнала, выдается сообщение об ошибке пользователю, и система переходит в начальное состояние Not_connect. Повторное сообщение от пользователя на начало инициализации в этом состоянии не ожидается, сопровождается сообщением об ошибке пользователю, и система не меняет текущего состояния. Возможно получение от пользователя команды на закрытие соединения, что сопровождается передачей соответствующего сигнала для TCP реализации и переходом в состояние Start_close.

Listen – после получения подтверждения об успехе инициализации, система ожидает подключения клиента. Негативный сигнал от TCP реализации и сообщение от пользователя на начало инициализации соединения не ожидаются, они сопровождаются сообщением об ошибке пользователю и не меняют текущего состояния системы. Получение положительного сигнала от TCP реализации обозначает для сервера входящее подключение, сопровождается сообщением пользователю о подключении (connect) и переходом в состояние передачи данных Estab. Возможно получение от пользователя команды на закрытие соединения, что сопровождается передачей соответствующего сигнала для TCP реализации и переходом в состояние Start_close.

Estab – в этом состоянии реализация ведет передачу данных поверх TCP, не ожидает ни сигналов от TCP реализации, ни сообщений от пользователя на начало инициализации, эти сигналы сопровождаются сообщением об ошибке пользователю и не меняют текущего состояния системы. Возможно получение от пользователя команды на закрытие соединения,

что сопровождается передачей соответствующего сигнала для TCP реализации и переходом в состояние Start_close.

Start_close – после передачи сигнала о закрытии соединения, система ждет ответа от TCP реализации. При получении утвердительного сигнала, пользователю выдается сообщение об успехе завершения (successful_exit_s), и система переходит в начальное состояние Not_connect. При получении негативного ответа от TCP реализации, система передает сообщение об ошибке пользователю и так же переходит в начальное состояние. Действия пользователя в этом состоянии не ожидаются, не меняют текущего состояния и сопровождаются сообщением об ошибке. Соответствующий автомат приведен на рисунке 14.

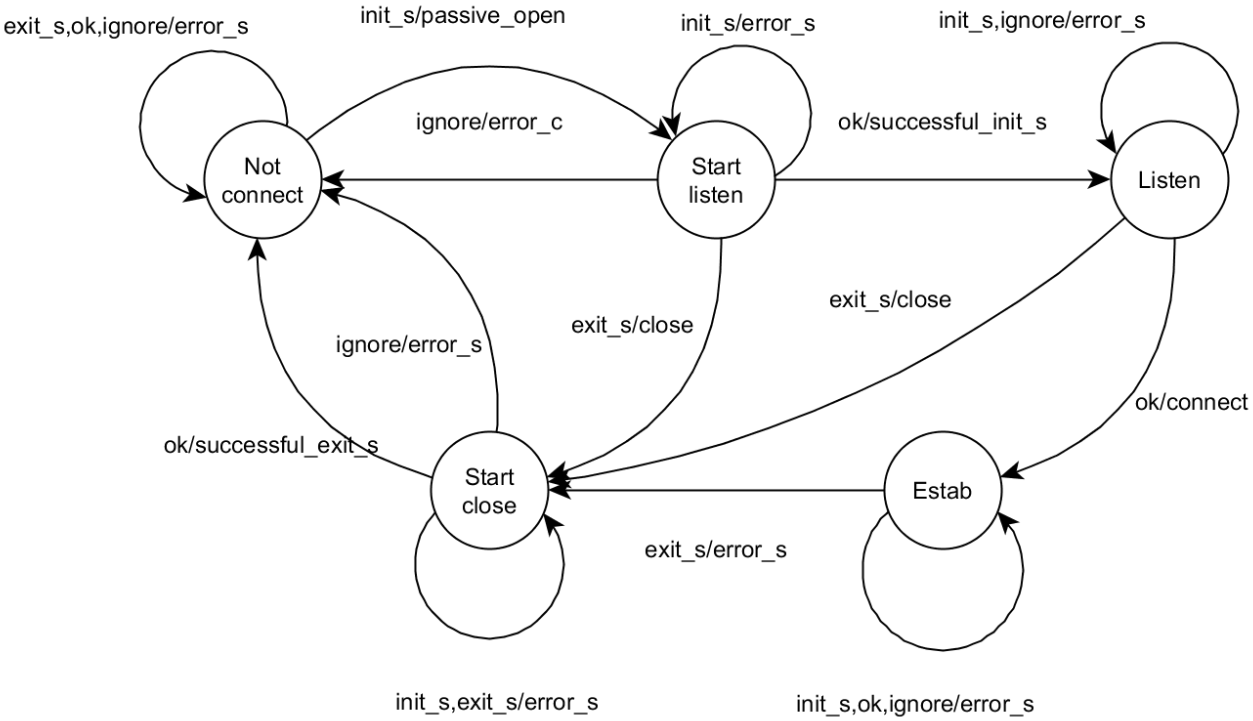


Рисунок 14 – Автомат Server

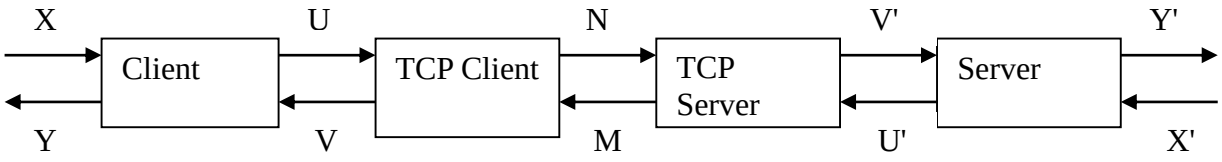


Рисунок 15 – Параллельная композиция автоматов в проводимом эксперименте

Ниже представлены алфавиты для соответствующих каналов связи (рисунок 15).

X = { *init_c* , *exit_c*};

Y = { *error_c* , *successful_exit_c* };

U = { *active_open*, *passive_open*, *send_data*, *close*};

V = { *ok*, *ignore*, *quest*};

X' = { *init_s*, *exit_s* };

Y' = { *error_s*, *successful_exit_s*, *successful_init_s*, *connect* };

U' = { *active_open*, *passive_open*, *send_data*, *close* };

V' = { *ok*, *ignore*, *quest* };

N = { *SYN_C*, *FIN_C*, *RST_C*, *ACK_C*, *SYN_ACK_C*, *FIN_ACK_C* };

M = { *SYN_S*, *FIN_S*, *RST_S*, *ACK_S*, *SYN_ACK_S*, *FIN_ACK_S* };

Для данных автоматов построены деревья преемников, усеченные на уже определенных состояниях (на текущем ярусе или ярусе выше). Данные деревья преемников изображены на рисунках 16 и 17, соответственно.

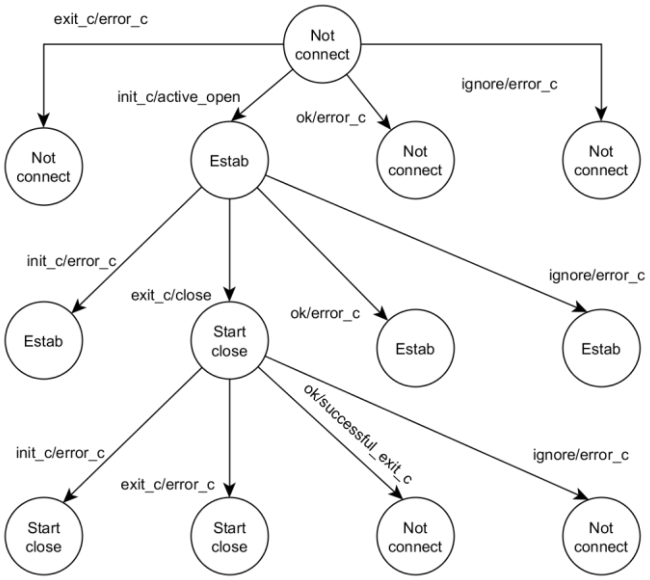


Рисунок 16 – Дерево преемников для автомата Client

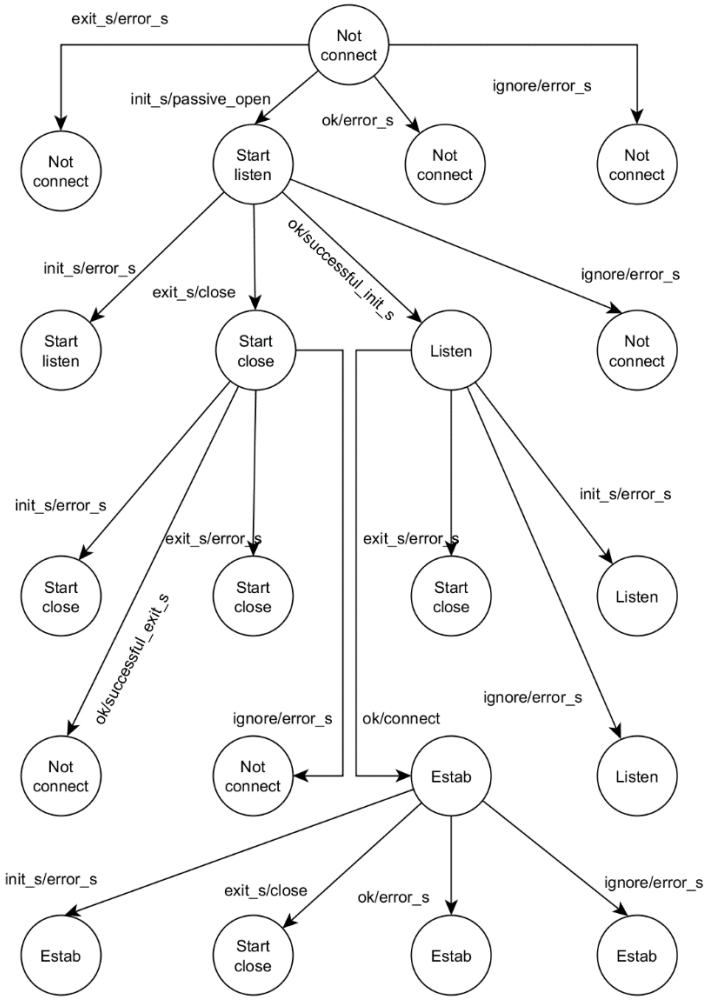


Рисунок 17 – Дерево преемников для автомата Server

Отметим, что при длине *l*, равной четырем, все ветви построенных деревьев преемников усекаются с применением первого правила. Этот факт позволяет построить соответствующие регулярные выражения для возможных выходных реакций с использованием символа "*".

Возможные последовательности выходных реакций представлены регулярными выражениями, где, напомним, используются следующие квантификаторы: "*" – для нуля и более повторений, "+" – для одного и более повторений. Вертикальная черта "|" разделяет допустимые варианты. Символ точки "." обозначает конкатенацию выходных символов автомата.

Возможные последовательности выходных реакций для Psi Client представляются регулярным выражением:

error_c *.active_open.error_c*.close.error_c *.(successful_exit_c|error_c).

(error_c *.active_open.error_c *.close.error_c *.(successful_exit_c|error_c))*

Возможные последовательности выходных реакций для Psi Server представляются регулярным выражением:

error_s*.passive_open.error_s*)+.((close.error_s *.

(error_s |successful_exit_s))|(successful_init_s.error_s*.

((close.error_s *.(error_s|successful_exit_s))|(connect.error_s*.

(close.error_s*.(error_s|successful_exit_s)))))))*

Из данных выражений были отброшены выходные реакции автоматов контекста, которые подаются не на проверяемую компоненту. По полученным выражениям в следствии были отсеяны некоторые последовательности из теста, построенного для тестируемой реализации.

4.3 Эксперименты над реализацией TCP

Цель экспериментов: оценка полноты тестов, синтезированных классическими конечно автоматными методами, при их дальнейшей трансляции через соответствующий контекст. Вместе с тем, в процессе экспериментов оценивается качество эвристики, предложенной в разделе 2 для эффективной трансляции тестовых последовательностей. В качестве одного из конечно автоматных методов синтеза тестов здесь и далее рассматривается метод Василевского (W метод) [17].

Эксперименты проводились над реализацией TCP, являющейся частью ПО Psi [28]. В качестве контекста на стороне сервера использовалось ПО Prosody [29].

4.3.1 Построение тестовых последовательностей

Для автомата TCP Client методом Василевского с помощью программы FSM Test 1.0 и прикладных инструментов преобразования формата конечного автомата table_hum2fsm.sh, fsm_hum2toolkit.sh и tests_toolkit2hum.sh (раздел 3) построены наборы тестовых последовательностей.

Вручную выделено подмножество тестовых последовательностей, трансляция которых возможна через контекст (Psi Client, Psi Server, TCP Server). В данном случае это множество включает лишь две следующих последовательности:

а) *active_open.SYN_ACK.close*

б) *active_open.SYN_ACK.FIN*

Следует отметить, что Данный данный список с большой вероятностью не включает в себя другие тестовые последовательности, которые так же могут быть транслированы. Проблема отсеивания тестовых последовательностей заключается в том, что нужно учитывать

как все возможные выходные последовательности Psi Client, так и все возможные выходные последовательности композиции автоматов Psi Server и TCP Server.

После решения задачи трансляции получены следующие тестовые последовательности:

а) *init_s.init_c.exit_c*

б) *init_s.init_c.exit_s*

4.3.2 Результаты экспериментов

Как отмечалось выше, при проведении эксперимента тестовые последовательности были поданы на тестируемую реализацию TCP Client через контекст [27]. На основе анализа передаваемых по сети пакетов, был сделан вывод, что в состояния Estab тестируемая реализация передает неизвестный выходной символ FIN_ACK, не определенный исходной и доопределенной автоматной моделью, описывающей поведение TCP.

Source	Destination	Protocol	Length	Info
192.168.205.156	176.196.43.232	TCP	76	40620 → 5222 [SYN] Seq=0 Win…
176.196.43.232	192.168.205.156	TCP	76	5222 → 40620 [SYN, ACK] Seq=…
192.168.205.156	176.196.43.232	TCP	68	40620 → 5222 [ACK] Seq=1 Ack…
192.168.205.156	176.196.43.232	TCP	68	40620 → 5222 [FIN, ACK] Seq=…
176.196.43.232	192.168.205.156	TCP	68	5222 → 40620 [FIN, ACK] Seq=…
192.168.205.156	176.196.43.232	TCP	68	40620 → 5222 [ACK] Seq=2111 …
176.196.43.232	192.168.205.156	TCP	68	5222 → 40620 [ACK] Seq=4669 …

Рисунок 18 – Результат наблюдения за каналами N и M, при подаче последовательности *init_s.init_c.exit_c*; неожидаемый символ отмечен

Source	Destination	Protoc	Len	Info
192.168.205.156	176.196.43.232	TCP	76	40618 → 5222 [SYN] Seq=0 W…
176.196.43.232	192.168.205.156	TCP	76	5222 → 40618 [SYN, ACK] Se…
192.168.205.156	176.196.43.232	TCP	68	40618 → 5222 [ACK] Seq=1 A…
176.196.43.232	192.168.205.156	TCP	68	5222 → 40618 [FIN, ACK] Se…
192.168.205.156	176.196.43.232	TCP	68	40618 → 5222 [FIN, ACK] …
192.168.205.156	176.196.43.232	TCP	68	40618 → 5222 [ACK] Seq=292…

Рисунок 19 – Результат наблюдения за каналами N и M, при подаче последовательности *init_s.init_c.exit_s*; неожидаемый символ отмечен

Таким образом, из общего числа тестовых последовательностей (492) для TCP через контекст удалось транслировать только две. Этот факт еще раз подтверждает актуальность задачи тестирования в контексте и необходимость поиска новых путей ее решений. Тем не менее, даже две тестовые последовательности позволили обнаружить несоответствие реализации телекоммуникационного протокола TCP, встроенной в ПО Psi версии 0.15-2, ее спецификации.

5 Обсуждение результатов для тестирования в контексте протоколов различных уровней и в условиях ограничений на управление контекстом

В данной главе обсуждаются приложения результатов, полученных в работе. В частности, приводится дискуссия о том, каким образом полученные результаты могут быть использованы при тестировании телекоммуникационных протоколов различных уровней. С другой стороны, во втором части главы приводится дискуссия, затрагивающая возможности управляемости и наблюдаемости не только проверяемой реализации, но и ее контекста. Показывается, каким образом конечно автоматные модели могут быть использованы при мониторинге проверяемой протокольной реализации с целью проверки ее функциональных и нефункциональных свойств.

5.1 Обсуждение результатов для тестирования в контексте протоколов различных уровней

Результаты, полученные в ходе эксперимента над телекоммуникационным протоколом TCP в контексте ПО Psi, являющегося реализацией протокола XMPP, могут быть использованы при тестировании других реализаций протокола TCP в контексте иных протоколов, использующих его для стабильной передачи данных. Например, построенные автоматы, описывающие поведение контекста, на данном уровне абстракции могут описывать поведение многих реализаций телекоммуникационных протоколов, таких как FTP [30], SMTP [31], POP3 [32], IMAP [33], SSH [34], L2TP [35], PPTP [36] и других.

В качестве примера рассмотрим конечный автомат, представляющий поведение протокола TFTP [38]. При тестировании реализации протокола TFTP [37], автоматная модель [38] которого представлена на рисунке 20, граница между проверяемой реализацией и контекстом может быть выбрана различно, что может влиять на длину тестовой последовательности, которую необходимо транслировать для подачи через контекст. При увеличении длины проверяющей последовательности, которая должна быть транслирована через контекст, время на трансляцию и тестирование будет увеличиваться, а количество возможных последовательностей, трансляция которых возможна, может уменьшаться. Несмотря на это, уменьшающаяся в таком случае сложность автомата (условно измеряется в числе состояний или числе его переходов), описывающего поведение контекста, может упростить задачу транслирования конкретного теста, в том числе возможно приближение полноты транслированного теста к аналогичному тесту проверяемой компоненты, построенному в изоляции от контекста. Отметим, что методика выбора оптимальной границы

между внешней средой и встраиваемой критической компоненты является перспективным направлением для дальнейшего изучения тестирования в контексте.

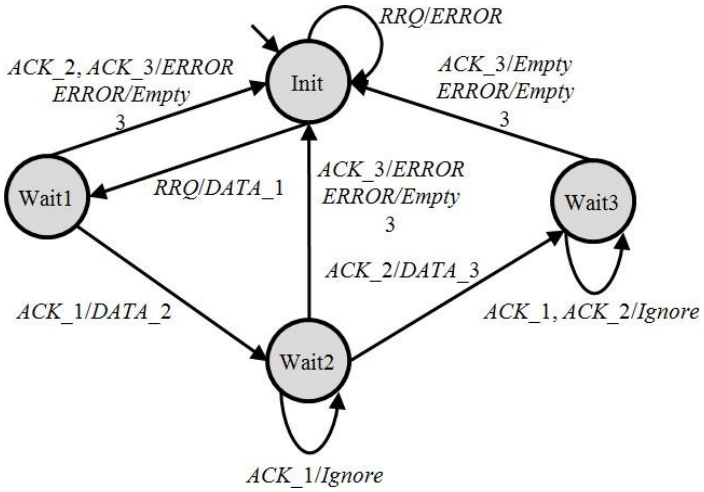


Рисунок 20 – Автоматная модель для протокола TFTP

Результаты экспериментального исследования показывают, что использование топологии с частичной наблюдаемостью эффективно при выявлении ошибок встраиваемой компоненты. В частности, в проведенном эксперименте именно частичная наблюдаемость прямых реакций тестируемой реализации позволила найти ошибку в программном обеспечении, использующем реализацию протокола TCP. Частичная наблюдаемость характерна для телекоммуникационных протоколов, которые описывают обмен информацией по сети, в которой испытатель может иметь точку наблюдения. К таким протоколам относится огромный перечень телекоммуникационных протоколов всех уровней сетевой модели от физического до прикладного. Исключение составляют протоколы, описывающие передачу данных и использующие для инкапсуляции своих пакетов протоколы с шифрованием. Тогда тестирование любых протоколов прикладного и представительского уровня, а иногда и протоколов сеансового и транспортного уровней, работающих поверх таких протоколов, использующих шифрование данных, например, SSH, HTTPS [39], IPsec [40] и других, невозможно при топологии с точкой наблюдения в сегменте сети, до тех пор, пока не будет решена задача расшифрования пакетов реализаций тестируемых протоколов.

Тестирование протоколов представления, среди которых – ASCII [41] и JPEG [42], возможно при исследовании топологии с частичной наблюдаемостью только в том случае, когда данные протоколы используются для преобразования информации с последующей передачей по сети с помощью телекоммуникационных протоколов уровнем ниже, или же при

возможности доступа к памяти, в которой записывается информация, преобразованная по проверяемому протоколу представления.

5.2 Обсуждение результатов для тестирования в контексте в условиях ограничений на управление контекстом

Как отмечается в первой главе работы, среди прочих методов тестирования телекоммуникационных протоколов, имеет большое значение пассивное тестирование программных реализаций телекоммуникационных протоколов в режиме наблюдения или мониторинга. Проверяемая реализация может находиться в контексте другого ПО, как и ранее, однако контекст не может получать никаких внешних стимулов – за его поведением разрешено только наблюдать. Такая ситуация характерна для ряда случаев. Например, в ходе разработки программного обеспечения не всегда могут быть выявлены все ошибки на этапе тестирования перед выпуском продукта в использование. Наблюдение за программной реализацией телекоммуникационного протокола после выпуска в процессе его использования может выявить оставшуюся часть ошибок. Другим обоснованием ограничения влияния на контекст может быть большая затрата ресурсов на конкретные воздействия над контекстом, которые необходимы для проведения активного тестирования. Несмотря на это, наблюдение за проверяемой реализацией, когда такое затратное воздействие происходит в ходе целевого применения тестируемой системы, может выявить ошибки, которые невозможно обнаружить иначе.

Учитывая, что испытатель (тестер) в данном случае не может контролировать последовательности, приходящие на вход окружающей встраиваемую проверяемую компоненту среды, встает вопрос, какие из последовательностей считать проверяющими и как оценить полноту такого тестирования.

Говоря об активном тестировании программных реализаций телекоммуникационных протоколов на основе конечно автоматных моделей, важно помнить, что конечный автомат, описывающий поведение проверяемой реализации, имеет начальное состояние. Именно в этом состоянии находится проверяемая реализация, когда на нее подается проверяющая последовательность. Для того, чтобы подать следующую проверяемую последовательность, испытателю необходимо привести тестируемую систему к известному начальному состоянию. При активном тестировании это не является затруднительным – например, с помощью перезапуска реализации. В этом случае тест для реализации телекоммуникационного протокола представляет собой итерационный запуск копий реализации с подачей очередной из проверяющих последовательностей, составляющих тест. Однако в условиях мониторинга

перезапуск реализации невозможен и возникает задача установки тестируемой реализации телекоммуникационного протокола в известное (начальное) состояние.

Таким образом, тестовая последовательность, которую тестер должен пронаблюдать в процессе тестирования выглядит следующим образом: $\alpha = \gamma.\beta$, где β – последовательность, проверяющая наличие ошибки, а γ – преамбула, приводящая реализацию в начальное состояние.

Для конечных автоматов задача идентификации текущего состояния может быть решена, в частности, посредством синтеза синхронизирующих последовательностей [24,43]. Синхронизирующая последовательность обладает следующим свойством: независимо от текущего состояния автомата и производимой выходной реакции, подача синхронизирующей последовательности переводит автомат из любого начального состояние в фиксированное (уникальное) состояние. Очевидно, что такие последовательности и могут выступать интересующим преамбулами γ.

С другой стороны, эксперименты с реализацией протокола TCP позволяют заключить, что часть символов не может быть передана контекстом на проверяемою реализацию. Однако эксперименты с конечным автоматом для TCP показывают, что при длине последовательности более 5 каждая третья последовательность является синхронизирующей. Вместе с тем, возникает вопрос, как часто такие последовательности могут быть произведен контекстом. Мы отмечаем, что задачи трансляции таких последовательностей через контекст не были решены в данной работе и являются перспективным направлением для дальнейших исследований.

ЗАКЛЮЧЕНИЕ

В работе рассмотрена актуальная задача синтеза проверяющих тестов для реализаций телекоммуникационных протоколов. В частности, исследован случай тестирования соответствующих реализаций в контексте другого программного обеспечения. Данная задача возникает при отсутствии прямого доступа к проверяемой протокольной реализации, и, соответственно, не каждая тестовая последовательность может быть транслирована через предъявленную внешнюю среду (контекст).

Работа посвящена экспериментальному исследованию конечно автоматных методов синтеза тестов для реализаций телекоммуникационных протоколов. В этом случае и проверяемая реализация, и ее контекст описываются подходящими конечными автоматами (или другими автоматными моделями). Для упрощения задачи трансляции теста в работе предложена эвристика для адаптации известных конечно автоматных методов для тестирования телекоммуникационных протоколов в контексте, которая основана на синтезе усеченного дерева приемников, представляющего контекст. Ее эффективность подтверждается результатами экспериментального исследования по тестированию телекоммуникационных протоколов в контексте на основе конечно автоматных моделей, которые, в частности, позволили выявить несоответствие реализации протокола TCP, встроенной в ПО Psi (версия 0.15-2) спецификации протокола TCP, представленной в RFC 793.

Следует отметить, что данная эвристика может быть использована при тестировании других технических систем в контексте, описанных автоматными моделями.

В работе также приводится дискуссия, каким образом предложенный подход может быть адаптирован на случай мониторинга телекоммуникационной сети с целью проверки корректности функционирования протокольных реализаций. В этом случае предложено использовать последовательности, наблюдение которых позволяет заключить о текущем состоянии реализации.

Ряд вопросов, непосредственно относящихся к задаче тестирования в контексте для протокольных реализаций, остался за рамками работы. В частности, не исследованы методы синтеза и трансляции тестов для неклассических автоматных моделей. Не автоматизирован процесс трансляции тестов для классических конечных автоматов. Компьютерные эксперименты проведены только для реализаций протокола TCP – другие протокола остались за рамками диссертационной работы. Отметим, что эти и другие вопросы открывают перспективы для дальнейших научных исследований.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Аджиев, В. М. Мифы о безопасном ПО: уроки знаменитых катастроф / В. М. Аджиев. – Открытые системы, № 06 – 1998. – 28 с.
2. Olds D. How one bad algorithm cost traders $440m / D. Olds, G. Consulting – TechTarget – 2012, №47 – 3 p.
3. Neumann, P. G. Computer-Related Risks / Peter G Neumann. – Addison-Wesley: Изд-во ACM Press – 1995. – Pp.178-181
4. Журнал "Новости из мира Компьютерной Безопасности, Интернет" – 2010, № 56 – 6 с.
5. Тэллес, М. Наука отладки / М. Тэллес, Ю. Хсих, Изд-во Кудиц-Образ – 2003. – 560 с
6. RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Core. Internet Engineering Task Force (IETF) // 2011
7. RFC 793: Transmission Control Protocol. DARPA Internet Program Protocol Specification // 1981
8. ГОСТ Р ИСО/МЭК 7498-1-99. - «ВОС. Базовая эталонная модель. Часть 1. Базовая модель». - ОКС: 35.100.70. - Действует с 01.01.2000. - 62с
9. N. Shabaldina, M. Gromov. FSMTest-1.0: a manual for researches / Proceedings of IEEE East-West Design & Test Symposium (EWDTS’2015), Batumi, Georgia, September 26-29, 2015, pp. 216-219
10. TCP Finite State Machine from School of Computer Science University of St Andrews: [официальный сайт] URL: http://tcp.cs.st-andrews.ac.uk/index.shtml?page=tcp_fsm, дата обращения (18.02.2015)
11. Гленфорд М. The Art of Software Testing. – М.: Диалектика, 2012
12. Лайза К. Agile Testing: A Practical Guide for Testers and Agile Teams. – М.: Вильямс, 2010
13. Бейзер Б. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем. – П.: Питер, 2004.
14. Синицын С. Верификация программного обеспечения. – М.: БИНОМ, 2008.
15. Канер С. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений. – Киев: ДиаСофт, 2001
16. Михаэль Р. Message Sequence Chart: Syntax and Semantics. – Eindhoven: Eindhoven University of Technology, 1999
17. Василевский, М. П. О распознавании неисправностей автоматов. / М. П. Василевский. // Кибернетика, 9(4) – 1973.– с. 93-108

18. ISO/IEC 9646 Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework. Parts 1-7, 1994
19. Гольдштейн Б.С., Рерле Р.Д., Ехриель И.М. Тестирование телекоммуникационных протоколов: проблемы и подходы // Сети и системы связи, Москва 2002г, №12.
20. ITU-T Q.780 Specifications of Signalling System No. 7. Signalling System No. 7 Test specification – General Description, 1995
21. Hakin9 Magazine. Issue 2/2008 (15) vol 3 No. 2, 2008
22. Yerrid K.C. Netcat Starter, Packt Publishing – 2013, 50c
23. Sanders C. Practical Packet Analysis, 2nd Edition – 2011, 280 pp
24. Методы синтеза установочных и различающих экспериментов с недетерминированными автоматами: диссертация ... кандидата физико-математических наук : 05.13.01 / Кушик Наталья Геннадьевна; [Место защиты: Нац. исслед. Том. гос. ун-т] – Томск, 2013
25. Щипачев Н. М. Инструмент для автоматического тестирования программного обеспечения //Новые информационные технологии в исследовании сложных структур: материалы Девятой российской конф. с межд. участием. Томск : Изд. дом Том. гос. ун-та, 2014. - с. 99-100
26. Перевод RFC 793 – Transmission Control Protocol. DARPA Internet Program Protocol Specification: [Электронный ресурс] URL: http://www.lissyara.su/doc/rfc/rfc793, дата обращения (20.10.2015)
27. Psi – free instant messaging application designed for the XMPP: [официальный сайт] URL: http://psi-im.org, дата обращения (17.09.2015)
28. Prosody – a modern XMPP communication server: [официальный сайт] URL: https://prosody.im/, дата обращения (2.11.2015)
29. Щипачев Н.М., Кушик Н.Г. Тестирование в контексте для реализации протокола TCP на основе автоматных моделей //Известия вузов. Физика. 2015. Т. 58, № 11/2. С. 115-118.
30. RFC 959: FILE TRANSFER PROTOCOL (FTP) // 1985
31. RFC 5321: Simple Mail Transfer Protocol // 2008
32. RFC 1081: Post Office Protocol - Version 3 // 1988
33. RFC 1730: INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4 // 1994
34. RFC 4253: The Secure Shell (SSH) Transport Layer Protocol// 2006
35. RFC 2661: Layer Two Tunneling Protocol "L2TP" // 1999
36. RFC 2637: Point-to-Point Tunneling Protocol (PPTP) // 1999

37. RFC 1350: THE TFTP PROTOCOL (REVISION 2) // 1992
38. Жигулин М. В., Прокопенко С. А. Синтез тестов для протокола TFTP на основе автоматной модели с таймаутами /М. В. Жигулин, С. А. Прокопенко – Известия высших учебных заведений. Физика 2012 Т. 55, № 9/2. С. 337-338
39. RFC 2660: The Secure HyperText Transfer Protocol // 1999
40. RFC 6071: IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap // 2011
41. RFC 20: ASCII format for Network Interchange // 1969
42. RFC 2435: RTP Payload Format for JPEG-compressed Video // 1998
43. M. Suhopluev, N. Schipachev, J. López, N. Kushik. Synchronizing sequences for effective network monitoring: An application to TCP //Новые информационные технологии в исследовании сложных структур: материалы 11-й международной конференции, 6–10 июня 2016 г. С. 40-41.

Уважаемый пользователь! Обращаем ваше внимание, что система «Антиплагиат» отвечает на вопрос, является ли тот или иной фрагмент текста заимствованным или нет. Ответ на вопрос, является ли заимствованный фрагмент именно плагиатом, а не законной цитатой, система оставляет на ваше усмотрение.

Отчет о проверке № 1

дата выгрузки: 13.06.2016 22:05:02
пользователь: b00men.pochta@gmail.com / ID: 3327260
отчет предоставлен сервисом «Антиплагиат»
на сайте http://www.antiplagiat.ru

Информация о документе

№ документа: 10
Имя исходного файла: Щипачев_НМ.docx
Размер текста: 796 кБ
Тип документа: Прочее
Символов в тексте: 107269
Слов в тексте: 13142
Число предложений: 534

Оригинальность: 97.85%
Заимствования: 2.15%
Цитирование: 0%

Информация об отчете

Дата: Отчет от 13.06.2016 22:05:02 - Последний готовый отчет
Комментарии: не указано
Оценка оригинальности: 97.85%
Заимствования: 2.15%
Цитирование: 0%

Источники

Доля в тексте	Источник	Ссылка	Дата	Найдено в
0.53%	[1] Тестирование программного обеспечения	http://knowledge.allbest.ru	раньше 2011 года	Модуль поиска Интернет
0.38%	[2] Лабораторная работа рассчитана на 4 академических часа. Подготовка к лабораторной работе	http://ru.convdocs.org	раньше 2011 года	Модуль поиска Интернет
0.35%	[3] МЕТОДЫ СИНТЕЗА ПРОВЕРЯЮЩИХ ТЕСТОВ С ГАРАНТИРОВАННОЙ ПОЛНОТОЙ ДЛЯ КОНТРОЛЯ ДИСКРЕТНЫХ УПРАВЛЯЮЩИХ СИСТЕМ НА ОСНОВЕ ВРЕМЕННЫХ АВТОМАТОВ 05.13.01 Системный анализ, управление и обработка информации (в отраслях информатики, вычислительной техники и автомат...	http://dissers.ru	26.06.2015	Модуль поиска Интернет