Локализация ошибок методом сокращенного воспроизведения трассы

С. Г. Грошев
sgroshev@ispras.ru

Аннотация. В статье предложен метод построения на основе существующего теста UniTESK, находящего ошибку в тестируемой системе, минимального теста, обнаруживающего ту же ошибку. Полученный минимальный тест может использоваться для локализации ошибки в реализации тестируемой системы. Приведено математическое обоснование предложенного метода. Предложен алгоритм, реализующий его, и доказана корректность алгоритма. Описана реализация предложенного метода для инструментария тестирования CTESK.

1. Введение

Для удовлетворения непрерывно возрастающих потребностей пользователей и поддержки стабильного развития современного общества требуется разрабатывать всё более сложное программное обеспечение (ПО). Разработка всегда включает в себя множество циклов вида «тестирование — обнаружение ошибок — исправление ошибок — проверочное тестирование». Чем сложнее становится разрабатываемое ПО и чем более серьёзные задачи оно решает, тем больше в нём может происходить разнообразных внутренних и внешних взаимодействий, и в результате могут возникать всё более изощрённые ошибки. Некоторые из них проявляются только при сложных комбинациях внутренних и внешних условий, в результате их гораздо сложнее своевременно обнаруживать, локализовывать в реализации и исправлять, и тем выше их потенциальная опасность.

Ряд технологий тестирования позволяют добиваться заданного уровня тестового покрытия. К ним относится, например, UniTESK [1-4] — технология автоматизации функционального тестирования на основе формальных методов, разработанная в Институте системного программирования РАН. Однако чем сложнее сама тестируемая система и разработанные для неё тесты, тем больше информации о наблюдаемом поведении приходится анализировать в случае обнаружении ошибки, и тем сложнее понять, какая часть этой информации имеет непосредственное отношение к обнаруженной ошибке, а какая — нет. Данная статья посвящена

методу локализации ошибок путём автоматического нахождения минимальных условий, необходимых для их воспроизведения.

2. Постановка задачи

Применяемые при тестировании методы зависят от сложности тестируемой системы. В простейших системах внутреннее состояние отсутствует или несущественно для поставленной задачи тестирования, а реакции, которые они выдают, однозначно определяются полученными стимулами. В этом случае при обнаружении ошибки для её воспроизведения достаточно повторить такой же тестовый стимул, а для её локализации в коде достаточно проверить код реализации, выполняющийся при обработке данного стимула.

Но гораздо чаще поведение системы зависит не только от полученного стимула, но и от её внутреннего состояния, которое, в свою очередь, зависит от предыстории взаимодействий с системой. Тогда для воспроизведения ошибки необходимо тем или иным образом привести систему в то же состояние и применить тот же тестовый стимул, что и при её обнаружении, а для локализации — проверить код, обрабатывающий данный стимул в данном состоянии.

Для тестирования таких систем широко применяется подход, при котором тестируемая система моделируется конечным автоматом (КА) [5]. Состояния автомата соответствуют состояниям этой системы, входной алфавит — набору всех возможных воздействий на систему (стимулов), выходной алфавит — набору возможных реакций системы, а начальное состояние — состоянию, в котором тестируемая система находится (или в которое приводится) в начале теста. Задача построения тестовых последовательностей при этом сводится к построению набора путей на графе переходов модельного КА, удовлетворяющего заданным требованиям к покрытию.

Такой подход применяется и в технологии UniTESK. Для создания тестового сценария UniTESK необходимо описать только способ вычисления состояния модельного КА из состояния реализации (обычно одно модельное состояние соответствует целому классу реализационных), а для каждого модельного состояния — способ вычисления возможных в нём тестовых стимулов [6]. Всю работу по построению обхода графа переходов КА, обеспечивающего создание конкретной тестовой последовательности, берёт на себя библиотечный компонент тестовой системы, называемый обходчиком, который автоматически строит граф переходов модельного КА по мере его обхода [5,6].

Для проверки корректности работы тестируемой системы используются формальные спецификации требований к ней. Каждый раз после применения очередного тестового стимула полученные от тестируемой системы реакции анализируются на соответствие спецификации, и определяется новое состояние модельного КА. Несоответствие наблюдаемой реакции описанным

в спецификации требованиям или расхождения между ожидаемым и наблюдаемым состоянием целевой системы считается ошибкой и требует дальнейшего анализа. В результате анализа может быть принято решение о некорректности спецификации или о существовании ошибки в реализации.

Поскольку зачастую состояние системы имеет достаточно сложную структуру, а единственный способ его изменить — подавать те же стимулы, которыми пользовался тест, то возникает естественный способ приведения системы в то же состояние для воспроизведения ошибки: точное повторение последовательности стимулов, которые подавал тест, обнаруживший эту ошибку. Последовательность стимулов при этом извлекается из сгенерированной тестом трассы, а процесс повторения тестовых стимулов называется воспроизведением трассы (trace replay). В силу используемого в технологии UniTESK способа описания модельного КА, этот способ надежнее повторного прогона тестов, так как в общем случае обходчик UniTESK не гарантирует тот же порядок обхода дуг графа.

Более того, в реальных проектах часто нет возможности во время тестирования точно определить внутреннее состояние тестируемой системы. В этом случае она моделируется как черный ящик [6], состояние которого не получается напрямую из состояния тестируемой системы, а вычисляется на основании предыстории взаимодействий с ней. При этом после каждого тестового воздействия новое состояние модели вычисляется на основании предыдущего состояния, поданного стимула и полученных реакций. По разным причинам (это могут быть как ошибки реализации, так и ошибки спецификации) со временем состояние модели может перестать соответствовать состоянию реализации. Это приводит к расхождению между ожидаемым и наблюдаемым поведением тестируемой системы, причем такое расхождение может проявиться не сразу, а только в реакциях на последующие стимулы, то есть, момент совершения ошибки и момент её обнаружения могут различаться. В этом случае задачи воспроизведения и локализации ошибки существенно усложняются: поскольку у нас нет достоверного знания о внутреннем состоянии системы в момент проявления ошибки. В связи с этим может потребоваться воспроизвести всю последовательность тестовых воздействий из трассы теста, обнаружившего ошибку, и проверить весь код, который был активирован ими, и все элементы состояния реализации, которые были при этом затронуты. Поскольку к моменту обнаружения ошибки тест UniTESK мог проделать достаточно длинный и запутанный путь по графу КА, совершив много переходов и побывав в некоторых состояниях много раз, анализ на предмет ошибки всей предшествующей трассы становится слишком сложен, поэтому возникает необходимость поиска минимального теста, обнаруживающего ту же ошибку. Для решения этой задачи был разработан метод сокращенного воспроизведения трассы, описанный в данной статье.

В технологии UniTESK тестовый сценарий содержит функцию вычисления состояния модельного КА и набор сценарных методов. Сценарный метод

содержит набор итерационных переменных (возможно пустой), правила перебора их значений и способ отображения набора значений итерационных переменных в конкретное тестовое воздействие. Входным алфавитом модельного КА считается при этом пара (имя сценарного метода, набор значений его итерационных переменных).

В процессе выполнения тестовый сценарий UniTESK автоматически генерирует трассу, содержащую различную информацию о работе сценария, тестовой и тестируемой систем. Для рассматриваемого метода нас интересует следующая информация.

1. Запускаемые тестовые сценарии.
2. Модельное состояние сценария.
3. Вызываемые в рамках сценария сценарные методы и их распределение по потокам управления.
4. Значения итерационных переменных сценарных методов.

Метод сокращенного воспроизведения трасс опирается на следующие гипотезы.

I. Поведение тестируемой системы детерминировано относительно предыстории взаимодействий с ней.

II. В случае обнаружения ошибки, с наибольшей вероятностью её причина лежит в том взаимодействии с тестируемой системой, в котором она была обнаружена, или в последних предшествующих ему взаимодействиях.

III. Переход КА (поведение тестовой системы и изменение её состояния) однозначно определяется его состоянием, вызванным сценарным методом и значениями итерационных переменных этого метода. Эта гипотеза является технологическим требованием обходчика UniTESK, то есть, требованием корректности сценария.

Рассмотрим гипотезу I подробнее. Поскольку наша задача — локализация уже заведомо существующей ошибки, детерминизм поведения в данном случае не означает гарантированного совпадения поведения тестируемой системы со столь же детерминированной спецификацией, описывающей зависимость полученного состояния системы и наблюдаемых выходных реакций от предшествующего состояния и полученных стимулов. Мы всего лишь предполагаем, что каждый раз при получении одной и той же последовательности стимулов тестируемая система будет выдавать одну и ту же последовательность реакций и переходить в одни и те же состояния; при этом с точки зрения спецификации её поведение может быть недетерминированным.

Допустим, тестируемая система, находясь в состоянии, которому соответствует модельное состояние A, и получив модельный стимул S, в первый раз перешла в состояние A1 и выдала реакцию R1, а в другой раз, попав в эквивалентное состояние, также соответствующее модельному

состоянию A и получив тот же стимул S, перешла в состояние A2 и выдала реакцию R2, причем (A1, R1) ≠ (A2, R2). Тогда с точки зрения спецификации система демонстрирует недетерминированное относительно состояний и стимулов поведение. Но если при многократном воздействии одной и той же последовательности стимулов целевая система каждый раз проходит по тем же самым состояниям и выдает ту же последовательность реакций (в том числе (A1, R1) в первый раз и (A2, R2) во второй), то мы говорим о детерминизме её поведения относительно предыстории и предполагаем, что расхождение между наблюдаемым поведением и спецификацией определяется какими-то элементами состояния реализации, которые не описаны в спецификации, но изменились в промежутке между этими двумя моментами. Тогда для устранения ошибки необходимо локализовать эти элементы состояния и исправить работу с ними или исправить спецификацию. Чтобы облегчить работу по локализации обнаруженной тестом ошибки, можно найти минимальный набор предшествующих событий, после которого расхождение между реализацией и спецификацией гарантированно проявляется.

Пример

Тестируемая система — менеджер памяти со следующими методами: `alloc()` — выделение блока памяти, `free()` — освобождение блока и `optimize()` — оптимизация и дефрагментация выделенной памяти. Модельное состояние системы — объем распределённой памяти.

Ситуация 1: метод `free()` на самом деле не освобождает память. Тогда мы можем получить следующую воспроизводимую последовательность событий.

```
...
state=10
alloc() возвращает блок памяти
state=11
alloc() возвращает NULL
state=11
free()    —    Возникновение ошибки в реализации
state=10
optimize()
state=10
alloc()  возвращает  NULL  —  Проявление  ошибки,
недетерминизм
```

Ситуация 2: в модели не учтено, что метод `optimize()` может существенно изменять состояния реализации, создавая достаточные для выделения участки непрерывной памяти. В результате мы имеем воспроизводимую последовательность событий.

```
...
state=10
alloc() возвращает блок памяти
```



```
state=11
alloc() возвращает NULL
state=11
free()
state=10
optimize()    —    Существенное    изменение    состояния
реализации, не отраженное в модели
state=10
alloc() возвращает блок памяти
state=11
alloc() возвращает блок памяти — Ошибка, недетерминизм
```

В обоих случаях мы имеем тестируемую систему с детерминированным относительно предыстории поведением, которая ведёт себя недетерминировано с точки зрения модели. В первом случае это вызвано ошибкой в самой системе, а во втором случае — неточностью спецификации. Однако в обоих случаях расхождение между моделью и реализацией возникло в промежутке между двумя одинаковыми тестовыми стимулами, которые были применены в одинаковых состояниях (эти состояния и стимулы выделены жирным шрифтом), а потому с точки зрения модели должны были приводить к одинаковым результатам, но при тестировании привели к разным.

3. Определения

Назовём *трассой* упорядоченный список состояний и переходов модельного КА, обладающий следующими свойствами.

1. Трасса или пуста, или начинается с состояния и заканчивается состоянием.
2. Состояния и переходы в трассе чередуются.
3. Каждый встретившийся в трассе переход переводит модельный КА из состояния, которое непосредственно ему предшествует в трассе, в состояние, непосредственно следующее в трассе за ним.

Для удобства мы будем также использовать две сокращенные формы записи трассы: в одной будет опускаться конечное состояние трассы, в другой — все состояния. Пустая трасса в любой форме записи представляет собой пустой список.

Стандартный тест UniTESK представляет собой один активный поток выполнения [7], в котором выполняется обход модельного КА. В процессе выполнения теста в трассу UniTESK последовательно сбрасываются достигнутые состояния КА и совершённые им переходы. В случае отсутствия сбоев в тестовой системе, можно в дальнейшем извлечь из файла трассы последовательный список состояний и переходов модельного КА; назовём этот список трассой теста. Очевидно, что трасса теста является трассой согласно данному выше определению и начинается в начальном состоянии

модельного КА. Тест UniTESK имеет два стандартных режима работы: в первом после обнаружения ошибки состояние модели тестируемой системы считается недостоверным, дальнейшие вердикты тестового оракула о корректности или некорректности поведения тестируемой системы также считаются недостоверными, поэтому тестирование на этом останавливается, и трасса завершается состоянием с ошибкой. Есть и второй режим, при котором в случае, если достоверно установлено, что нарушений в состоянии тестируемой системы и расхождений между её состоянием и состоянием модели нет, то тест продолжается и может даже обнаруживать другие ошибки. Но поскольку зачастую таких гарантий у нас нет, обычно используется первый режим. Так как рассматриваемый в статье метод минимизации тестов предназначен в первую очередь для сложных случаев, когда возможно не проявляющееся сразу расхождение между состояниями тестируемой системы и модели, мы будем считать недостоверной всю трассу после обнаружения первой ошибки.

Трасса теста может содержать один и тот же переход КА несколько раз, но для работы с трассами (в силу возможного расхождения между модельным состоянием и истинным состоянием тестируемой системы, а также в соответствии с гипотезой II) нам необходимо различать вхождения одного и того же перехода КА в трассе. Поэтому переход трассы мы считаем экземпляром (instance) перехода КА, и в дальнейшем под переходом понимаем именно переход трассы. Поскольку для рассматриваемого метода неважен способ, которым совершается тот или иной переход в реализации, но важен порядок этих переходов, введём индексацию переходов трассы, пронумеровав их возрастающей последовательностью натуральных чисел. В силу гипотезы III, вся информация, необходимая для отображения индексов переходов в конкретные тестовые стимулы при последующем воспроизведении (полном или частичном) трассы, содержится в исходной трассе теста UniTESK.

Список состояний и переходов **T1** назовём *подтрассой* трассы **T**, если он является трассой согласно данному выше определению и его можно получить из **T** путём выбрасывания некоторых элементов списка без перестановки оставшихся (то есть, индексация входящих в подтрассу переходов остаётся возрастающей).

Назовём *ациклической* трассу, в которой все состояния различны.

Циклической трассой или *циклом* назовём такую трассу, у которой начальное и конечное состояния совпадают.

Простым циклом назовём такую циклическую трассу, у которой совпадают только начальное и конечное состояние, а все остальные состояния отличны от них и попарно различны.

Введём многоместную операцию сложения «+», отображающую наборы подтрасс одной исходной трассы, удовлетворяющие данному ниже предусловию операции, в подтрассу той же исходной трассы следующим образом:

1. Нульместная операция «+» всегда возвращает пустую трассу.
2. Одноместная операция «+» возвращает в качестве результата свой аргумент.
3. Для двух и более аргументов определим сумму следующим образом:
    3.1. Выпишем подряд все переходы из всех трасс, заданных в качестве аргументов операции, упорядочив их в порядке индексации.
    3.2. Если в заданном наборе в разных трассах встречаются повторяющиеся индексы — ошибка, операция сложения к данному набору трасс неприменима.
    3.3. Добавим в полученный список переходов символы состояний (один символ в начале списка, один в конце и по одному между каждыми двумя последовательными переходами) так, чтобы перед каждым переходом стояло состояние, в котором он начинается, а после каждого перехода — состояние, в котором он завершается. Если это сделать невозможно, то есть существуют два таких последовательных элемента полученного на шаге 3.1 списка переходов, что конечное состояние первого не совпадает с начальным состоянием второго — ошибка, операция сложения к данному набору трасс неприменима.

Предусловие операции сложения заключается в том, что приведённый алгоритм её вычисления не должен выдавать ошибку.

4. 4. Метод сокращенного воспроизведения трассы

Рассмотрим следующую трассу теста:
T = <A, 1, B, 2, C, 3, D, 4, E, 5, C, 6, D, 7, E, 8, B, 9, F, 10, D*>.

Здесь буквами обозначены состояния модельного КА, числами — переходы, символом «звёздочка» — ошибка. Поскольку, как было указано выше, после обнаружения ошибки состояние тестируемой системы недостоверно, мы считаем состояние D* (модельное состояние D после обнаружения ошибки) отличным от состояния D. Отметим также, что в силу этого соглашения в трассе теста, обнаружившего ошибку, конечное состояние всегда отличается от начального.

Графически эта трасса представлена на Рис. 1.

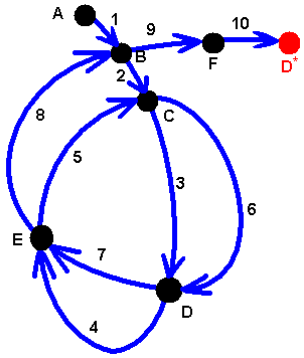

Рис. 1. Трасса выполнения теста.

Если гипотеза I выполняется, то мы можем воспроизвести всю трассу заново и убедиться, что ошибка также воспроизводится. Исходя из этой гипотезы, для локализации ошибки мы будем искать кратчайшую подтрассу трассы **T**, на которой ошибка всё ещё воспроизводится.

В данном примере кратчайший путь из начального состояния в конечное очевиден:

$\mathbf{T_1}$ = <A, 1, B, 9, F, 10, D*>.

Трасса $\mathbf{T_1}$ является кратчайшей подтрассой трассы **T**, ведущей из начального состояния в конечное. Мы можем воспроизвести эту подтрассу и проверить, повторяется ли при этом ошибка. В случае если повторить ошибку удалось, для её локализации достаточно исследовать подтрассу $\mathbf{T_1}$. В противном случае, для локализации ошибки требуется дальнейший анализ исходной трассы теста.

Запишем трассу T в следующем виде:

T = <A, 1, **<B, 2, C, 3, D, 4, E, 5, C, 6, D, 7, E, 8>**, B, 9, F, 10, D*>.

Трасса $\mathbf{T_2}$ = <B, 2, C, 3, D, 4, E, 5, C, 6, D, 7, E, 8> (или, в другой форме записи, <B, 2, C, 3, D, 4, E, 5, C, 6, D, 7, E, 8, B>) также является подтрассой трассы **T**, причем $\mathbf{T} = \mathbf{T_1} + \mathbf{T_2}$.

Как мы здесь видим, трасса $\mathbf{T_2}$ вложена в трассу $\mathbf{T_1}$, то есть сначала воспроизводится часть трассы $\mathbf{T_1}$, затем её воспроизведение прерывается, целиком воспроизводится циклическая трасса $\mathbf{T_2}$, после чего воспроизведение трассы $\mathbf{T_1}$ продолжается с той же точки. Можно представить это как вызов подпрограммы в тесте. Будем также говорить в таких случаях, что подтрасса $\mathbf{T_2}$ *вложена* в подтрассу $\mathbf{T_1}$, причем отношение вложенности будем считать транзитивным.

Очевидно, что трасса $\mathbf{T_2}$ не является простым циклом. Она разложима на сумму простых циклов, причем не единственным образом, как это показано на Рис. 2-4.

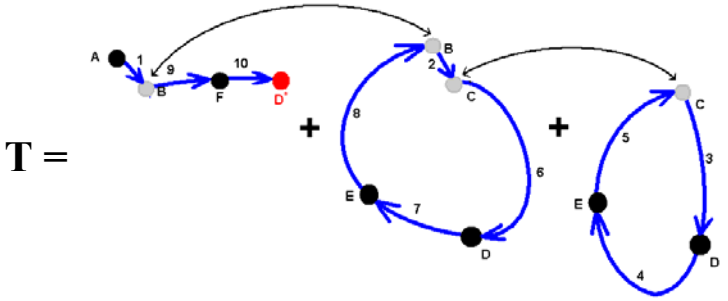

Рис. 2. Пример разложения трассы T.

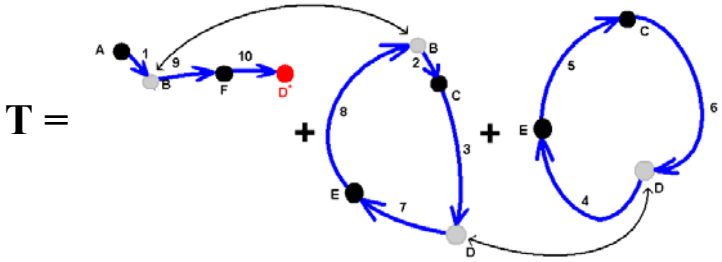

Рис. 3. второй пример разложения трассы T.

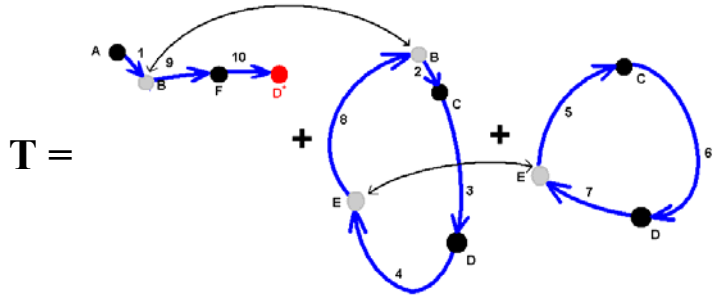

Рис. 4. Еще один пример разложения трассы T.

Все приведенные на Рис. 2-4 разбиения трассы $\mathbf{T_2}$ представимы таким же образом, как и разбиение $\mathbf{T} = \mathbf{T_1} + \mathbf{T_2}$: начало одной подтрассы, прерывание её вызовом вложенного цикла и завершение исходной подтрассы (такие вызовы показаны на Рис. 2-4 тонкими стрелками).

Рассмотрим разбиение трассы теста на сумму своих непустых подтрасс $\mathbf{T} = \mathbf{T_1} + \mathbf{T_2} + \ldots + \mathbf{T_{N-1}} + \mathbf{T_N}$, удовлетворяющее следующим требованиям.

1. Для $\forall \mathbf{k=1\ldots N}$ определена сумма трасс $\mathbf{\Sigma_k} = \mathbf{T_1} + \mathbf{T_2} + \ldots + \mathbf{T_{k-1}} + \mathbf{T_k}$, то есть к ним применима операция сложения.
2. Трассы $\mathbf{T_1} \ldots \mathbf{T_N}$ пронумерованы в порядке, обратном индексам последних переходов, входящих в эти трассы (то есть, $\forall \mathbf{i, j}$: $\mathbf{1 \le i < j \le N} \Rightarrow$ последний переход трассы $\mathbf{T_i}$ имеет индекс больший, чем последний переход из $\mathbf{T_j}$)
3. Трасса $\mathbf{T_1}$ ведёт из начального состояния трассы $\mathbf{T}$ в конечное и является простым циклом или ациклична (поскольку нас интересуют только трассы тестов, обнаруживающие ошибку, трасса $\mathbf{T_1}$ будет для них ациклична)
4. Подтрассы $\mathbf{T_2} \ldots \mathbf{T_N}$ — простые циклы.

Требование 1 гарантирует возможность инкрементального наращивания подтрасс $\mathbf{\Sigma_k}$.

Требование 2 гарантирует, что каждый раз при добавлении к воспроизводимой части трассы очередной подтрассы $\mathbf{T_k}$, $\mathbf{\Sigma_k}$ содержит всё более длинную непрерывную хвостовую цепочку переходов из исходной трассы теста, обеспечивая, согласно гипотезе II, инкрементальный поиск места, где была допущена ошибка.

Требования 3 и 4 минимизируют шаг инкрементального поиска при наращивании воспроизводимой части трассы.

Теорема 1. Для любой трассы существует разбиение на сумму подтрасс, удовлетворяющее требованиям 1-4.

Доказательство. В Приложении 1 приведён алгоритм, строящий для любой трассы разбиение на подтрассы. Докажем, что полученное в результате его работы разбиение обладает всеми требуемыми свойствами, и что элементы этого разбиения являются трассами согласно определению.

Введём на исходной трассе `fullPath` индексацию, обозначив каждый переход его порядковым номером в трассе.

Лемма. В начале каждой итерации главного цикла алгоритма переменная `currentPath` содержит трассу: корректную (согласно определению трассы), с возрастающей индексацией переходов, ациклическую. При этом если текущий шаг — не последний, то присоединение к этой трассе очередного перехода из `fullPath` корректно.

Очевидно, что в начале работы алгоритма утверждение Леммы выполняется, так как `currentPath` содержит пустой список переходов. Далее по индукции. Если в начале **i**-й итерации цикла эти условия выполняются, то после присоединения к `currentPath` очередного перехода получаем корректную трассу, причем, поскольку `fullPath` является корректной трассой, присоединение на следующем шаге очередного перехода, если он

существует, также будет корректным. Поскольку переходы входной трассы обрабатываются строго по порядку, индексация переходов также останется возрастающей.

В начале шага алгоритма переменная `currentPath` содержала ациклическую трассу, то есть, все встречающиеся в ней переходы были попарно различны. После добавления очередного перехода может возникнуть цикл, то есть, конечное состояние этого перехода может совпадать с одним из ранее встречающихся в трассе `currentPath` состояний. Если такого совпадения не произошло, то, очевидно, утверждение Леммы также выполняется. Если произошло совпадение, то конечное состояние трассы `currentPath` могло совпасть только с начальным состоянием ровно одного из её переходов (это может быть и последний переход, если добавленный переход является петлёй). Таким образом, получившийся цикл — простой. Так как этот цикл является подсписком без перестановок содержащейся в `currentPath` трассы, его индексация будет возрастающей. После удаления полученного цикла из `currentPath` мы опять получаем ациклическую трассу с возрастающей индексацией, причем её конечное состояние при этом не меняется, то есть, в конце текущей итерации (и, соответственно, в начале следующей итерации) можно будет корректно добавить очередной переход исходной трассы. Таким образом, индуктивный переход доказан, и вся Лемма доказана.

Из утверждения Леммы следует, что все элементы построенного разбиения будут корректными трассами. Одновременно мы доказали, что для построенного разбиения выполняется Требование 4.

После последней итерации алгоритм добавляет при необходимости в начало разбиения оставшийся прямой путь ($\mathbf{T_1}$) из начального состояния исходной трассы в конечное. Если начальное и конечное состояния совпадают, то цикл $\mathbf{T_1}$ будет добавлен в начало разбиения во время последней итерации. Таким образом, Требование 3 также выполняется.

На каждой итерации алгоритма к `currentPath` добавляется очередной переход из исходной трассы, и в случае обнаружения цикла этот переход входит в него. Полученный цикл добавляется в начало разбиения. Таким образом, обеспечивается выполнение Требования 2.

Докажем, что Требование 1 также выполняется построенного алгоритмом разбиения.

Очевидно, что для $\mathbf{\Sigma_1} = \mathbf{T_1}$ оно выполняется.

Рассмотрим шаг работы алгоритма, на котором к списку `pathes` добавляется элемент $\mathbf{T_k}$ ($\mathbf{2 \le k \le N}$). Непосредственно перед таким добавлением переменная `currentPath` содержит корректную трассу с возрастающей индексацией $\mathbf{\Sigma} = \mathbf{S} + \mathbf{T_k}$, где $\mathbf{S}$ соответствует некой цепочке переходов (возможно, пустой), с индексами, предшествующими индексам из трассы $\mathbf{T_k}$. Обозначим индексы входящих в цикл $\mathbf{T_k}$ переходов через $\mathbf{i_1}, \ldots, \mathbf{i_2}$.

Рассмотрим теперь шаг, на котором алгоритм обнаруживает следующий цикл, выделенный в подтрассу $\mathbf{T_{k-1}}$ (элементы искомого разбиения строятся в обратном порядке). По построению видим, что индекс каждого перехода, входящего $\mathbf{T_{k-1}}$, или больше всех индексов, входящих в $\mathbf{T_k}$, или меньше их всех. Выписав подряд в возрастающем порядке индексы всех переходов, входящих в любую из этих двух подтрасс, получаем возрастающий список индексов вида $\mathbf{<j_1, \ldots, j_2, i_1, \ldots, i_2, j_3, \ldots, j_4>}$, где индексы $\mathbf{j_1, \ldots, j_2}$ и $\mathbf{j_3, \ldots, j_4}$ (список $\mathbf{j_1, \ldots, j_2}$ может быть пуст) соответствуют переходам, принадлежащим трассе $\mathbf{T_{k-1}}$, а $\mathbf{i_1, \ldots, i_2}$ — переходам, принадлежащим трассе $\mathbf{T_k}$. По построению начальное состояние перехода $\mathbf{i_1}$ совпадает с конечным состоянием перехода $\mathbf{j_2}$, если он существует, или с начальным состоянием трассы $\mathbf{T_{k-1}}$ в обратном случае, а конечное состояние перехода $\mathbf{i_2}$ совпадает с начальным состоянием перехода $\mathbf{j_3}$. Таким образом, для любого номера $\mathbf{k=2..N}$ определена сумма трасс $\mathbf{T_{k-1} + T_k}$, причем начальное и конечное состояния этой суммы трасс совпадают соответственно с начальным и конечным состояниями трассы $\mathbf{T_{k-1}}$. Можно сказать также, что каждая следующая подтрасса в полученном разбиении или вложена в предыдущую, или непосредственно предшествует ей в исходной трассе.

В силу доказанного свойства, не существует индекса, относящегося к трассе $\mathbf{T_{k-2}}$ и лежащего между двумя индексами, относящимися к трассе $\mathbf{T_{k-1}}$. Следовательно, добавление трассы $\mathbf{T_{k-2}}$ к сумме трасс $\mathbf{T_{k-1} + T_k}$ не нарушает предусловие операции сложения трасс, то есть, для $\mathbf{\forall k \geq 3}$ определена также и сумма трасс $\mathbf{T_{k-2} + T_{k-1} + T_k}$, а её начальное и конечное состояние совпадают соответственно с начальным и конечным состояниями трассы $\mathbf{T_{k-2}}$. Повторяя аналогичные рассуждения необходимое количество раз, мы доказываем даже более сильное утверждение относительно построенного алгоритмом разбиения:

Для $\mathbf{\forall i, j}$: $\mathbf{1 \leq i \leq j \leq N}$, следовательно, определена сумма трасс $\mathbf{T_i + \ldots + T_j}$.

Следовательно, Требование 1 также выполняется для построенного разбиения.

Теорема доказана.

Доказав существование таких разбиений, мы можем пользоваться методом локализации ошибок путём сокращенного проигрывания трасс, который опирается на них. Метод заключается в следующем.

1. Считываем из трассы теста, обнаруживающего ошибку, список пройденных состояний и переходов и выкидываем из него все элементы, следующие за состоянием, в котором была обнаружена ошибка.
2. Строим разбиение полученной трассы на набор подтрасс, удовлетворяющий вышеописанным свойствам.
3. Находим минимальный номер **k** такой, что трасса $\mathbf{\Sigma_k = T_1 + T_2 + \ldots + T_{k-1} + T_k}$ находит ошибку, а трасса $\mathbf{\Sigma_{k-1} = T_1 + T_2 + \ldots + T_{k-1}}$ — не находит. Таким образом, мы обнаруживаем, что ошибка впервые

начинает проявляться после добавления к проигрываемой части трассы теста простого цикла $\mathbf{T_k}$. Этот цикл естественным образом становится первым «подозреваемым» на внесение ошибки в состояние тестируемой системы.

Как мы видели в примере выше, такое разбиение не единственно. Более того, различные разбиения одной и той же трассы, удовлетворяющие приведённым выше требованиям, могут содержать даже разное количество элементов, как показано на Рис. 5.

Поскольку в общем случае мы не знаем заранее, где именно допущена ошибка в тестируемой реализации, мы тем более не можем также знать, какое из разбиений будет более полезным для локализации ошибки. Однако если выполняются гипотезы I и II, мы можем, тем не менее, ожидать, что разбиение, которое содержит большое количество коротких циклов, будет лучше разбиения, содержащего малое количество длинных.

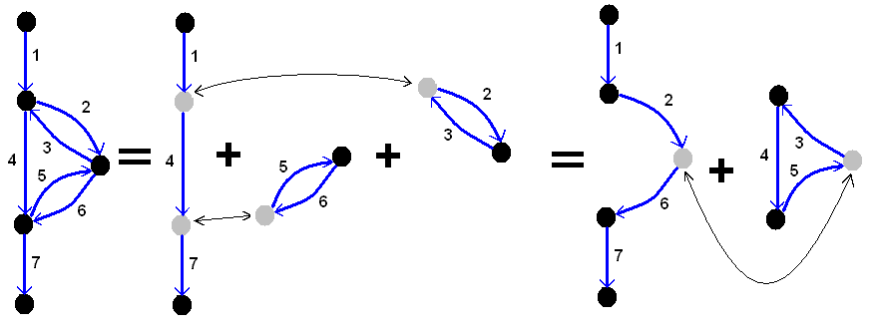

Рис. 5. Пример разбиений одной трассы с разным числом элементов.

На данный момент создана и апробирована реализация метода, использующая приведенный в Приложении 1 алгоритм и строящая с его помощью единственное разбиение.

5. Реализация

Создана реализация предложенного метода для технологии CTESK. Инструментарий CTESK [8,9] реализует технологию UniTESK для языка Си, поддерживая спецификационное расширение языка Си SeC [10].

5.1. Описание инструмента

Инструмент написан на языке Java и инсталлируется как дополнение к пакету программ UniTESK Trace Tools. Он принимает на вход набор файлов с трассами CTESK, анализирует их и для каждой трассы генерирует следующие выходные файлы.

- Результат анализа графа трассы. Этот файл содержит следующую информацию.
    - Все обнаруженные во входной трассе состояния тестового сценария.
    - Информацию о первой обнаруженной в трассе ошибке (сценарий UniTESK в некоторых случаях может продолжать работу после ошибок, но в рассматриваемом методе оставшаяся часть трассы считается недостоверной и игнорируется).
    - Сигнатуры встретившихся во входной трассе сценарных методов. Сигнатура состоит из имени сценарного метода, а также типов и имён используемых в нём итерационных переменных.
    - Разбиение входной трассы на подтрассы согласно приведённым выше требованиям к такому разбиению.
    - Некоторая дополнительная информация, полученная в результате анализа графа состояний модельного КА. Эта информация не используется инструментом, но может быть полезна для разработчика.
- Сценарий, позволяющий воспроизводить записанную в трассе последовательность вызовов сценарных методов целиком или частично для поиска ошибок. Поскольку язык SeC не предусматривает непосредственного вызова сценарных методов с указанием значений итерационных переменных, сценарий генерируется на языке Си и работает совместно с результатом трансляции сценария на SeC в язык Си транслятором CTESK.

Инструмент налагает на входные трассы следующие ограничения.

- Трасса должна быть сгенерирована с помощью CTESK.
- Трасса содержит запуск ровно одного тестового сценария (работа с множеством сценариев, работающих последовательно или параллельно, пока что не реализована).
- Запущенный в трассе сценарий имеет тип DFSM (такой сценарий имеет единственный активный поток управления [7], в котором происходит обход детерминированного КА [6]).
- Некоторые дополнительные ограничения на внутреннюю целостность трассы, гарантирующие корректность извлеченной из неё информации о структуре модельного КА и сигнатурах вызываемых методов.

При нарушении некоторых из этих условий инструмент выдает соответствующее сообщение об ошибке и прекращает обработку трассы, при нарушении других сгенерированный сценарий может не компилироваться или быть неработоспособным. Например, если подать на вход инструменту трассу,

сгенерированную с помощью JavaTESK, на выходе можно получить корректный анализ трассы (который может быть полезен разработчику) и не имеющий смысла сценарий на языке Си.

5.2. Сгенерированный сценарий

Пример сгенерированного инструментом сценария приведён с некоторыми сокращениями в Приложении 2.

Сценарий содержит следующие методы.

- `replay_check_state(const char *state)` — проверка текущего модельного состояния исходного сценария CTESK.
- `replay_call__initialize(int argc, char **argv)` — инициализация сценария CTESK.
- `replay_call__finalize()` — завершение сценария CTESK.
- `replay_call__<имя_метода>(…)` для каждого встретившегося во входной трассе сценарного метода — вызов соответствующего сценарного метода с заданными значениями итерационных переменных. Имена и типы параметров метода соответствуют именам и типам итерационных переменных исходного сценарного метода, с точностью до некоторых преобразований типов.
- `do_replay(int path_number)` — воспроизведение подтрассы $\Sigma_{\text{path\_number}}$
- `main_<имя_исходной_трассы>(int argc, char **argv)` — главный метод сгенерированного сценария.
- `parse_<тип>(char *val)` для каждого встреченного в исходной трассе пользовательского типа итерационной переменной — декларация внешней пользовательской функции, восстанавливающей значения данного пользовательского типа из используемого в файле трассы строкового представления.

Тела и сигнатуры сгенерированных методов согласованы с извлеченными из входной трассы сигнатурами вызванных сценарных методов, с реализуемой транслятором CTESK схемой генерации кода и с библиотекой поддержки тестовой системы времени выполнения TSBasis. Для реализации инструмента автор совместно с разработчиками инструментария CTESK доработал схему трансляции языка SeC: в неё была добавлена возможность вызова на уровне сгенерированного кода сценарных методов с точно указанными значениями итерационных переменных. Эти изменения вошли в очередную официальную версию CTESK.

Для использования сгенерированного сценария его можно собрать (например, в проекте Microsoft Visual Studio с подключенным модулем интеграции CTESK) вместе со сценарием, сгенерировавшим исходную трассу, и со всем, что необходимо для его исполнения (спецификации, медиаторы, тестируемая система, библиотека поддержки выполнения тестов TSBasis, внешние

модули). После этого можно (а иногда необходимо) доработать сгенерированный сценарий (такая доработка необходима, если в исходном сценарии использовались итерационные переменные сложных спецификационных типов; в противном случае, сценарий работоспособен сразу после создания), заменить в процедуре `main()` запуск исходного сценария CTESK на вызов процедуры `main_…()` сгенерированного сценария, откомпилировать и запустить проект.

5.3. Запуск сценария

При запуске сценарий ищет в параметрах командной строки параметр `-path` *<path_number>*. Если параметр найден и имеет положительное значение в диапазоне

`1 ≤ path_number ≤ MAX_PATH_NUMBER`, то воспроизводится подтрасса $\Sigma_{\text{path\_number}}$. Если параметр отсутствует, то воспроизводится подтрасса $\Sigma_{\text{DEFAULT\_PATH\_NUMBER}}$. При значении `path_number = 0` сценарий переходит в режим автоматической локализации ошибок методом инкрементального поиска — он последовательно перебирает все значения этого параметра от 1 до максимально возможного для построенного разбиения и останавливается после первой обнаруженной ошибки. При этом перед воспроизведением каждой подтрассы состояние тестовой системы инициализируется методом `init()` исходного сценария, а после воспроизведения сбрасывается методом `finish()`, если таковые в нём определены.

Остальные параметры командной строки обрабатываются тестовой системой и исходным тестовым сценарием.

В ходе выполнения сценарий генерирует трассу UniTESK, которую разработчик может впоследствии анализировать.

Перед воспроизведением подтрассы Σ_k сценарий сбрасывает в выходную трассу сообщение `Scenario replay: trying path` *<k>*.

При наличии в исходной трассе ошибок, в полученной при запуске сгенерированного сценария трассе обязательно встречается одно из следующих сообщений: `Scenario replay: repeatable failure` — если при воспроизведении подтрассы удалось воспроизвести ожидаемую ошибку; `Scenario replay: could not repeat failure` — если ожидаемую ошибку воспроизвести не удалось.

Также независимо от наличия ошибок в исходной трассе возможно появление сообщения `Scenario replay: unexpected failure` — в случае, если при работе сценария возникла неожиданная ошибка.

В режиме автоматической локализации ошибок в выходную трассу сценария дополнительно сбрасываются сообщения следующего вида: `Scenario replay: failure found at path` *<k>* — если ошибка впервые возникла при воспроизведении подтрассы Σ_k; `Scenario replay: could not repeat failure at any path` — если ошибку не удалось воспроизвести ни на одной из подтрасс Σ_k

Ошибкой считается возникновение любого из следующих событий.

1. Вызванный сценарный метод возвращает сообщение об ошибке (оно может быть вызвано сообщением об ошибке от тестового оракула или нарушением внутренних условий целостности тестового сценария). Такая ошибка считается ожидаемой, если она произошла в том же переходе, что и в исходной трассе; в противном случае, ошибка считается неожиданной. Какие-либо другие характеристики ошибки при этом не проверяются.
2. При воспроизведении трассы тестируемая система пришла в состояние, отличное от ожидаемого. Такая ошибка не может быть ожидаемой.

При возникновении любой ошибки, соответствующая информация сбрасывается в выходную трассу, а работа сценария завершается.

6. Апробация

Описываемый метод был опробован на примерах, поставляемых вместе с инструментами CTESK. Примеры состоят из реализации небольшой системы, формальных спецификаций, медиаторов [4] и тестовых сценариев. Для апробации в реализацию, спецификации и тестовые сценарии (для технологии UniTESK несущественно, где именно возникла ошибка) вносились ошибки следующих видов.

1. Неверный результат работы метода при определённой комбинации состояния и входных параметров.
2. Вызов одного целевого метода с определёнными параметрами «портит» состояние так, что последующий вызов другого метода работает с ошибкой.
3. Накапливающаяся ошибка, которая проявляется только после определённого числа вызовов целевого метода.

Все эти ошибки обнаруживались тестовыми сценариями CTESK, при этом генерировались соответствующие трассы тестов, которые потом обрабатывались инструментом и воспроизводились с использованием той же ошибочной реализации, а также с другими реализациями (без ошибок и с другими ошибками).

Как и следовало ожидать, наилучшие результаты достигнуты для ошибок первого вида: сгенерированный из трассы сценарий успешно находил эти ошибки на кратчайшем пути от начального состояния к ошибочному (Σ_1). Ошибки второго вида обнаруживались на любой подтрассе Σ_k, начиная с первой включающей вызов, вносящий ошибку (вызов, обнаруживающий ошибку, всегда входит в эти подтрассы, так как расположен в исходной трассе

позже). Даже для ошибок третьего вида иногда удавалось сократить трассу, выкинув из неё находящиеся в начале трассы циклы, не участвующие в накоплении ошибки; но в целом для таких ошибок предлагаемый метод малоэффективен, поскольку минимальный тест, находящий ошибку, должен содержать все те взаимодействия с целевой системой из исходного теста, в которых ошибка накапливалась, а они обычно более-менее равномерно распределены по трассе теста, то есть не выполняется гипотеза II.

Инструмент также испытывался на трассах тестов, полученных из проекта верификации ОС Linux OLVER [11] и ОС реального времени для встроенных систем «Багет». Для этих проектов в ИСП РАН были разработаны тесты, моделирующие разнообразные части ядра ОС и стандартных библиотек конечными автоматами с состояниями различной сложности: от вырожденных до имеющих весьма сложную структуру. Наиболее простым и эффективным было применение инструмента для подсистем, имеющих вырожденное состояние (например, семейство функций `…printf()`), но в целом для таких случаев ценность рассмотренного метода невелика, поскольку тестируемые функции не зависят от предыстории, и возникновение ошибки зависит только от входных параметров, которые можно извлекать из отчетов UniTESK напрямую.

В случае сложного внутреннего состояния системы (например, сценарии, работающие с подсистемами mutex, pthread) сгенерированные инструментом файлы анализа во многих случаях достаточно наглядно описывали внутреннюю структуру состояния, в котором была обнаружена ошибка, и способ его получения. При работе с этими подсистемами было обнаружено еще одно ограничение, налагаемое инструментом на тесты: эквивалентные состояния модельного КА, в том числе полученные из разных запусков, должны иметь одинаковое строковое представление. Например, если состояние моделируется неупорядоченным множеством, то сравнение двух состояний как объектов может сообщать об их равенстве, а сравнение их строковых представлений — о неравенстве, так как порядок вывода элементов множества стандартными библиотечными функциями не регламентирован. Так как инструменту доступно только строковое представление состояния из входной трассы, для его применения в таких случаях необходима соответствующая доработка исходного сценария (написание детерминированных преобразований в строковое представление) или сгенерированного сценария (например, с помощью пользовательских функций, восстанавливающих значения соответствующих типов из строкового представления). Однако даже несмотря на эти ограничения, в некоторых случаях удавалось быстро получить сокращенный тест, находящий ту же ошибку.

7. Заключение

В данной статье описан метод сокращения тестов, находящих ошибку, предназначенный для помощи разработчикам в анализе результатов

тестирования и локализации обнаруженных ошибок. Приведённый метод предназначен в первую очередь для тестирования систем, обладающих сложным и недоступным для теста напрямую внутренним состоянием. Метод сокращенного воспроизведения трасс позволяет автоматически обнаруживать тестовые воздействия, которые с большой вероятностью ответственны за возникновение ошибки в тестируемой системе, даже если проявления ошибки были зафиксированы позже.

Приведено математическое обоснование предложенного метода. Предложен алгоритм, реализующий его, и доказана корректность этого алгоритма.

Разработан и апробирован инструмент, реализующий предложенный метод для технологии CTESK [8], которая реализует унифицированную технологию тестирования UniTESK [1] для языка Си. Результаты апробации показывают хорошие перспективы применения данного метода в промышленных проектах.

В дальнейшем планируется развивать предложенный метод в следующих направлениях.

- Разработка аналогичных инструментов для других языков, поддерживаемых технологией UniTESK.
- Создание библиотечных компонентов, упрощающих работу с пользовательскими типами данных.
- Реализация возможности работы с несколькими последовательными сценариями в одной трассе.
- Исследование возможности работы с параллельно работающими в одной трассе сценариями.
- Исследование других стратегий нахождения минимальных подтрасс, обнаруживающих заданную ошибку.
- Исследование возможности обработки более чем одной ошибки исходной трассы.

Литература

[1] http://www.unitesk.com/ru/ — сайт, посвященный технологии тестирования UniTesK и поддерживающим ее инструментам.

[2] В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. Подход UniTesK к разработке тестов. Программирование, 29(6):25–43, 2003. http://www.ispras.ru/~kuliamin/docs/Unitesk-2003-ru.pdf

[3] I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. FM'99: Formal Methods. LNCS 1708:608–621, Springer-Verlag, 1999.

[4] I. Bourdonov, A. Kossatchev, V. Kuliamin and A. Petrenko. UniTesK Test Suite Architecture. Proceedings of FME 2002, LNCS 2391:77–88, Springer-Verlag, 2002. http://www.ispras.ru/~kuliamin/docs/FME-2002-en.pdf

[5] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Использование конечных автоматов для тестирования программ. Программирование, 26(2):61–73, 2000. http://www.ispras.ru/~kuliamin/docs/FSM-2000-ru.pdf

[6] И. Б. Бурдонов, А. С. Косачев, В. В. Кулямин. Неизбыточные алгоритмы обхода ориентированных графов: детерминированный случай. Программирование, 29(5):59–69, 2003. http://www.ispras.ru/~kuliamin/docs/Graphs-2003-ru.pdf
[7] С. Г. Грошев. Применение технологии UniTesK для тестирования систем с различной конфигурацией активных потоков управления. Труды Института Системного Программирования РАН, 9:67–81, 2006. http://www.citforum.ru/SE/testing/unitest_use/
[8] CTESK. Документация. http://www.unitesk.ru/content/category/7/18/51/
[9] CTESK. Automating Testing of C Applications. Official CTESK whitepaper. http://www.unitesk.ru/download/papers/ctesk/ctesk_wp.pdf
[10] CTESK 2.2: Описание языка SeC. http://www.unitesk.ru/download/papers/ctesk/CTesK2.2LanguageReference.rus.pdf
[11] OLVER — проект по верификации ОС Linux. http://linuxtesting.ru/project/olver

Приложение 1. Алгоритм разбиения трассы

В данном разделе приведёна реализация предложенного алгоритма на языке Java версии 1.5.

В модели данных определен класс `State`, моделирующий состояние КА, и класс `Arc`, моделирующий переход трассы. Класс `Arc` содержит поля `startState` и `endState` типа `State`, ссылающиеся соответственно на начальное и конечное состояние соответствующего перехода.

Входные данные:

- `List<Arc> fullPath` — исходная трасса (сокращенная форма записи, в которой фигурируют только переходы). Согласно определению трассы, для любых двух последовательных переходов трассы `A1` и `A2`, `A1.endState == A2.startState`

Выходные данные:

- `List<Arc[]> pathes` — искомое разбиение входной трассы на последователь-ность подтрасс.

Код реализации алгоритма приведен ниже.

```
import java.util.*;
...
List<Arc[]> pathes = new LinkedList<Arc[]>();
// Хранилище для результатов алгоритма
List<Arc> currentPath = new ArrayList<Arc>();
for (int i = 0; i < fullPath.size(); i++)
{      // Главный цикл алгоритма
   Arc currentArc = fullPath.get(i);
   currentPath.add(currentArc);
```



```
   for (int j = 0; j < currentPath.size(); j++)
{  // Поиск цикла
       if (currentArc.endState ==
       currentPath.get(j).startState)
       {   // Выделяем обнаруженный цикл
           List<Arc> newLoop = currentPath.subList(j,
           currentPath.size());
           // Добавляем его в начало списка pathes
           pathes.add(0, newLoop.toArray(new
           Arc[newLoop.size()]) );
           // Удаляем входящие в него переходы из currentPath
           newLoop.clear();
           break;
       }
   }
}
// Добавляем в список pathes оставшийся прямой путь из начального состояния в конечное
if (currentPath.size() > 0)
   pathes.add(0, currentPath.toArray(new
Arc[currentPath.size()]) );
```

Приложение 2. Пример использования инструмента

Дан следующий исходный сценарий CTESK, тестирующий реализацию банковского счета.

```
   AccountModel acct;

   static bool account_init (int argc, char **argv)
   {...}
   static Integer* account_state() {
     return create_Integer(acct.balance);
   }
   scenario bool deposit_scen() {
     if (acct.balance <= 5) {
       iterate (int i = 1; i <= 5; i++;) {
         deposit_spec(&acct, i);
       }
     }
   }
   scenario bool withdraw_scen() {
     iterate (int i = 1; i <= 5; i++;)
       withdraw_spec(&acct, i);
     return true;
   }
```

```
  scenario dfsm account_scenario = {
    .init     = account_init,
    .getState = (PtrGetState)account_state,
    .actions  = {
      deposit_scen,
      withdraw_scen,
      NULL
    }
  };
```

В реализацию внесена ошибка. Сценарий обнаруживает её и генерирует соответствующую трассу. Из этой трассы инструмент генерирует сценарий на языке Си следующего вида.

```
#define DEFAULT_PATH_NUMBER 1
#define MAX_PATH_NUMBER 66

#define GOOD_CALL(call) if (!(call)) goto
__unexpected_failure;
#define BAD_CALL(call) if (call) goto __no_expected_failure;
else goto __expected_failure;

static bool replay_check_state(const char *state) {...}
static bool replay_call__initialize(int argc, char **argv)
{...}
static bool replay_call__finalize() {...}
static bool replay_call__deposit_scen(int i) {...}
static bool replay_call__withdraw_scen(int i) {...}

static int do_replay(int path_number)
{
// Straight path
  // pseudo-call: replay_check_state("start")
  // pseudo-call: replay_call__initialize() // transition #0
if (path_number > 16) {
  GOOD_CALL(replay_check_state("0"))
  GOOD_CALL(replay_call__deposit_scen(1)) // transition #1
}
if (path_number > 61) {
  GOOD_CALL(replay_check_state("1"))
  GOOD_CALL(replay_call__deposit_scen(1)) // transition #2
}
if (path_number > 62) {
  ...
}
if (path_number > 63) {
  ...
}
```



```
if (path_number > 64) {
  GOOD_CALL(replay_check_state("4"))
  GOOD_CALL(replay_call__deposit_scen(1)) // transition #5
}
if (path_number > 65) {
  GOOD_CALL(replay_check_state("5"))
  GOOD_CALL(replay_call__deposit_scen(1)) // transition #6
  GOOD_CALL(replay_check_state("6"))
  GOOD_CALL(replay_call__withdraw_scen(1)) // transition #7}
}
if (path_number > 64) {
  GOOD_CALL(replay_check_state("5"))
  ...
}
if (path_number > 63) {
  ...
}

......

if (path_number > 2) {
  ...
}
if (path_number > 1) {
  GOOD_CALL(replay_check_state("1"))
  GOOD_CALL(replay_call__deposit_scen(1)) // transition #400
  GOOD_CALL(replay_check_state("2"))
  GOOD_CALL(replay_call__deposit_scen(1)) // transition #401
  GOOD_CALL(replay_check_state("3"))
  GOOD_CALL(replay_call__deposit_scen(1)) // transition #402
  GOOD_CALL(replay_check_state("4"))
  GOOD_CALL(replay_call__deposit_scen(5)) // transition #403
  GOOD_CALL(replay_check_state("9"))
  GOOD_CALL(replay_call__withdraw_scen(1)) // transition #404
  GOOD_CALL(replay_check_state("8"))
  GOOD_CALL(replay_call__withdraw_scen(1)) // transition #405
  GOOD_CALL(replay_check_state("7"))
  GOOD_CALL(replay_call__withdraw_scen(1)) // transition #406
  GOOD_CALL(replay_check_state("6"))
  GOOD_CALL(replay_call__withdraw_scen(5)) // transition #407
}
// Straight path
  GOOD_CALL(replay_check_state("1"))
  GOOD_CALL(replay_call__deposit_scen(1)) // transition #408
  GOOD_CALL(replay_check_state("2"))
  GOOD_CALL(replay_call__deposit_scen(1)) // transition #409
  GOOD_CALL(replay_check_state("3"))
  GOOD_CALL(replay_call__deposit_scen(1)) // transition #410
  GOOD_CALL(replay_check_state("4"))
```

```
  // Original error: info = 'Scenario function failed', kind =
'Scenario Function Failed'
  BAD_CALL(replay_call__deposit_scen(1)) // transition #411
/* The rest of replay is skipped */

  return 0;

__unexpected_failure:
  traceException("Scenario replay: unexpected failure");
  return 1;
__no_expected_failure:
  traceException("Scenario replay: could not repeat failure");
  return 2;
__expected_failure:
  traceUserInfo("Scenario replay: repeatable failure");
  return 3;
}

int main_account_scenario_2007_06_15_11_00_01(int argc, char
**argv)
{ ... }
```

Из приведённого отрывка кода мы можем, например, видеть, что переходы 408-411 принадлежат подтрассе $\mathbf{T_1}$, причем переход 411 вызывал в исходной трассе ошибку. Кроме того, подтрасса $\mathbf{T_{66}}$ состоит из переходов с индексами 6 и 7, начинается в состоянии, имеющем строковое представление «5» и заканчивается в нём же, вложена в $\mathbf{T_{65}}$ и не содержит других вложенных подтрасс; подтрассы $\mathbf{T_{65}}$, $\mathbf{T_{64}}$, $\mathbf{T_{63}}$ и $\mathbf{T_{62}}$ последовательно вложены друг в друга. Что касается подтрасс $\mathbf{T_{62}}$…$\mathbf{T_{18}}$, то они вложены напрямую в $\mathbf{T_{17}}$; такого рода ситуации возникают, когда несколько подряд циклов (не обязательно простых) начинаются в одном и том же состоянии.

Более подробный анализ входной трассы содержится в файле анализа, который включает описание каждой подтрассы $\mathbf{T_k}$ в виде списка входящих в неё состояний и переходов и некоторую дополнительную информацию о графе КА, извлеченном из исходной трассы.