Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский Томский государственный университет»
Радиофизический факультет
Кафедра информационных технологий в исследовании дискретных структур

АВТОМАТНЫЕ МЕТОДЫ И АЛГОРИТМЫ СИНТЕЗА ТЕСТОВ ДЛЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ С ИСПОЛЬЗОВАНИЕМ ПОДХОДОВ ФОРМАЛЬНОЙ ВЕРИФИКАЦИИ

05.13.01 – Системный анализ, управление и обработка информации
(в отраслях информатики, вычислительной техники и автоматизации)

Автор работы:

Ермаков Антон Дмитриевич

Научный руководитель:

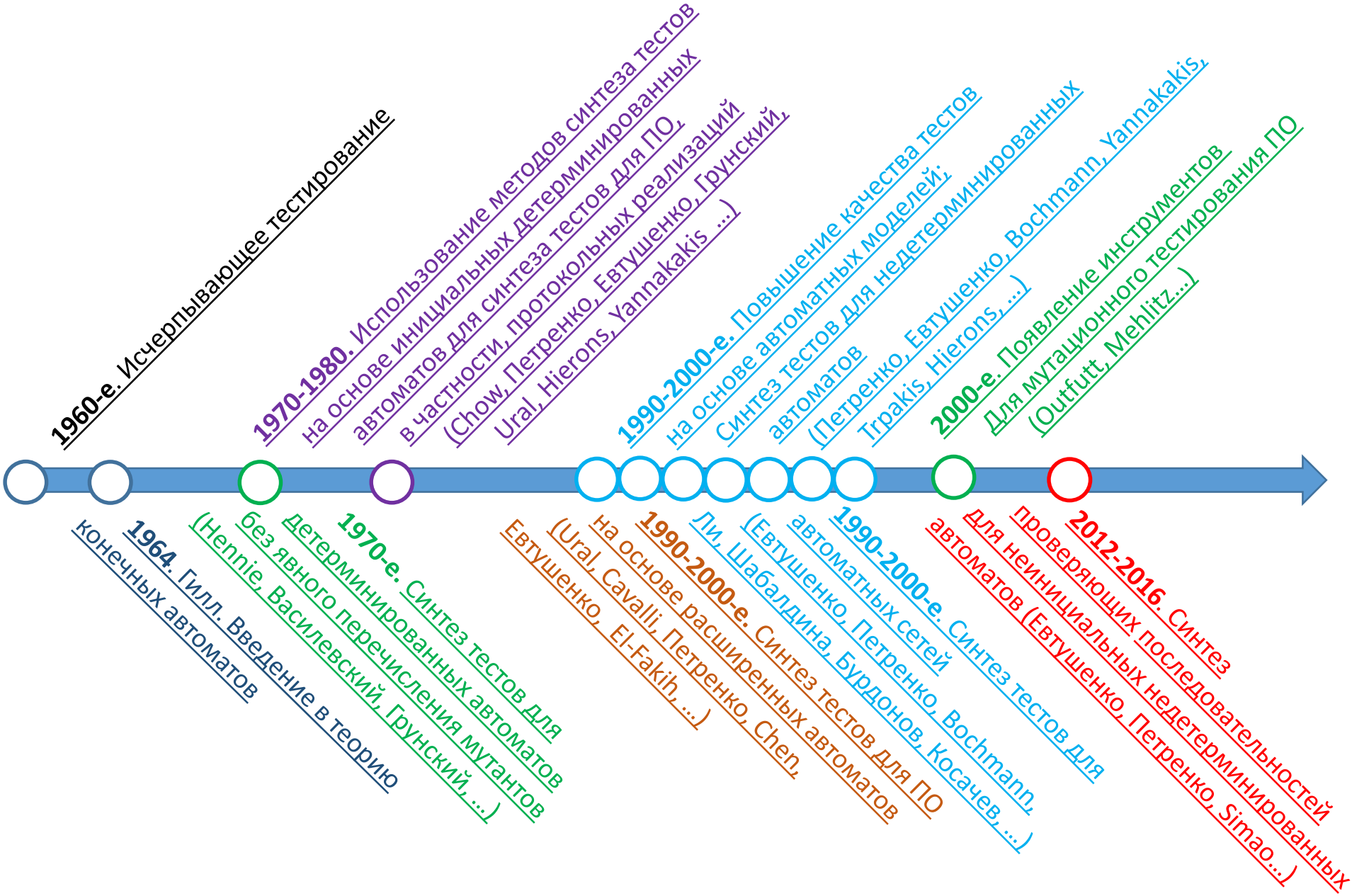
д. т. н., проф. Евтушенко Нина Владимировна

ТОМСК – 2017

АКТУАЛЬНОСТЬ

Октябрь 2012		В MS Word 2012, 2010, 2007 обнаружена уязвимость переполнения буфера стека http://www.securitylab.ru/vulnerability/431770.php	Возможен несанкционированный доступ к персональному компьютеру
Сентябрь 2013		Найдена ошибка в ПО Android KeyStore http://www.securitylab.ru/vulnerability/454368.php Уязвимы все устройства на базе Android 4.3 и младше	Возможен несанкционированный доступ к сохраненным ключам
Январь 2014		Сбой в работе Gmail лишил почты 42 миллиона пользователей в течении часа https://googleblog.blogspot.ru/2014/01/todays-outage-for-several-google.html	Работа сервиса электронной почты была нарушена в результате «внутренней ошибки в программном обеспечении»
Февраль 2016		В библиотеке GNU C Library (glibc) 2.x обнаружена критическая уязвимость http://www.securitylab.ru/vulnerability/479539.php	Уязвимы программы, использующие функцию getaddrinfo(). Удаленный пользователь может выполнить произвольный код на целевой системе
Март 2016		Японский рентгеновский телескоп Hitomi был утрачен после его распада на орбите во время неконтролируемого вращения http://global.jaxa.jp/projects/sat/astro_h/files/topics_20160415.pdf	Первичный анализ указывает на ошибочные данные в пакете программного обеспечения. Системы STT (системы слежения за звездам) и IRU (инерциальной системы наведения) пришли в несогласие по поводу положения спутника.
Ноябрь 2016		В маршрутизаторах D-Link обнаружена критическая уязвимость http://www.securitylab.ru/vulnerability/483371.php	Возможен переход пользователя на вредоносный веб- сервис

ХРОНОЛОГИЯ


1960-е. Исчерпывающее тестирование
1964. Гилл. Введение в теорию конечных автоматов
1970-1980. Использование методов синтеза тестов на основе инициальных детерминированных автоматов для синтеза тестов для ПО, в частности, протокольных реализаций (Chow, Петренко, Евтушенко, Грунский, Ural, Hierons, Yannakakis ...)
1970-е. Синтез тестов для детерминированных автоматов без явного перечисления мутантов (Hennie, Василевский, Грунский, ...)
1990-2000-е. Повышение качества тестов на основе автоматных моделей; Синтез тестов для недетерминированных автоматов (Петренко, Евтушенко, Bochmann, Yannakakis, Trpakis, Hierons, ...)
1990-2000-е. Синтез тестов для автоматных сетей (Евтушенко, Петренко, Bochmann, Ли, Шабалдина, Бурдонов, Косачев, ...)
1990-2000-е. Синтез тестов для ПО на основе расширенных автоматов (Ural, Cavalli, Петренко, Chen, Евтушенко, El-Fakih, ...)
2000-е. Появление инструментов для мутационного тестирования ПО (Outfutt, Mehlitz...)
2012-2016. Синтез проверяющих последовательностей для неинициальных недетерминированных автоматов (Евтушенко, Петренко, Simao...)

Некоторые нерешенные задачи

1. **а.** Тесты, построенные по расширенному автомату, используют различные критерии покрытия условий, путей, переменных и т.п, и при этом оставляют необнаруженными большое количество функциональных ошибок

   б. Существуют инструменты мутационного тестирования, которые позволяют обнаружить функциональные ошибки, но при этом возникает задача различения двух программных реализаций в языке высокого уровня
2. При обнаружении ошибки в программном обеспечении необходимым этапом является процесс её локализации
3. Автоматные методы тестирования ПО в основном направлены на построение тестов для детерминированных автоматов, в то время как многие реальные системы (телекоммуникационные протоколы) обладают недетерминированной спецификацией
4. Большинство тестов предполагают наличие в системе надежного сигнала СБРОС, что не всегда имеет место. Для систем, не обладающих надежным сигналом СБРОС, возникает задача построения проверяющей последовательности (в том числе, по недетерминированной спецификации)
5. Большое внимание уделяется проверке нефункциональных требований к ПО, в частности, проверке наличия уязвимостей в ПО. В особенности, важна проверка уязвимостей типа переполнения различных буферов, которые влияют на конфиденциальность и целостность информации

Цель

Разработать и программно реализовать автоматные методы синтеза тестов для повышения качества тестирования ПО

Задачи

1. С использованием инструментов мутационного тестирования **разработать** алгоритм повышения полноты тестов, построенных по расширенному автомату

2. **Разработать** алгоритм локализации неисправных компонент ПО; провести компьютерные эксперименты с программными реализациями разработанных алгоритмов

3. **Разработать** метод построения адаптивной проверяющей последовательности для недетерминированного конечного автомата относительно редукции

4. **Разработать и программно реализовать** алгоритмы по поиску уязвимостей в программном коде на языке C/C++

Глава 1. Определения и обозначения

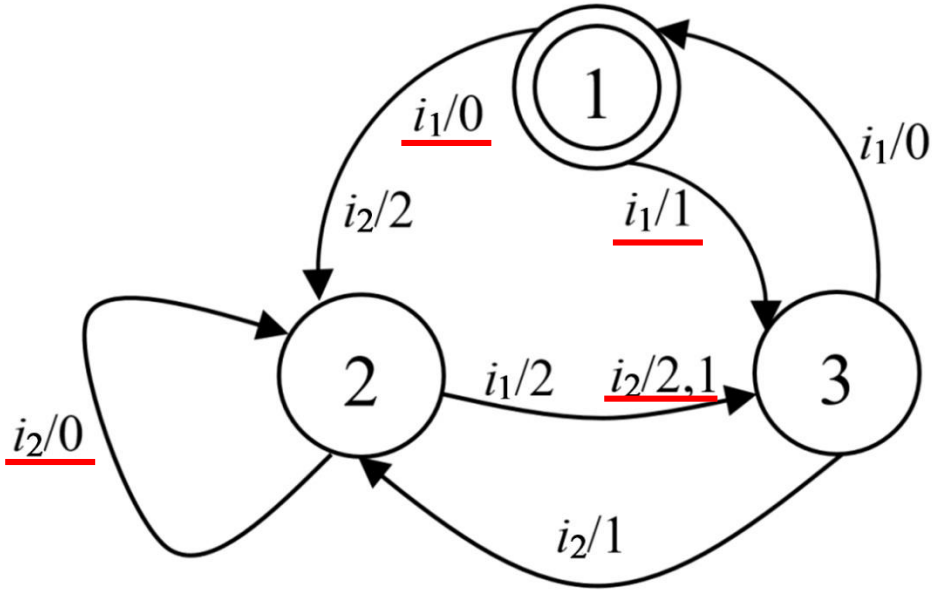
$S = (S, I, O, T_S)$

S – множество *состояний*

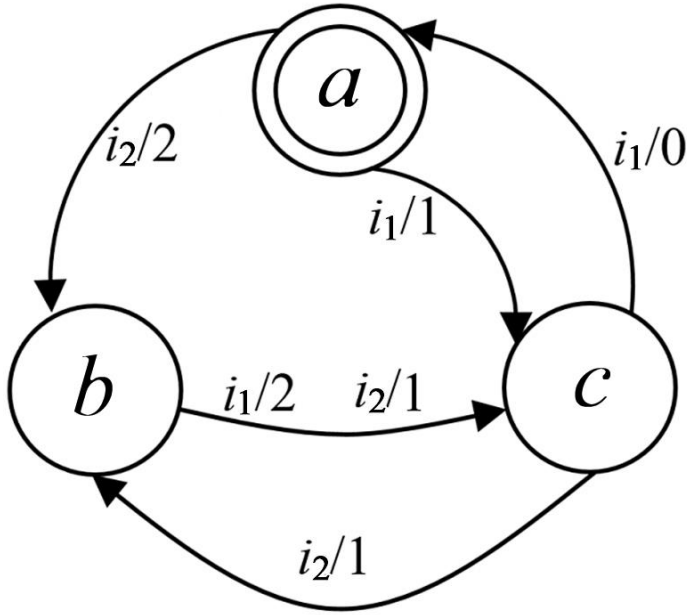
I – *входной* алфавит

O – *выходной* алфавит

T_s – отношение *переходов*



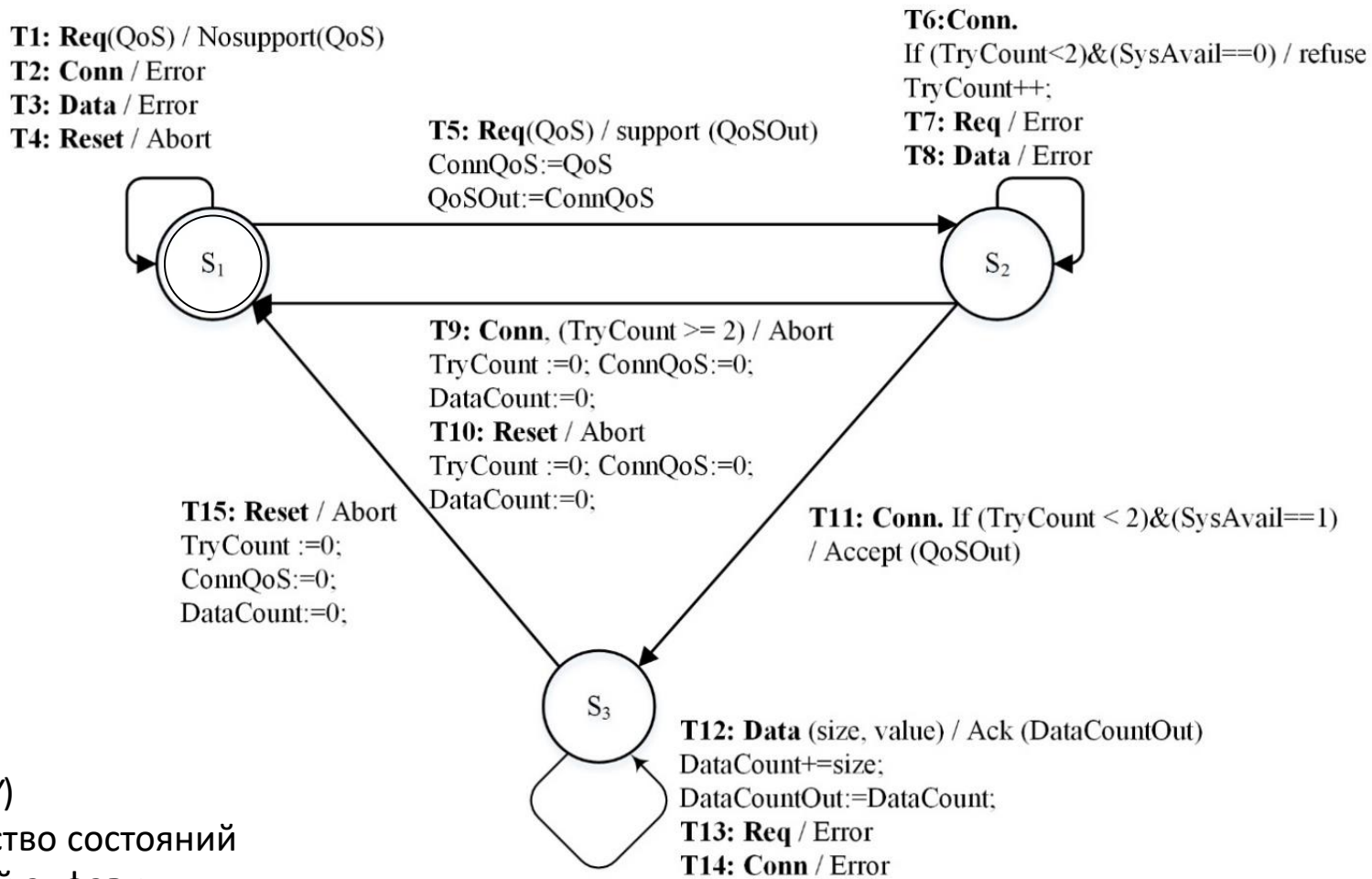
Пример недетерминированного автомата



Пример детерминированного автомата

Автомат *инициальный*, если в нем выделено *начальное* состояние

Расширенный автомат



(*S*, *X*, *Y*, *T*, *V*)

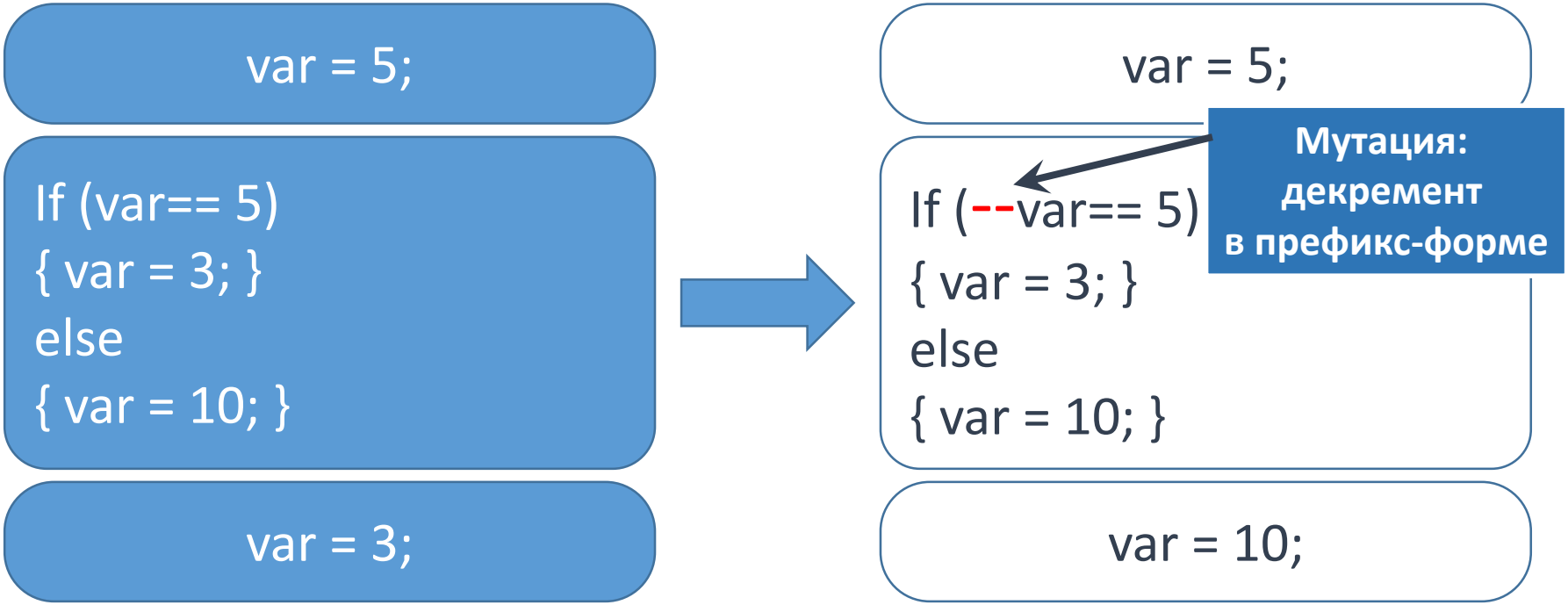
S – множество состояний

X – входной алфавит

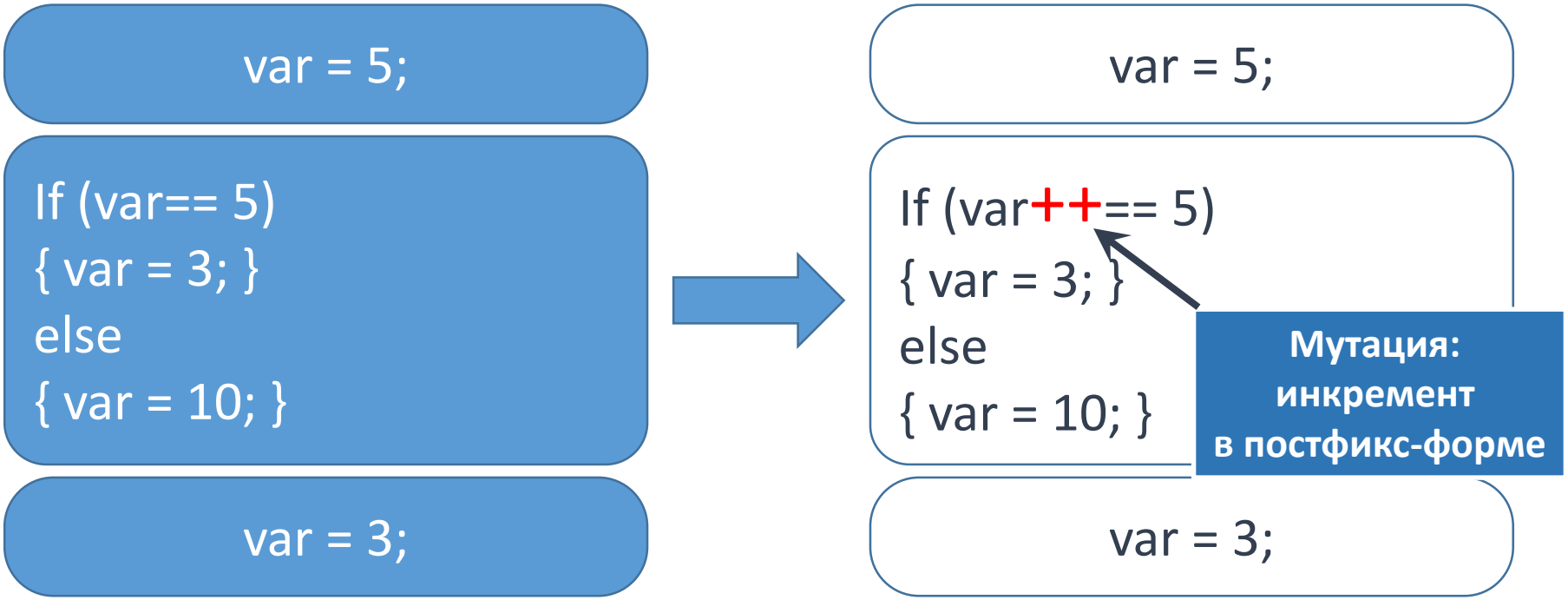
Y – выходной алфавит

T – отношение переходов

V – множество контекстных переменных



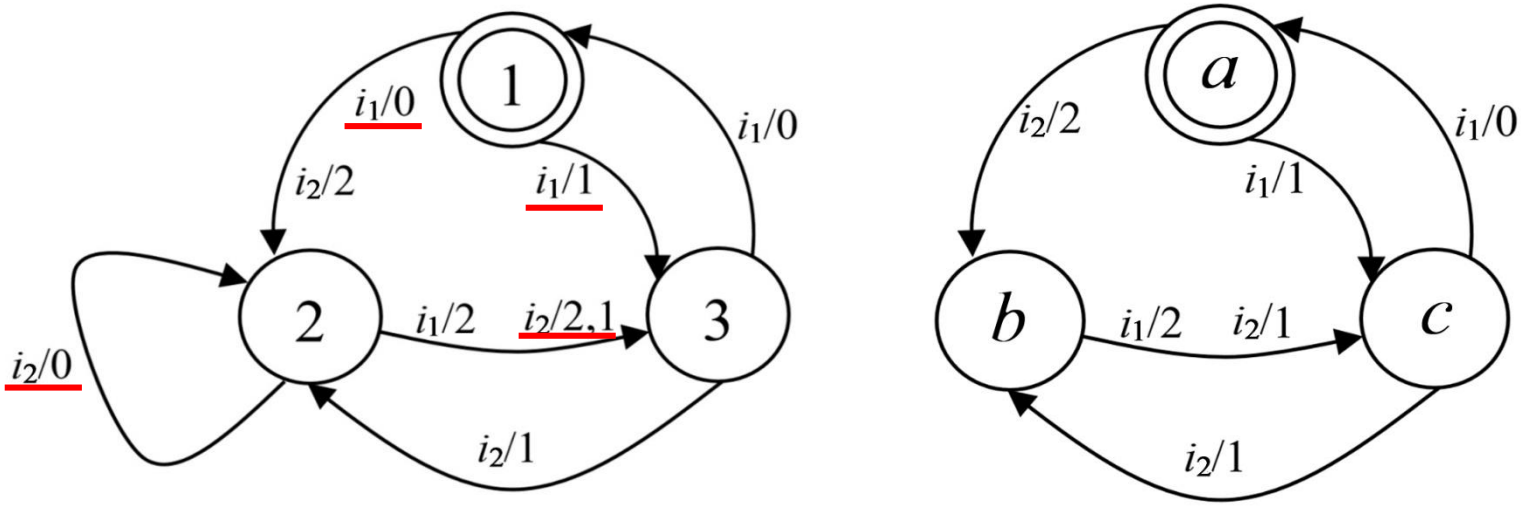
Мутант *обнаруживается* тестовой последовательностью, если реакция мутанта на тестовую последовательность не соответствует реакции спецификации



Мутант называется *эквивалентным* спецификации, если внесенная мутация не влияет на поведение программы
(для детерминированных спецификаций)

Отношения редукции и эквивалентности

Состояние p полностью определенного автомата $P = (P, I, O, T_P)$ называется ***редукцией*** состояния s полностью определенного автомата $S = (S, I, O, T_S)$, если для любой входной последовательности α справедливо $out_P(p, \alpha) \subseteq out_S(s, \alpha)$



Автомат P называется ***редукцией*** автомата S, если каждое состояние автомата P является редукцией некоторого состояния автомата S

Состояния p и s автоматов P и S называются ***эквивалентными***, если p есть редукция s, и s есть редукция p

Автоматы P и S ***эквивалентны***, если для каждого состояния автомата P существует эквивалентное состояние в автомате S, и обратно

Инициальные автоматы P и S ***эквивалентны*** (P является ***редукцией*** S), если свойство выполняется для начальных состояний автоматов

Глава 2. Синтез проверяющих тестов с гарантированной полнотой по модели расширенного автомата

Вход: Спецификация программного обеспечения в виде расширенного автомата

Выход: Множество тестовых последовательностей с гарантированной полнотой

Шаг 1. По расширенному автомату-спецификации строится **“шаблонная” программная реализация**. Такая программа сохраняет все контекстные переменные, входные/выходные параметры и содержит состояния как метки

Шаг 2. По расширенному автомату-спецификации **строится начальный тест** одним из известных (достаточно простых) методов

Шаг 3. По «шаблонной» программной реализации с помощью генератора мутантов *μJava* **строится множество мутантов**

Шаг 4. На построенные мутанты подается начальный тест, и каждый необнаруженный мутант отображается в **мутант расширенного автомата-спецификации**

Шаг 5. Для двух автоматов, автомата-спецификации и автомата-мутанта, **строится различающая последовательность** (если существует), которая добавляется к тесту

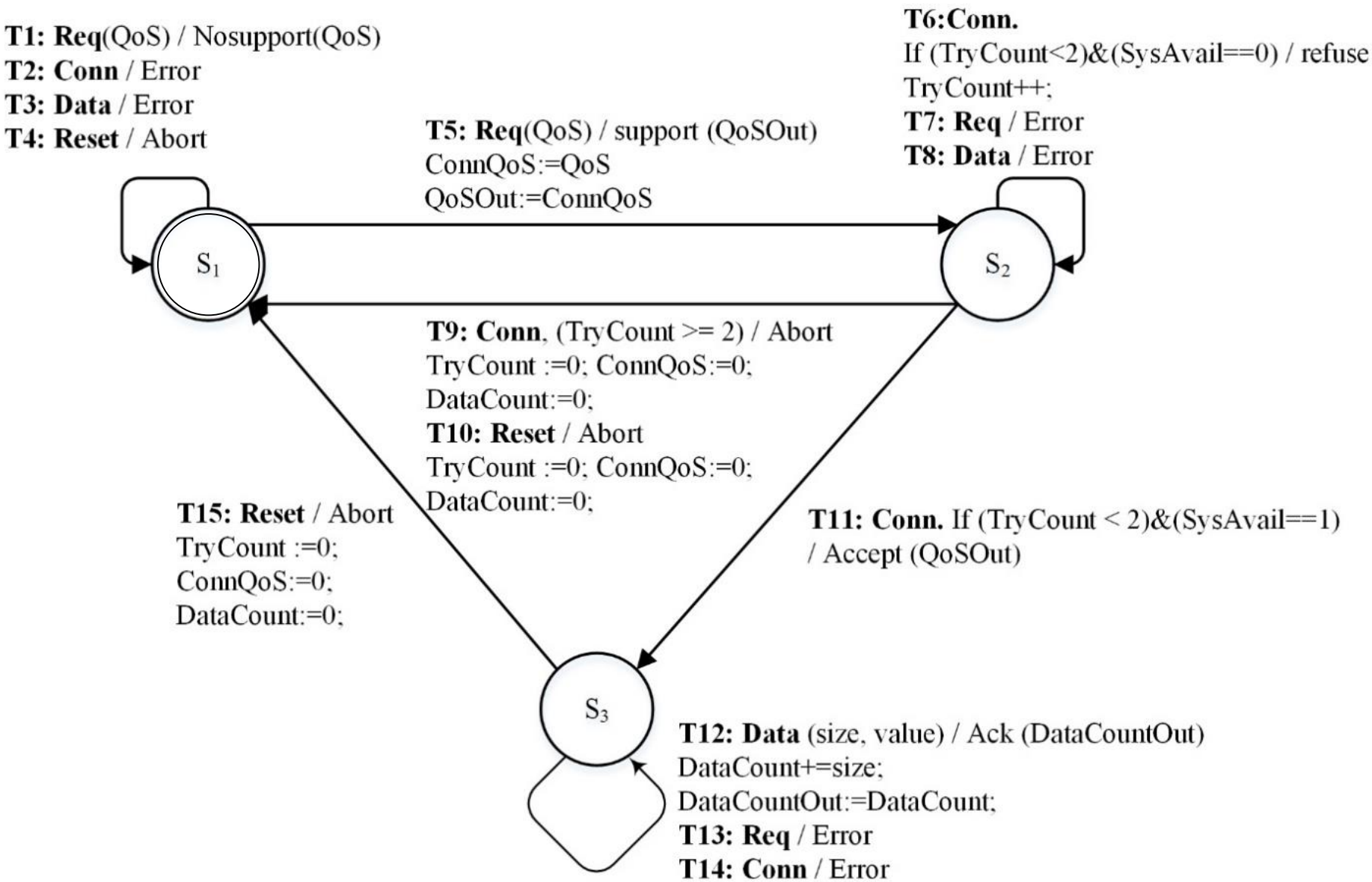
Достаточно часто неопределенные в расширенном автомате переходы доопределяются как переходы в безразличное (*don't_care*) состояние

Утверждение 2.1. Если существуют детерминированные конечные автоматы, моделирующие поведение расширенных автомата-спецификации и каждого построенного мутанта, то алгоритм, содержащий выше описанные шаги, возвращает полный проверяющий тест относительно редукции, т.е. построенный тест обнаруживает каждый мутант, поведение которого отличается от спецификации на некоторой определенной в спецификации (параметризированной) входной последовательности

Конечно-автоматные абстракции

Чтобы отличить два расширенных автомата, используются конечно автоматные абстракции расширенного автомата, и задача сводится к построению (адаптивной) различающей последовательности для двух конечно автоматных моделей

- Моделирование расширенного автомата на последовательностях конечной длины
- Полное удаление всех предикатов и параметров
- Предикатные абстракции расширенного автомата (удаляются предикаты)
- Контекстно свободные срезы


T1: Req(QoS) / Nosupport(QoS)
T2: Conn / Error
T3: Data / Error
T4: Reset / Abort
T5: Req(QoS) / support (QoSOut)
ConnQoS:=QoS
QoSOut:=ConnQoS
T6:Conn.
If (TryCount<2)&(SysAvail==0) / refuse
TryCount++;
T7: Req / Error
T8: Data / Error
S1
S2
T9: Conn, (TryCount >= 2) / Abort
TryCount :=0; ConnQoS:=0;
DataCount:=0;
T10: Reset / Abort
TryCount :=0; ConnQoS:=0;
DataCount:=0;
T15: Reset / Abort
TryCount :=0;
ConnQoS:=0;
DataCount:=0;
T11: Conn. If (TryCount < 2)&(SysAvail==1)
/ Accept (QoSOut)
S3
T12: Data (size, value) / Ack (DataCountOut)
DataCount+=size;
DataCountOut:=DataCount;
T13: Req / Error
T14: Conn / Error

Сгенерированные мутанты

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

Запуск начального теста (обход графа переходов)
- **62 мутанта** (**24,6**%) сработали **в соответствии** с расширенным автоматом-спецификацией
- Для необнаруженных тестом мутантов были построены расширенные автоматы-мутанты

С использованием конечно автоматных абстракций выяснилось, что

9 (**3,6%**) мутантов различимы с расширенным автоматом-спецификацией

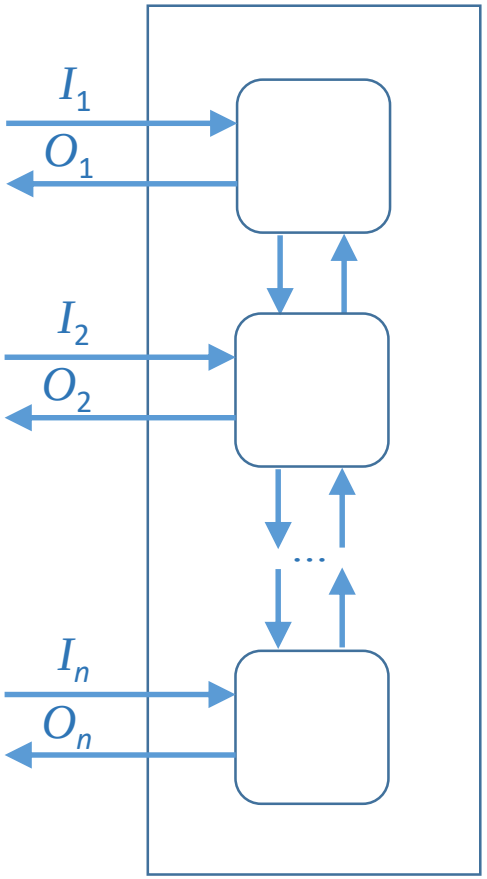
Остальные **53** (**21%**) мутанта не вносили изменений в поведение расширенного автомата-спецификации

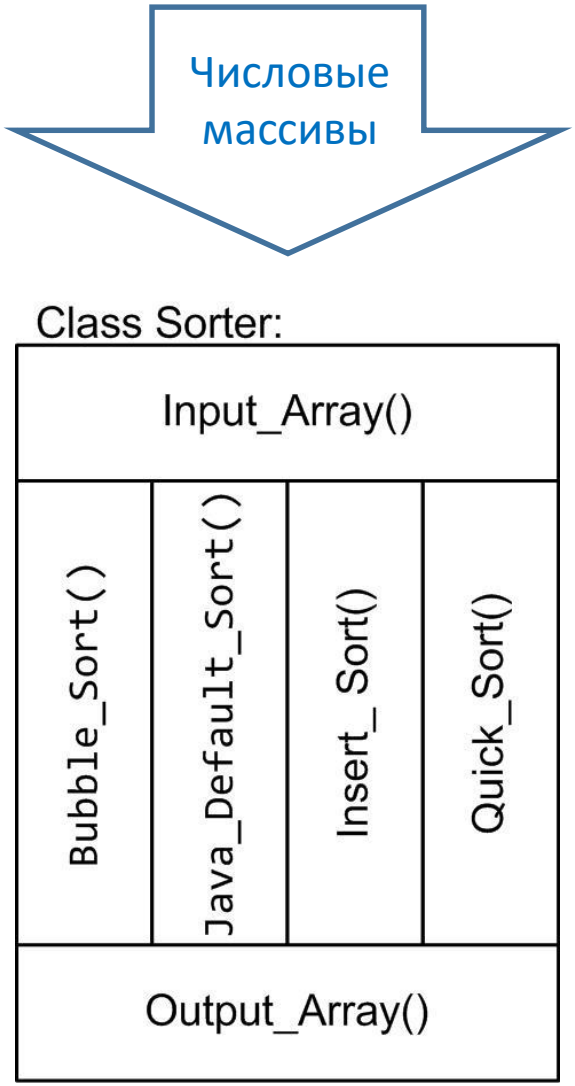
Тест достроен путем добавления трех различающих последовательностей суммарной длины 11
Длина теста увеличилась с 18 до 29 входных символов

Полнота теста выросла с **75,4**% до **100**%
(относительно мутантов, сгенерированных с использованием инструмента μ*Java*)

Сколько тестовых последовательностей, на которые реакция не соответствует спецификации, проходят через каждую компоненту?

Компонента, через которую проходит большее число тестовых последовательностей, на которые реакция не соответствует спецификации, неисправна с высокой вероятностью





Фрагмент метода *Bubble_Sort*():

Эталон

```
double tmp = arr[j];
arr[j] = arr[j+1];
arr[j+1] = tmp;
```

Мутация

```
int tmp = (int) arr[j];
arr[j] = arr[j+1];
arr[j+1] = tmp;
```

Распределение весов по строкам ПО

Line	Weight
200	-4
101	-34
104	-34
107	-34
120	-34
26	-4
29	-4
30	-4
31	-4
143	-34
146	-34
157	-4
161	-4
191	-10
209	-10
218	-10

1. Предложен алгоритм повышения полноты тестов, построенных по расширенному автомату, за счет использования инструмента μ*Java* для мутационного тестирования в «шаблонной» программной реализации

2. Тесты, построенные как обход графа переходов расширенного автомата-спецификации, оказались неполными относительно ошибок, вносимых инструментом μ*Java*

3. Для полного обнаружения ошибок, вносимых инструментом μ*Java*, тест дополнялся различающими последовательностями для конечно автоматных абстракций спецификации и необнаруженного мутанта

4. Проведенные эксперименты показали, что такой подход позволил выявить мутанты, соответствующие спецификации, а также достроить проверяющий тест различающими последовательностями для неконформных мутантов

5. Предложен подход к обнаружению ошибочных компонент в сетях из конечных автоматов. Компьютерные эксперименты показали, что идея, лежащая в основе представленного подхода, достаточно перспективна для обнаружения ошибок в программном обеспечении

Глава 3. Построение проверяющих последовательностей для недетерминированных автоматов

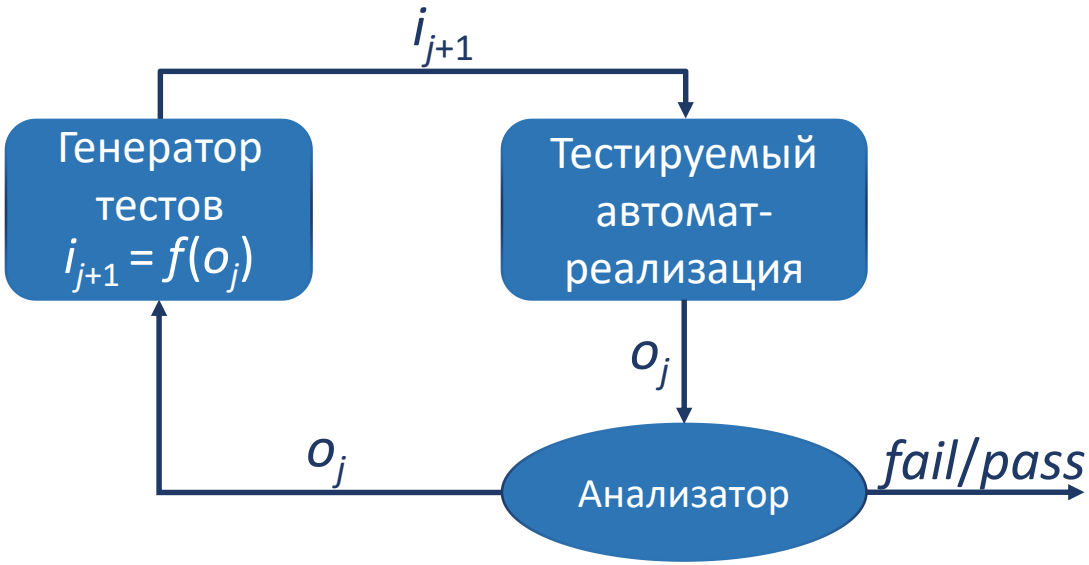
Автомат-спецификация

- полностью определенный наблюдаемый, возможно недетерминированный автомат с ***n*** состояниями

Проверяемый автомат-реализация

- полностью определенный детерминированный автомат, имеющий не более ***n*** состояний

Требуется разработать адаптивную стратегию построения проверяющей последовательности, которая проверяет, является ли тестируемая реализация редукцией спецификации



В случае адаптивной стратегии следующий входной символ определяется по выходной реакции тестируемой реализации на предыдущие входные символы

Тестовые последовательности, построенные посредством адаптивного эксперимента с тестируемой реализацией, обычно короче, чем тестовые последовательности, синтезированные заранее

Раздел 3.1. Построение адаптивной проверяющей последовательности при наличии разделяющей последовательности

Автомат-спецификация обладает **разделяющей** последовательностью и для любой пары состояний существует ***д*-передаточная последовательность**

Раздел 3.2. Построение адаптивной проверяющей последовательности с использованием (адаптивного) различающего примера

Автомат-спецификация обладает **различающим тестовым примером и** для любой пары состояний существует ***д*-передаточная последовательность**

Раздел 3.3. Построение адаптивной проверяющей последовательности с использованием различающего тестового примера и уникально достижимых состояний

Автомат-спецификация обладает **различающим тестовым примером** и для любой пары состояний существует **адаптивная передаточная последовательность**

Состояния автомата S называются ***разделимыми***, если существует входная последовательность α, такая, что множества выходных последовательностей в любых двух состояниях s и p автомата S на последовательность α не пересекаются, т.е. $out_S(s, i) \cap out_P(p, i) = \varnothing$

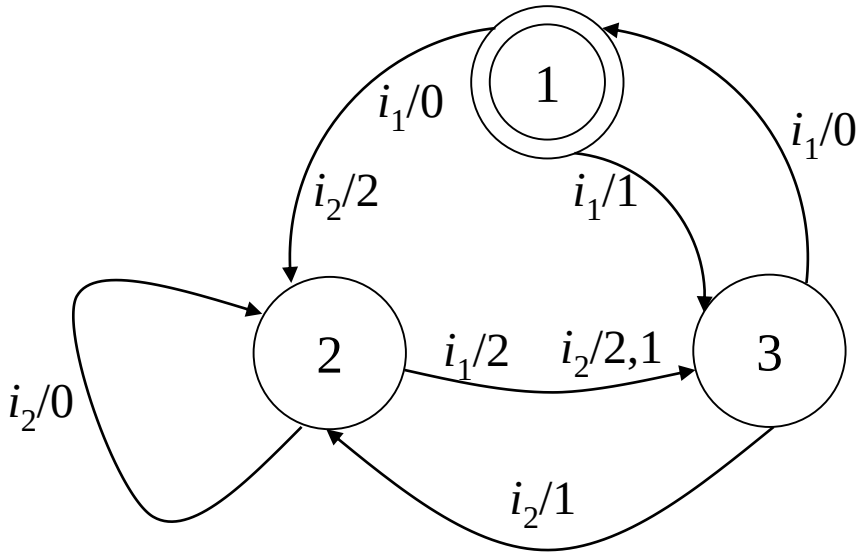
Последовательность α называется ***разделяющей*** для автомата S, если эта последовательность является разделяющей для любых двух различных состояний автомата

Состояние s' автомата S называется ***детерминировано достижимым*** (***д-достижимым***) из состояния s, если существует входная последовательность α, такая, что для любой последовательности $\beta \in out_S(s, \alpha)$, последовательность α/β переводит автомат из состояния s в состояние s'

Последовательность α в этом случае называется ***д-передаточной*** последовательностью из состояния s в состояние s' (обозначение: $\alpha_{ss'}$)

Если автомат-спецификация обладает разделяющей последовательностью и все состояния попарно *д*-достижимы, а реализация является детерминированным автоматом, число состояний которого не превосходит число состояний спецификации, то существует исчерпывающая адаптивная стратегия построения проверяющей последовательности по автомату-реализации, на которую выдается вердикт *pass*, если и только если *р*еализация является редукцией спецификации

Утверждение 3.1. Полностью определенный детерминированный автомат *P* не более чем с *n* состояниями является редукцией полностью определенного наблюдаемого автомата *S* с *n* состояниями, который обладает разделяющей последовательностью и все состояния которого попарно *д*-достижимы, если и только если автомат *P* имеет *n* состояний и изоморфен подавтомату автомата *S*



$i_2 i_1$ - разделяющая последовательность

Состояние	Выходные реакции на $i_2 i_1$
1	22
2	02, 20, 10
3	12

***д*-передаточные последовательности**

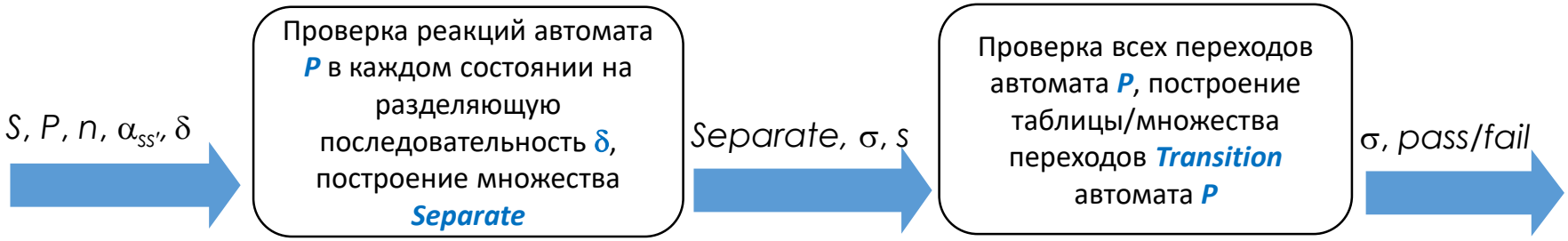
$\alpha_{12} = i_2$, $\alpha_{23} = i_1$, и $\alpha_{32} = i_2$

Утверждение 3.2. Если полностью определенный детерминированный автомат *P* не более чем с *n* состояниями является редукцией полностью определенного наблюдаемого автомата *S* с *n* состояниями, который обладает разделяющей последовательностью и все состояния которого попарно *д*-достижимы, то разделяющая последовательность является диагностической последовательностью в автомате *P*

Алгоритм построения адаптивной проверяющей последовательности

Вход: Автомат S с n состояниями, разделяющая последовательность δ для автомата S, ∂-передаточные последовательности $\alpha_{ss'}$ для каждой пары различных состояний s и s', полностью определенный детерминированный автомат P с числом состояний не более n

Выход:
- Вердикт *pass*, если P есть редукция S
- Вердикт *fail*, если P не есть редукция S
- Проверяющая последовательность σ для автомата P



Если перед переходом ко второй части алгоритма не было выдано сообщение, что предъявленный автомат не является редукцией автомата S, то можно сделать следующие выводы о предъявленном автомате P:

- автомат P имеет в точности n состояний

- разделяющая последовательность является диагностической для проверяемого автомата и может быть использована для проверки конечных состояний переходов в автомате P

Алгоритм 3.1. Построение множества Separate/ Проверка состояний в тестируемом автомате-реализации

Вход. Автомат-спецификация S, разделяющая последовательность δ и ∂-передаточные последовательности $\alpha_{ss'}$, проверяемый автомат P, текущая проверяющая последовательность σ

Выход: Множество *Separate* или сообщение, что предъявленный автомат не является редукцией автомата S

σ**: =** ε;

Separate: **=** $\varnothing$;

s: = s_0;

flag: = 0;

Шаг 1.

Пока *flag* = 0 выполнить:

{

Подать на автомат P последовательность δ;

σ**: =** $\sigma\delta$;

Если реакция ρ на δ автомата P не содержится в множестве реакций автомата S на последовательность δ,

то выдать вердикт '*fail*', сообщение, что предъявленный автомат не является редукцией автомата S, **КОНЕЦ** алгоритма;

Иначе определить преемник s' состояния s по траектории δ/ρ;

Если в множестве *Separate* есть тройка (s, ρ, s'), то *flag* : = 1;

/*произошло зацикливание по разделяющей последовательности;*/

Иначе занести тройку (s, ρ, s') в множество *Separate*;

s: = s';

}

Шаг 2.

Если мощность множества *Separate* равна n, **то КОНЕЦ** алгоритма;

/*установлено взаимно однозначное соответствие между состояниями автоматов *P* и *S* */

Иначе определить состояние s', для которого не проверена выходная реакция на последовательность δ,

т.е. в множестве *Separate* нет тройки (s', *, *);

{

Подать $\alpha_{ss'}$ на автомат P

σ**: =** $\sigma\ \alpha_{ss'}$;

}

Если реакция автомата P не содержится в множестве реакций автомата S на последовательность $\alpha_{ss'}$,

то выдать вердикт '*fail*', сообщение, что предъявленный автомат не является редукцией автомата S, **КОНЕЦ** алгоритма;

Иначе Шаг 1.

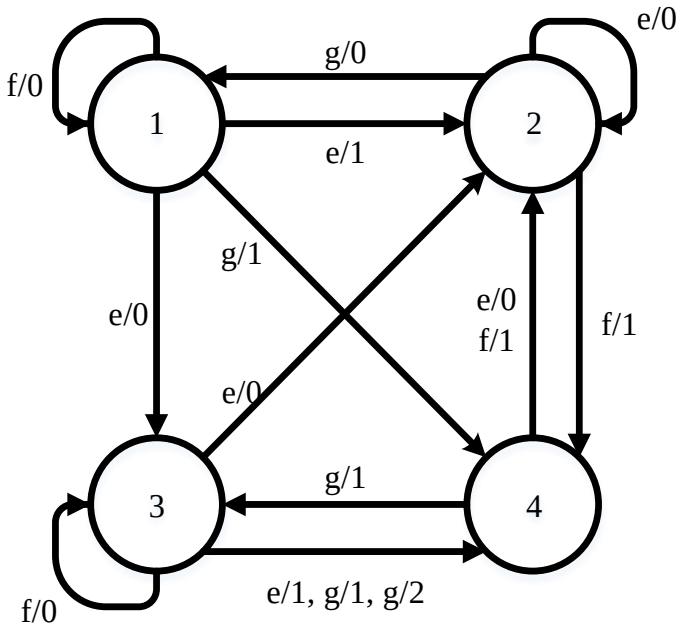
Алгоритм 3.2. Построение множества Transition/ Проверка всех переходов в тестируемом автомате-реализации

Вход. Автомат-спецификация *S*, разделяющая последовательность δ и текущее состояние автомата после подачи первой части проверяющей последовательности на проверяемый автомат *P*, множество *Separate*;
Выход. Вердикт *pass*, если *P* есть редукция *S*, или вердикт *fail*, если *P* не есть редукция *S*; множество переходов автомата *P*;

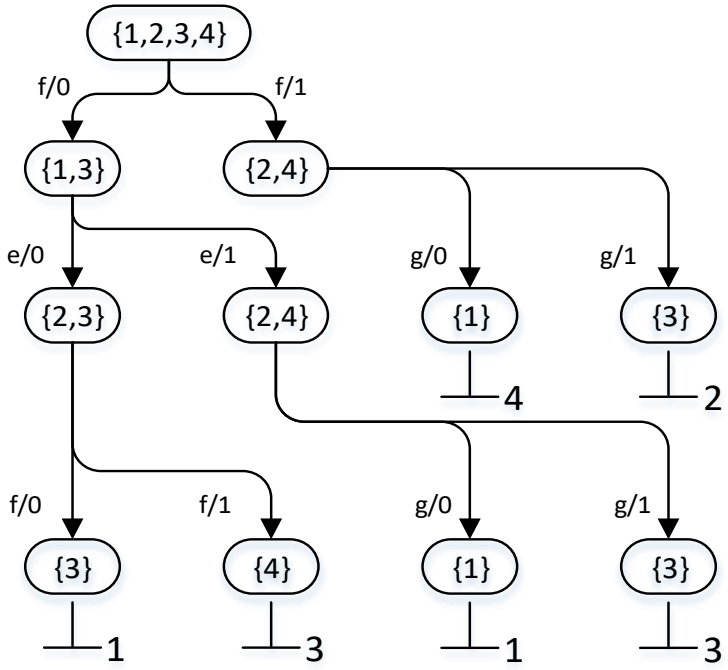
Transition: **=** $\varnothing$;
Пока мощность множества *Transition* не равна произведению *n* на число входных символов выполнить:
{
Если в текущем состоянии *s* есть переход по входному символу *i*, не входящий в множество *Transition*, **то**
Подать на автомат *P* последовательность *i* δ;
σ**: =** σ *i* δ;
Если реакция $o\rho$ автомата *P* на последовательность *i* δ не содержится в множестве реакций автомата *S* на эту последовательность, **то** выдать вердикт '*fail*', и сообщение, что предъявленный автомат не является редукцией автомата *S*, КОНЕЦ алгоритма;
Иначе по множеству *Separate* определяем тройку (s', ρ, s'') $\in$ *Separate*;
Занести четверку (*s*, *i*, *o*, s') в множество *Transition*;
s: = s'';
Иначе по множеству *Transition* найти входную последовательность γ для перехода из текущего состояния *s* в состояние s', в котором есть переход по входному символу *i*, не входящий в множество *Transition*;
/* Последовательность существует, т.к. по свойствам автомата-спецификации *S*, редукциями *S* могут быть только автоматы с тем же числом состояний*/
Подать на проверяемый автомат последовательность γ;
σ**: =** σ γ;
s: = s';
}
Выдать вердикт 'pass', и сообщение, что предъявленный автомат является редукцией автомата *S*, **КОНЕЦ** алгоритма;

Различающий тестовый пример – ациклический автомат, который содержит вход-выходные последовательности, различающие каждую пару состояний

Тестовый пример есть связный наблюдаемый инициальный автомат $T = (T, I, O, h_T, t_0)$, граф переходов которого ациклический и в каждом нетупиковом состоянии определены переходы только по одному входному символу со всеми возможными выходными символами



Автомат-спецификация *S*



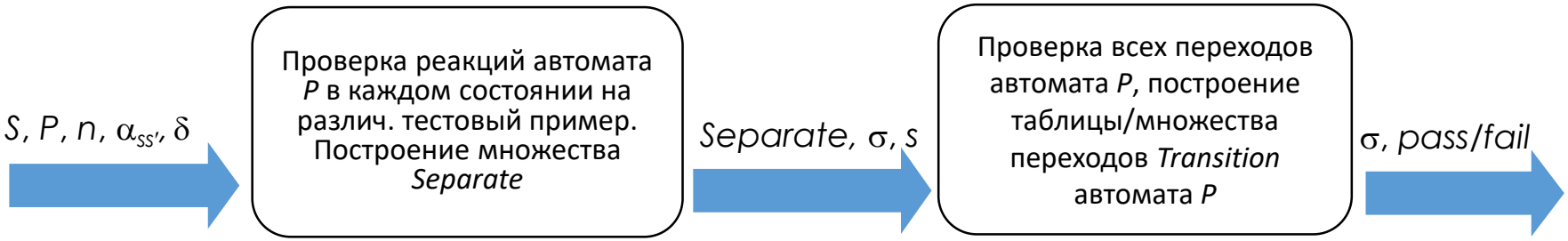
Различающий тестовый пример *T* для *S*

! В автомате – спецификации отсутствует разделяющая последовательность!

Утверждение 3.4. Пусть наблюдаемый полностью определенный автомат S с n состояниями обладает различающим тестовым примером и является ∂-связным. Полностью определенный наблюдаемый автомат P не более чем с n состояниями является редукцией S, если и только если P изоморфен подавтомату автомата S

Алгоритм аналогичен алгоритму 3.1

Отличие состоит в том, что **вместо разделяющей последовательности** мы применяем **различающий тестовый пример**, и соответственно в множество *Separate* вместо реакций на диагностическую последовательность **записываются вход-выходные последовательности для идентификаторов состояний**

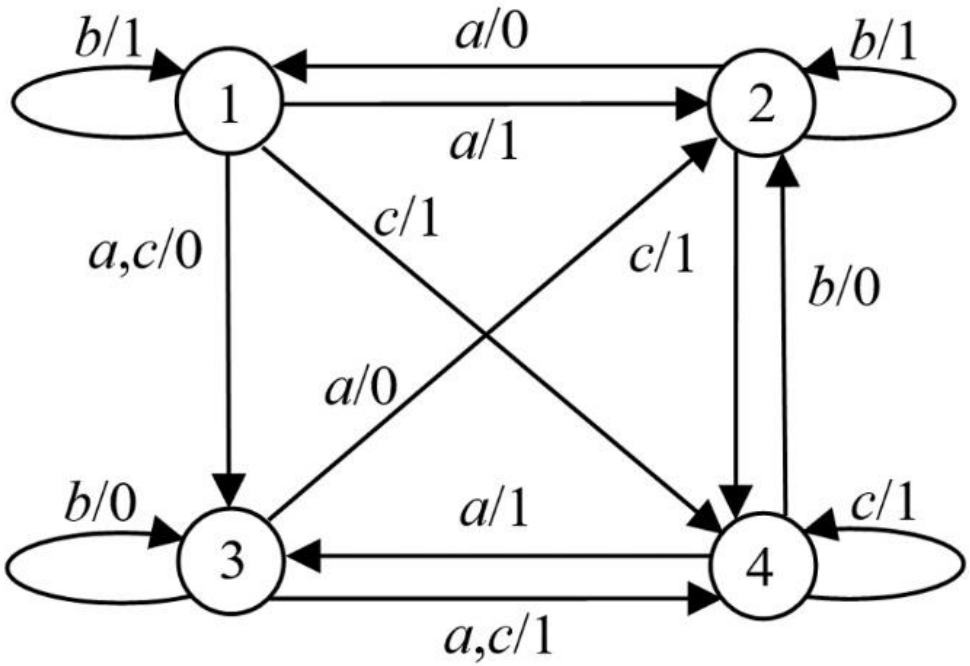


Утверждение 3.5. Пусть автомат-спецификация S обладает различающим тестовым примером DTC и детерминированный полностью определенный автомат P является DTC-совместимым с S, автомат S является ∂-связным и число состояний P не превышает числа состояний автомата S. Для каждого состояния p автомата P, различающий тестовый пример DTC имеет полную трассу α/β, которая есть трасса в состоянии p; более того, α есть **идентификатор** состояния p в автомате P

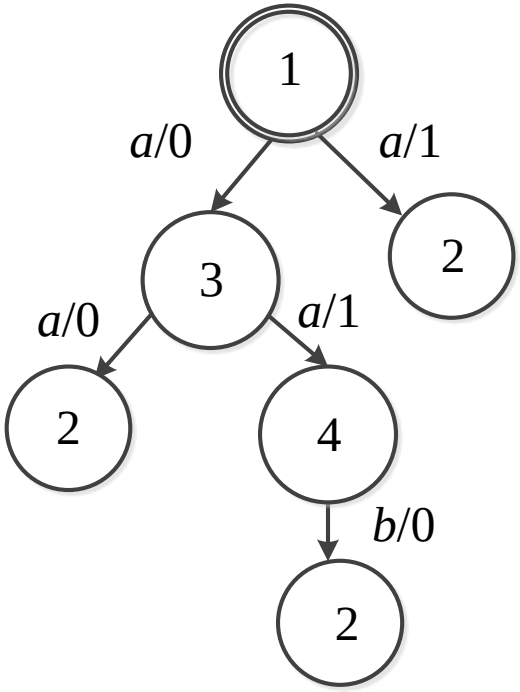
Состояние $s' \in S$ ***уникально достижимо*** из состояния $s \in S$, если существует тестовый пример $T_{s,s'}$ со следующими свойствами

- Начальное состояние соответствует подмножеству {s}
- Для каждой вход-выходной последовательности γ примера $T_{s,s}$ из начального в тупиковое состояние соответствующим преемником множества {s} является множество {s'}

Автомат-спецификация



Адаптивная передаточная последовательность из состояния 1 в 2



Если в проверяемом автомате реализовано состояние s спецификации, то и любое уникально достижимое состояние s' из состояния s тоже должно быть реализовано в проверяемом автомате

Утверждение 3.6. Полностью определенный детерминированный автомат P является редукцией полностью определенного наблюдаемого автомата S с n состояниями, обладающего различающим тестовым примером, все состояния которого попарно уникально достижимы, если и только если автомат P имеет n состояний и изоморфен подавтомату автомата S

На основании утверждения 3.6 **алгоритм аналогичен алгоритму 3.1**

Предложена адаптивная стратегия построения проверяющей последовательности для случая, когда автомат-спецификация и проверяемый автомат являются полностью определенными автоматами: автомат-реализация является детерминированным автоматом, число состояний которого не превышает число состояний автомата-спецификации, и отношением конформности является отношением редукции

Разработаны **3 алгоритма** построения **адаптивной проверяющей последовательности** на основе:

- *разделяющей последовательности + д*-передаточных последовательностей
- *различающего тестового примера + д*-передаточных последовательностей
- *различающего тестового примера + адаптивных* передаточных последовательностей

Если длина разделяющей последовательности (различающего тестового примера) не слишком большая, то и проверяющая тестовая последовательность оказывается не слишком длинной; ее длина пропорциональна произведению $n \times |I|$

Глава 4. Проверка безопасности ПО с использованием верификаторов

Уязвимость – свойство ПО, обуславливающее возможность реализации угроз нарушения конфиденциальности и целостности обрабатываемой информации

В работе рассматривались следующие типы уязвимостей:

- уязвимость переполнения типа
- уязвимость переполнения приведения типа
- уязвимость переполнения в массивах
- отрицательное переполнение в массивах
- повторное освобождение памяти
- повторное выделение памяти

Цель

Разработать метод обнаружения уязвимостей типа переполнения буфера в *C/C++* программах на основе верификации, сочетающий в себе достоинства статического и динамического подходов к обнаружению уязвимостей в программном обеспечении (ПО)

Задачи, решаемые в главе 4

1. **Анализ существующих методов** поиска уязвимостей и соответствующих программных продуктов

2. **Разработка метода поиска уязвимостей** в ПО, основанного на использовании верификаторов

3. **Программная реализация** предложенного метода и проведение компьютерных экспериментов по анализу эффективности предложенного метода

Статические и динамические методы

Статические методы	Динамические методы
Анализ программного кода без непосредственного исполнения программы	Требуется исполнение программы
Ориентированы на узкий класс программ и уязвимостей	**Часто сводятся к посылке на программу случайных входных данных**

Проанализированы программные средства по проверке наличия уязвимостей в программах на языках высокого уровня

Проанализировано **6** статических анализаторов

ITS4, *Graudit*, *Flawfinder*, *Cqual*, *Aegis*, *CppCheck*

2 динамических инструмента тестирования
Valgrind, *Dmalloc*

Верификация – формальное доказательство того, что поведение программы соответствует формализованному представлению требований, которые должны быть удовлетворены при разработке программы

Spin

- Автоматический поиск выполнимости заданного свойства
- Работает с языком ***Promela***
- Отсутствие автоматических трансляторов из ***C*** в ***Promela***

Java Path Finder

- Автоматический поиск выполнимости заданного свойства
- Работает с языком ***Java***
- Существуют трансляторы ***C++*** в ***Java*** (*C++ to Java Converter*)

Вход: программная реализация и список искомых уязвимостей

Выход: информация о наличии уязвимости в программном коде и входные параметры (контрпримеры), обнаруживающие данный факт

Шаг 1. Исходная программа представляется на языке, который требуется для верификатора

Шаг 2. В соответствии со списком уязвимостей последовательно формулируются соответствующие утверждения, которые внедряются в текст программного кода

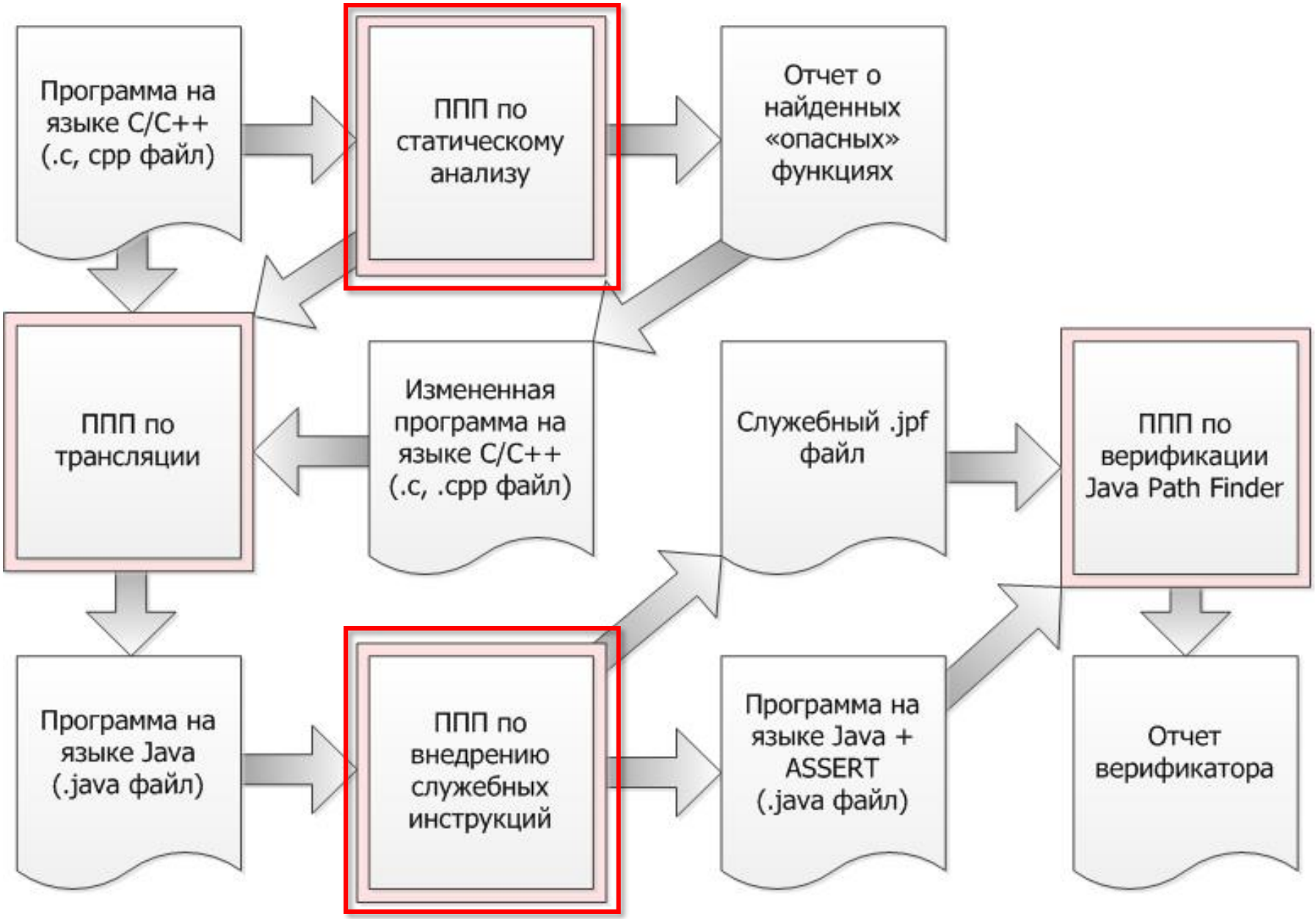
Шаг 3. Текст программного кода с внедренным утверждением подается на вход верификатора, который запускает поиск входных параметров (контрпримеров), приводящих к нарушению искомого утверждения (если таковые существуют)

Шаг 4. На вход исходной программы подается полученный контрпример и начинается выполнение программы

Если в программе произошел сбой, т.е. «оракул» выдает сообщение об ошибке, то в программе существует соответствующая уязвимость

Если сообщения об ошибке не было, то возможны два варианта:

- обнаруженная уязвимость не является уязвимостью по мнению «оракула»
- уязвимость образовалась как результат трансляции кода в язык верификатора


Программа на языке C/C++ (.c, cpp файл)
ППП по статическому анализу
Отчет о найденных «опасных» функциях
ППП по трансляции
Измененная программа на языке C/C++ (.c, .cpp файл)
Служебный .jpf файл
ППП по верификации Java Path Finder
Программа на языке Java (.java файл)
ППП по внедрению служебных инструкций
Программа на языке Java + ASSERT (.java файл)
Отчет верификатора

Название	Версия	Интерфейс	Язык UI	Поддерживаемые языки	Переполнение типа	Переполнение в массиве
Aegis	Платный	GUI	Рус	C/C++	-	+
CppCheck	FREE	GUI	Анг	C/C++	-	-
Cqual, Graudit, Flawfinder, ITS4	FREE	Консоль	Анг	C/C++	-	-
Daikon, Dmalloc, Valgrind	FREE	Консоль	Анг	C/C++	-	-

Используемые верификаторы

ППП на основе верификатора JavaPathFinder		GUI	Рус	C/C++	+	+
SPIN	FREE	Консоль	Анг	Promela	+	+

1. Статические и динамические анализаторы (за исключением платного анализатора *Aegis*) не обнаружили уязвимостей типа переполнения буфера в тестовых программах

2. Верификатор *Spin* обнаружил все уязвимости, однако отсутствуют средства автоматической трансляции *C* программ в *Promela*, соответственно перевод С программы в *Promela* осуществлялся «вручную»

3. Реализован комплекс прикладных программ на основе верификатора *JavaPathFinder для* обнаружения уязвимостей типа переполнения буфера Разработанный ППП обнаружил все уязвимости переполнения буферов в тестовых С реализациях различных комбинаторных методов

1. **Алгоритм повышения полноты тестов**, построенных по расширенному автомату, с использованием инструментов мутационного тестирования; построенные тесты обнаруживают ошибки разработчика, которые не обнаруживаются тестами, ориентированными на покрытие путей, условий, операторов, и др.

2. **Алгоритм локализации неисправной компоненты** в многокомпонентной автоматной сети на основе трассировки исполнения программы; предложенный алгоритм использует метрику, которая вычисляет количество тестовых последовательностей, проходящих через данную компоненту с эталонными и ошибочными реакциями и оказался достаточно эффективным для обнаружения ошибочных инструкций в много модульном программном коде.

3. **Метод построения адаптивной проверяющей последовательности** для полностью определенного, возможно недетерминированного автомата относительно редукции. Такая последовательность обнаруживает всякий автомат, не являющийся редукцией автомата-спецификации, за конечное время тестирования.

4. **Алгоритм обнаружения уязвимостей в программном обеспечении** на языках высокого уровня на основе верификатора *JavaPathFinder*, сочетающий в себе элементы верификации ПО и динамический подход к тестированию безопасности Предложенный алгоритм ориентирован на обнаружение следующих типов уязвимостей

- уязвимость переполнения типа
- уязвимость переполнения приведения типа
- уязвимость переполнения в массивах
- отрицательное переполнение в массивах
- повторное освобождение памяти
- повторное выделение памяти

Практическая значимость

Предложенные метод и алгоритмы могут быть использованы (и были использованы) при тестировании программного обеспечения, поведение которого описывается расширенным автоматом, в том числе для программных реализаций телекоммуникационных протоколов, программ автоматизированного управления и т.д.

Пакет программ для поиска уязвимостей в программном обеспечении на основе верификатора *JavaPathFinder* может быть использован (и был использован) при тестировании безопасности ПО

Результаты докладывались на:

1. 28-й Международной конференции тестирования программного обеспечения и систем (28th International Conference on Testing Software and Systems (**ICTSS'2016**)), г. Грац, Австрия, 2016 г.
2. 10-м коллоквиуме молодых ученых в области программной инженерии **SYRCoSE'2016**, Красновидово, Московская обл., 2016 г.
3. 7-м международном семинаре "Программные семантики, спецификации и верификация (**PSSV'2016**)", г. Санкт-Петербург, 2016 г.
4. 15-й международной конференции молодых специалистов по микро/нанотехнологиям и электронным приборам, **EDM'2014**, Новый Шарап, Новосибирская обл., 2014 г.
5. X Российской конференции с международным участием «Новые информационные технологии в исследовании дискретных структур», пос. Катунь, Алтайский край, 2014 г.
6. 16-й международной конференции молодых специалистов по микро/нанотехнологиям и электронным приборам, **EDM'2015**, пос. Чемал, Республика Алтай, 2015 г.
7. 1-м Франко-Российском семинаре по верификации, тестированию и оценке качества программного обеспечения, г. Париж, Франция, 2014 г.
8. V международной научно-практической конференции «Актуальные проблемы радиофизики» «**АПР – 2013**», г. Томск, 2013 г.
9. Всероссийской научной конференции молодых ученых «НАУКА. ТЕХНОЛОГИИ. ИННОВАЦИИ», (**НТИ-2011**), г. Новосибирск, 2011 г.
10. 5-м коллоквиуме Международном коллоквиуме молодых ученых **SYRCoSE**, Екатеринбург, **2011**г.
11. VIII Российской конференции с международным участием «Новые информационные технологии в исследовании сложных структур»(**ICAM 2010**), г. Томск, 2010 г.

Публикации (1)

1. Ермаков А. Д. Тестирование безопасности программного обеспечения с использованием верификаторов / А. Д. Ермаков // Известия высших учебных заведений. Физика, 2013. – Т.56, №9/2. – С. 181-183.
2. Ермаков А. Д. Синтез проверяющих последовательностей для недетерминированных автоматов относительно редукции / А. Д. Ермаков // Труды Института системного программирования РАН. – Т. 26, Вып. 6. – 2014. – С. 111-124.
3. Ермаков А. Д. К синтезу адаптивных проверяющих последовательностей для недетерминированных автоматов / А. Д. Ермаков, Н. В. Евтушенко // Труды ИСП РАН, 2016. – Т. 28, вып. 3. – С. 123-144.
4. Ермаков А. Д. Метод синтеза тестов с гарантированной полнотой по модели расширенного автомата / А. Д. Ермаков, Н. В. Евтушенко // Моделирование и анализ информационных систем. – Т.23, №6, 2016. – С. 728-739.
5. Ermakov A.D. Deriving Conformance Tests for Telecommunication Protocols Using µJava Tool / A. D. Ermakov // 15th International Conference of Young Specialists on Micro/Nanotechnologies and Electron Devices (EDM 2014). – Novosibirsk: Nobosibirsk State Technical University, 2014. – pp. 150-153.
6. Ermakov A. D. FSM Based Approach for Locating Faulty Software Components / A. D. Ermakov, S. A. Prokopenko. N. V. Yevtushenko // 16th International Conference of Young Specialists on Micro/Nanotechnologies and Electron Devices (EDM 2015). – Novosibirsk: Novosibirsk State Technical University, 2015. – pp. 133-136.

Публикации (2)

7. Ермаков А.Д., Динамический поиск уязвимостей в программном обеспечении на основе его верификации / А. Д. Ермаков, Н. Г. Кушик // Тезисы докладов Восьмой Российской конференции с международным участием "Новые информационные технологии в исследовании сложных структур (ICAM 2010)". – Томск: НТЛ, 2010. – С. 52.
8. Ermakov A. Detecting of C vulnerabilities / A. Ermakov, N. Kushik // Proceedings of the 5th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE-2011). – Yekaterinburg, 2011. – pp. 61.
9. Ермаков А.Д. Обзор статических и динамических методов и средств поиска уязвимостей в программном коде. / А. Д. Ермаков. НАУКА. ТЕХНОЛОГИИ. ИННОВАЦИИ //Материалы всероссийской научной конференции молодых ученых в 6-ти частях. Новосибирск: Изд-во НГТУ, 2011. Часть 1. – С. 228 – 232.
10. Ермаков А.Д. Мутационное тестирование с использованием генератора мутантов μJAVA / А. Д. Ермаков // Материалы Десятой Российской конференции с международным участием "Новые информационные технологии в исследовании сложных структур (ICAM 2014)". – Томск: Изд.дом ТГУ, 2014. – С. 50-51.
11. Ermakov A. Increasing the fault coverage of tests derived against Extended Finite State Machine / A. Ermakov, N. Yevtushenko // System informatics. – №7, 2016. – pp. 23-31.
12. Yevtushenko N. On-the-fly Construction of Adaptive Checking Sequences for Testing Deterministic Implementations of Nondeterministic Specifications / N. Yevtushenko, K. El-Fakih A. Ermakov // Proceedings of The International Conference on Testing Software and Systems (ICTSS), 2016. – pp. 139-152.

СПАСИБО ЗА ВНИМАНИЕ

Уязвимость позволяет удаленному пользователю выполнить произвольный код на целевой системе.

Microsoft Office 2007
Microsoft Office Word 2007
Microsoft Office 2010
Microsoft Word 2010

Уязвимость существует из-за недостаточной проверки границ данных при обработке таблиц в .doc файлах. Удаленный пользователь может с помощью специально сформированного .doc файла вызвать переполнение буфера в стеке и аварийно завершить работу приложения или выполнить произвольный код на целевой системе.

Android KeyStore service

Уязвимые версии: Android 4.3, возможно более ранние версии

Уязвимость позволяет удаленному пользователю вызвать отказ в обслуживании приложения.

Уязвимость существует из-за ошибки проверки границ данных в AES при шифровании ключа в функции "PKCS5_PBKDF2_HMAC_SHA1()" в сценарии /system/bin/keystore. Удаленный пользователь может вызвать переполнение буфера стека, получить доступ к определенной конфиденциальной информации и манипулировать определенными важными данными от имени пользователя.

D-Link DIR-818L
D-Link DIR-817L
D-Link DIR-868L
D-Link DIR-880L
D-Link DIR-885L
D-Link DIR-890L
D-Link DIR-895L
D-Link DIR-823
D-Link DIR-822
D-Link DIR-850L

Уязвимость существует из-за ошибки проверки границ данных в приложении cgibin, отвечающим за обработку файлов куки. Удаленный пользователь может с помощью специально сформированного запроса, отправленного на WAN-порт устройства 8181/TCP вызвать переполнение буфера и выполнить произвольный код на целевой системе, например заставить пользователя перейти на вредоносный веб-сервис.