Санкт-Петербургский государственный университет информационных технологий, механики и оптики
Кафедра «Компьютерные технологии»

А.А. Ахи

Автоматические методы модификации решений для тестирования проверяющих программ

Бакалаврская работа

Научный руководитель: А.С. Станкевич

Санкт-Петербург
2011

Введение

Часто в олимпиадных задачах по программированию и информатике правильный результат работы программы является неоднозначным или имеет очень сложную структуру. В таких случаях членам жюри олимпиады для каждой задачи приходится писать проверяющую программу, которая сопоставляет выходные данные проверяемой программы с выходными данными теста (набор входных и выходных данных). Набор тестов создается жюри. Естественно, в такой программе могут быть допущены ошибки. Результаты соревнований становятся менее объективными, возрастает доля везения, участники соревнования начинают проникаться недоверием к представителям жюри. Нахождение ошибок в процессе соревнования может привести к продлению соревнования или перетестированию всех решений участников.

Проблема проверки проверяющих программ уходит корнями в философский вопрос «*Quis custodiet ipsos custodes?*», который можно перевести, как «Кто будет охранять охранников?» Обычно, в корректности тестов и проверяющих программ убеждаются, с помощью их беглого, а в некоторых случаях даже детального, просмотра. Однако, не все ошибки, которые могут быть допущены в проверяющих программах, являются простыми для нахождения. Некоторые из них проявляются на очень сложных по структуре тестах, которые сложно придумать человеку.

В настоящей работе разработан автоматический метод выявления ошибок в проверяющих программах. Идея метода состоит в автоматической генерации неверных решений из правильного решения задачи и в последующем тестировании полученных решений. При тестировании проводится анализ покрытия кода и результата работы проверяющей программы. Ошибки, вносимые в решения, симулируют распространенные ошибки, допускаемые участниками соревнований. Как показывает практика, даже самые незначительные ошибки могут приводить к весьма неожиданным результатам. Именно это наблюдение позволяется предполагать, что при тестировании таких измененных решений,

множество создаваемых ими выходных файлов будет большим и весьма разнообразным.

Для анализа работы проверяющей программы в настоящей работе использовалось покрытие ее кода. Считается, что в хорошей проверяющей программе все строки кода должны исполняться при тестировании различных правильных и неправильных решений. Некоторые ветви кода проверяющей программы отвечают проверке корректности ответа жюри или сравнения его с ответом участника. Для покрытия этих ветвей проверяющая программа запускается не только в обычном режиме, но и с поменянными местами ответами жюри и участника. Также производятся запуски проверяющей программы на выходных файлах, полученных правильным решением. Наличие неполного покрытия кода может указывать на целый ряд причин. Наиболее очевидными из них являются отсутствие необходимости в непокрытых частях исходного кода, неправильность проверяющей программы, неполнота имеющегося набора тестов и невезение.

Также происходит анализ результата тестирования. Ответ модифицированного решения не должен быть лучше ответа жюри, если возможно сравнение ответов в рамках поставленной задачи. Ответ жюри всегда должен быть корректным с точки зрения формата выходного файла, описанного в условии задачи. Такой анализ позволяет сразу выявить неправильное поведение проверяющей программы и даже предоставить набор входных и выходных файлов, на которых достигается ошибочный результат проверки.

Описанный в настоящей работе метод позволяет выявлять ошибки и ненужные части в проверяющей программе, проверять корректность и полноту предоставленных тестов, а также корректность решения жюри. Представленный метод не доказывает и не показывает правильность проверяющей программы, он способен лишь указать на наличие ошибок или же на странности поведения. Доказать правильность программы с помощью тестов не возможно, что подтверждает высказывание Эдсгера Дейкстры «Тестирование программы

может весьма эффективно продемонстрировать наличие ошибок, но безнадежно неадекватно для демонстрации их отсутствия» [1].

Глава 1. Основные понятия и обзор существующих методов

1.1. Основные понятия

Будем называть *решением* программу, которая должна по входному файлу выдавать правильный выходной файл. Заметим, что решение может являться неправильным. Это означает, что выдаваемый ей выходной файл не удовлетворяет условиям задачи.

Под *решением жюри* будем понимать решение, которое жюри использует для генерации выходных файлов по входным файлам из подготовленных жюри тестов. Решение жюри также может оказать неверным.

Будем называть *ответом жюри* выходной файл, который генерирует решение жюри. Ответ жюри должен быть правильным и корректным с точки зрения поставленной задачи, однако, он может не являться таковым вследствие ошибок в решение жюри.

Решение участника — решение, тестирование которого производится. Аналогично *ответ участника* — генерируемый решением участника выходной файл.

Проверяющая программа — программа, получающая входной файл, ответы жюри и участника и проверяющая правильность ответа участника.

Проверяющая программа может иметь несколько различных результатов проверки:

- *OK* — ответ участника признается верным;
- *WA (Wrong Answer)* — ответ участника признается неверным;
- *PE (Presentation Error)* — ответ участника признается не корректным из-за несоответствия формату выходного файла;
- *FAIL* — ответ участника оказался лучше ответа жюри в смысле условия задачи, ответ жюри оказался некорректным, входной файл

оказался некорректным, проверяющей программе не удалось завершить свою работу из-за ошибки.

1.2. Покрытие кода (*code coverage*)

1.2.1. Использование покрытия кода

Первое упоминание термина «*покрытие кода*» относится к 1963 году [2]. Покрытие кода является мерой, используемой при тестировании программного обеспечения. Эта мера показывает, насколько исходный код программы был протестирован на наборе тестов. Считается, что набор тестов должен давать полное покрытие кода, иначе какие-то части программы останутся не протестированными.

Существует несколько критериев покрытия. Ключевыми из них являются:

- *Покрытие функций* — верно ли, что функция или процедура хоть раз была вызвана?
- *Покрытие строк* — верно ли, что каждая строка исходного кода была хоть раз выполнена?
- *Покрытие решений* — верно ли, что каждое условие в циклах и операторах ветвления выполнилось всеми возможными способами, например, выполнились обе ветви условного оператора *IF*?
- *Покрытие условий* — верно ли, что каждое условие в программе приняло оба возможных значения?
- *Покрытие путей* — верно ли, что были пройдены все возможные пути исполнения программы?

Проверка покрытия путей обычно не является возможной, так как их число экспоненциально зависит от числа условий в исходном коде. Применение остальных метрик не представляет особых сложностей.

Обычное использование покрытия кода заключается в обнаружении непокрытых в результате тестирования областей кода и последующем составлении тестов для покрытия непокрытых областей. Таким образом, повторяя про-

цесс, можно получить набор тестов, который полностью тестирует программу. Если же какие-то части исходного кода покрыть не удается, то, возможно, программа составлена неверно.

1.2.2. Существующие продукты для анализа покрытия кода

Существуют различные продукты, анализирующих покрытие кода программ на различных языках. Для языка *Java* распространенными открытыми продуктами по анализу покрытия кода являются *Clover*, *EMMA* и *Serenity* [3 - 5]. Эти программы анализируют не покрытие исходного кода, а покрытие байт-кода, получающегося в результате компиляции программ на языке *Java*. Байт-код модифицируется в процессе первой загрузки кода класса. Для анализа покрытия используются метрики покрытия функций, строк и решений.

1.3. Мутационное тестирование (*mutation testing*)

1.3.1. Использование мутационного тестирования

Метод мутационного тестирования зародился в 70-х годах прошлого века и получил первое практическое применение в 80-х. Мутационное тестирование — метод тестирования программного обеспечения, заключающийся во внесении небольших изменений в исходный код программы. Отсутствие ошибок и неверных результатов при тестировании измененной программы на наборе тестов может означать низкое качество предоставленного набора тестов или же низкую значимость измененной части кода. Данный метод направлен на построение хорошего набора тестов, а также выявление частей кода, которые никогда или очень редко достигаются в процессе исполнения программы.

Первоначально метод был предложен для выявления слабости набора тестов. Тесты призваны показать правильность (или неправильность) программы, но кто покажет правильность тестов? Таким образом, происходит отсылка к более глубокой философской проблеме «*Quis custodiet ipsos custodes?*», которую можно перевести, как «Кто охраняет охранников?»

Для применения мутационного тестирования выбирается набор мутационных операторов, которые по одному применяются к исходному коду. Мута-

ционные операторы обычно включают в себя удаление строки кода, замену одного оператора другим, замену переменной на другую переменную того же типа, а также другие характерные для программистов ошибки. Некоторые из мутационных операторов являются уникальными для языка программирования, так как используют его особенности.

Мутационное тестирование разделяют на два класса: слабое мутационное тестирование и сильное мутационное тестирование. Основной целью слабого мутационного тестирования является проверка покрытия у первоначального и измененного исходных кодов. При этом считается, что тест пройден, если результирующие покрытия не различаются. В отличие от слабого, сильное мутационное тестирование также требует того, чтобы значения все переменных во время исполнения совпадали.

В данной работе применяются идеи мутационного тестирования. При этом выполняется модификация исходного кода решения с помощью мутационных операторов для генерации различных вариантов программы с ошибками.

1.3.2. Пример продукта для мутационного тестирования

Одним из открытых продуктов, предоставляющих возможности мутационного тестирования программ на языке *Java*, является программа *MuJava* [6]. Эта программа производит компиляцию исходного кода в байт-код, после этого к полученному байт-коду применяются различные мутационные операторы. Программа использует более двадцати различных мутационных операторов, условно разбитых на две группы: структурные и поведенческие. Часть используемых операторов не зависит от языка программирования. Они были описаны в работе [7], однако, в программе *MuJava* были также реализованы операторы, использующие особенности языка *Java*.

В данной работе применяется только часть из описанных в работе [7] мутационных операторов. Эти операторы в настоящей работе используются иначе, чем в работе [7].

Глава 2. Разработка средств анализа покрытия кода проверяющей программы и мутации решений

2.1. Анализ покрытия кода проверяющей программы

2.1.1. Метрика покрытия

В данной работе используются две метрики покрытия кода: покрытие строк и покрытие решений. Также регистрируются факты создания объектов и статической инициализации классов. Применение таких метрик позволяет собрать статистику по частоте использования каждой отдельной строки исходного кода программы, а также убедиться в отсутствии «бесполезных» условий в проверяющей программе — условий, которые всегда выполняются или всегда не выполняются, и всегда одинаково влияют на исполнение программы.

Отсутствие покрытия не гарантирует неправильность проверяющей программы, однако, оно указывает на подозрительные места в исходном коде, на которые стоило бы обратить особое внимание. Никакая метрика не способна показать правильность или неправильность исходного кода проверяющей программы. Выбор метрик в данной работе обусловлен относительной простотой их реализации. В случае отсутствия полного покрытия кода любой из метрик требуется вмешательство человека для оценки правильности исходного кода проверяющей программы и полноты имеющихся тестов.

2.1.2. Способ сбора статистических данных и модификация исходного кода проверяющей программы

Для сбора информации о покрытии исходного кода проверяющей программы была разработана программа для ее модификации. В исходный код добавляется сбор статистических данных перед выполнением каждой команды. Это позволяется получать информацию о покрытии строк. Сбор информации выглядит, как увеличение счетчика в массиве покрытия.

Для сбора информации о покрытии решений все циклы и операторы ветвления были модифицированы. Рассмотрим модификацию цикла *while* с условием *cond* (табл. 1). Условие в цикле было заменено на значение *true*, что де-

лает цикл бесконечным. Однако, для получения информации о выполнимости условия и выходе из цикла при невыполнении условия в начало тела цикла вставляется оператор ветвления *if* с первоначальным условием цикла, в *else* ветви которого происходит выход из цикла. Могут возникнуть проблемы, если условием цикла было *true*. Так, если цикл был внутри функции, то компилятор потребует добавления операции возврата значения после цикла. Для этих целей случай цикла *while (true)* рассматривается отдельно и оператор выхода из цикла *break* не добавляется. Очевидно, что покрытие такого условия будет неполным. Анализ таких строк должен осуществляться человеком отдельно от работы программы.

Таблица 1. Пример модификации цикла

Цикл до модификации	Цикл после модификации

`while (cond) {`
`  ... // тело цикла`
`}`
	`while (true) {`
	`  if (cond) {`
	`  ... // помечаем, что условие выполнилось`
	`  } else {`
	`  ... // помечаем, что условие не выполнилось`
	`  break;`
	`  }`
	`  ... // тело цикла`
	`}`

Модификация оператора ветвления *if* происходит другим образом (табл. 2). Прежде всего, в случае отсутствия у конструкции *else*-части, происходит ее добавление. Далее, если какая-либо часть конструкции не является *блоком* (набор операций в фигурных скобках), то происходит ее превращение в блок. В начало каждого блока добавляются команды сбора статистической ин-

формации, соответствующие отметке выполнения и невыполнения условия ветвления.

Таблица 2. Пример модификации оператора *if*

Оператор *if* до модификации	Оператор *if* после модификации

`if (cond)`
`  break;`
	`if (cond) {`
	`  ... // помечаем, что условие выполнилось`
	`  break;`
	`} else {`
	`  ... // помечаем, что условие не выполнилось`
	`}`

2.2. Мутация правильного решения

2.2.1. Мутационные операторы

Для мутации правильных решений использовалось несколько десятков различных мутационных операторов. Операторы были разделены на классы, соответствующие конструкциям языка программирования *Java*. При этом операторы более сложных конструкций могли использовать операторы других конструкций. Такой способ классификации операторов привел к появлению одного оператора мутации всего решения, который и применяется для создания модифицированных решений.

Полученный оператор подсчитывает число единичных мутаций, которые можно применить к исходному коду, и обладает функцией выдачи кода с одной мутацией-ошибкой по номеру ошибки. Это позволяет организовывать как случайный перебор мутированных решений, так и, например, последовательный перебор всех единичных мутаций.

Мутации имитируют распространенные ошибки, встречающиеся в решениях участников. Даже небольшие ошибки могут существенно влиять на ра-

боту программы, что приводит к разнообразным выходным данным решений. Наличие большого числа выходных файлов позволяет надеяться на достижение полного покрытия исходного кода проверяющей программы, а также качественное ее тестирование.

Наиболее сложной стороной применения мутаций к исходному коду программы является необходимость поддерживать множество доступных в этой области кода переменных и совместимость типов. При неправильной реализации этих случаев могут получаться некомпилирующиеся модификации решений, а некоторые очевидные модификации, напротив, получены не будут. В табл. 3 рассмотрены примеры основных реализованных мутаций.

Таблица 3. Основные мутационные операторы

Структура языка	Ошибки в данной структуре	Пример исходного кода	Пример модифицированного кода
Составной оператор присваивания (*AssignOp*)	Замена одно составного оператора присваивания другим	`a += b;`	`a ^= b;`
Двухместный оператор (*Binary*)	Замена одного оператора другим	`a + b`	`a - b`
	В случае оператора равенства (неравенства) замена на вызов метода equals	`s1 == s2`	`s1.equals(s2)`
Вызов метода *equals*	Замена на сравнение по ссылки	`s1.equals(s2)`	`s1 == s2`
Операторы управления циклами	Удаление, добавление или замена метки цикла, а	`break loop1;`	`break loop2;`

(*Break, Continue*)	также замена на другой оператор		
Условный оператор (*Conditional*)	Замена оператора на одну из ветвей	`a > b ? c : d`	`c`
Константа или переменная	Замена на константу или переменную того же типа	`a[0] = 5;`	`a[i] = 5;`
Числовое выражение	Замена на константу или переменную, прибавление небольшого числа	`x = y;`	`x = y - 2;`
Булево выражение	Замена на константу или отрицание выражения	`x > y`	`!(x > y)`

Оператор ветвления (*If*)	Оставить только код соответствующий одной из ветвей	`if (x > y) {`
		`  x = y`
		`} else {`
		`  y = x`
		`}`
			`y = x`

Цикл (*while, do-while, for*)	Изменение условия в цикле	`while (x < 2)`
		`{`
		`  x++;`
		`}`
			`while (x < 3)`
			`{`
			`  x++;`
			`}`

	Изменение тела цикла	`while (x < 2)`
		`{`
		`  x++;`
		`}`
			`while (x < 2)`
			`{`
			`  y++;`
			`}`

Помимо мутаций, указанных в табл. 3, также использовались следующие мутации: удаление блока кода или обмен двух соседних блоков местами. Под блоком кода понимается строка кода или сегмент кода, отвечающий определенной конструкции языка, например, конструкция *if-else* или же несколько строк, заключенные в фигурные скобки.

Также любое выражение, тип которого удалось выяснить с помощью простейшего анализа, может быть заменено на константу того же типа. Список констант можно наблюдать в табл. 4. Список констант не является фиксированным и может изменяться в конфигурационном файле для оптимизации под конкретную задачу, например, для задач, созданных в Польше, в которых вместо «Yes» или «No» необходимо выводить «TAK» или «NIE». В случае, когда тип выражения установить не удалось, используются все возможные варианты.

Таблица 4. Константы, используемые для замены выражений

Тип	Константы
`Boolean`	`false` `true`

`byte, short, int, long`	`-1`
	`0`
	`1`

`float, double`	`-1`
	`0`
	`0.5`
	`1`

`char`	`'a'`
	`'z'`
	`'A'`
	`'Z'`
	`'0'`
	`'9'`
	`' '`

String	“” “ ” “Yes” “No” “Impossible” “ 0” “ 1”

В случае модификации числового или строкового выражения возможны дополнительные мутации, например добавление некоторого числа/строки к выражению. Так для чисел наиболее характерными являются добавление -1 или 1. Для строковых выражений можно произвести прибавление одной из строк, используемых для замены.

Строки являются гораздо более сложными объектами, нежели примитивные типы, что приводит к наличию большего числа мутаций строк и строковых выражений. У любого строкового выражения могут быть вызваны операторы *toLowerCase()* и *toUpperCase()*. Для каждой строки, присутствующей в исходном коде программы как константа, возможен целый ряд изменений: удаление одного символа из строки, замена символа.

Наличие большого числа мутационных операторов приводит к большому числу возможных мутированных решений. Таким образом, при расширении набора операторов, происходит увеличение времени работы, необходимого для тестирования проверяющей программы. Метод регулирования константных выражений позволяет управлять числом мутаций и временем работы.

2.2.2. Применение мутационных операторов

Применение мутационных операторов состоит в изменении исходного кода правильного решения. Для удобства работы с исходным кодом производится построение его дерева разбора. Для каждого вида узла в дереве реализован класс, содержащий две функции: узнать число единичных мутаций части кода и вернуть измененный код по номеру единичной мутации.

Нахождение числа возможных мутаций происходит с помощью разбиения кода на более простые части и суммирования числа возможных модификаций в каждой из частей. Также к ответу прибавляется число изменений, которое можно совершить с рассматриваемой структурой языка, исходя из используемых мутаций.

Получение мутированного кода по номеру происходит очевидным образом: исходный код, к которому требуется применить мутацию, разбивается на составные части, для каждой из которых производится расчет числа мутаций. С помощью этих данных вычисляется, в какой из частей следует сделать модификацию, или же изменению следует подвергнуть рассматриваемую структуру языка программирования.

2.2.3. Анализ примитивных типов выражений

Некоторые мутации, например, замена одной переменной на другую, требуют знания о классах и всех видимых переменных. Для эффективного использования таких мутаций был реализован класс *VariableEnvironment*, содержащий список доступных переменных и их типы. Данный класс позволяет узнавать тип переменной, получать все переменные определенного типа.

Первоначальные испытания продукта для тестирования проверяющих программ показали, что очень велика доля некомпилирующихся решений, получаемых в результате единичных мутаций правильных решений, а доля компилирующихся программ составляла всего 12% от общего числа единичных мутаций. Для повышения процента компилирующихся решений в классе был реализован механизм простейшего анализа примитивных типов выражений.

Производится анализ только примитивных типов и типа *String*, так как в исходном коде программы могут встречаться константы и константные выражения только этих типов и операции, отличные от присвоения, обращения к элементу массива и вызова метода, могут быть применены только к выражениям таких типов.

Анализ типов выражений производится в двух направлениях: сверху вниз и снизу вверх. Анализ снизу вверх позволяет агрегировать тип выражения

из его составных частей, например, зная типы слагаемых, несложно вычислить тип суммы. В случае арифметических или битовых операций требуется просто выбрать наиболее общий тип. Однако, из-за отсутствия поддержки типов функций классом *VariableEnvironment*, не всегда представляется возможным вычислить типы частей выражения. Тогда в случае арифметических операций, можно предполагать тип выражения не ниже известных – всегда выбирать самый общий из известных типов. При рассмотрении операций сравнения сразу становится ясно, что выражение имеет тип *boolean*.

Анализ сверху вниз сообщает о возможных типах выражения. Эти данные используются при попытке замены выражения на константу или переменную. Изначально предполагается, что выражение может иметь любой тип. Однако в некоторых случаях на типы начинают накладываться ограничения. Виды ограничений приведены в табл. 5.

Таблица 5. Ограничения типов выражений

Выражение	Пример	Возможные типы частей
Присваивание	`a = expr`	Тип переменной *a* является обобщением типа выражения *expr*
Сравнение	`expr1 < expr2`	*expr1* и *expr2* имеют примитивный тип, но не *boolean*
Равенство	`expr1 != expr2`	*expr1* и *expr2* могут иметь любой тип
Сложение	`expr1 + expr2`	*expr1* и *expr2* имеют примитивный тип, но не *boolean*. Если про выражение известно, что оно может являться строкой, то *expr1* и *expr2* могут иметь любой тип
Прочие арифмети-	`expr1 - expr2`	*expr1* и *expr2* имеют примитив-

ческие операции		ный тип, но не *boolean*
Битовые операции	`expr1 | expr2`	*expr1* и *expr2* имеют целочисленные примитивные типы, *char* или *boolean*
Логические операции	`expr1 || expr2`	*expr1* и *expr2* имеют тип *boolean*
Обращение к массиву	`a[expr]`	*expr* имеет тип входящий в тип *int*
Вызов метода	`expr.method(args)`	*expr* не может иметь примитивный тип
Условие в цикле или *If*	`if (expr) { }`	*expr* имеет тип *Boolean*

Сверх указанных в табл. 5 ограничений, в случае битовых и арифметических операций используются дополнительные ограничения, что тип каждой из частей должен умещаться в тип всего выражения.

Все указанные ограничения на типы позволяют повысить процент компилирующихся мутаций с 12% до 60%. Очевидно, что применяя более тонкий анализ типов выражений, можно совсем избавиться от мутаций, приводящих в бесполезным, некомпилирующимся решениям. Однако, такой анализ довольно сложен, именно по этой причине он и не был произведен в данной работе.

Глава 3. Применение мутации решений и анализа покрытия кода для оценки проверяющих программ и тестов

В данной главе описан способ применения мутации решений и анализа покрытия кода для тестирования проверяющих программ задач соревнований по программированию и информатике. Схему работы предложенного метода можно наблюдать на рис. 1. Метод заключается в генерации измененных решений с помощью применения операторов мутации к решению жюри, запуске полученных решений на имеющемся наборе тестов и последующим исполнении проверяющей программы на выходных файлах, полученных от запуска модифицированных решений. В процессе выполнения проверяющей программы происходит сбор статистических данных для последующего анализа покрытия исходного кода. Также анализу подвергаются результаты работы проверяющей программы. Такой механизм позволяет выявлять ряд ошибок в проверяющих программах. Данная глава содержит описание процесса запуска измененных решений и проверяющей программы и анализа собранных данных с целью выявления ошибок и странностей в поведении проверяющей программы.

Рис. 1.Схема работы предложенного метода

3.1. Тестирование мутированных решений

3.1.1. Запуск модифицированных решений

Перед тем как запустить модифицированное решение, его, прежде всего, необходимо скомпилировать. Компиляция осуществляется с помощью компилятора языка *Java*, входящего в поставку *Java SE Development Kit* [8].

Так как классы всех модифицированных решений имеют одинаковые имена, то возникает проблема с загрузкой таких классов в виртуальную машину. Все дело в том, что виртуальная машина загружает каждый класс всего один раз. Для того чтобы перезагружать классы решений каждый раз, был реализован отдельный загрузчик классов класс *MyClassReloader*. Этот загрузчик перезагружает все классы, содержащие имя задачи как подстроку, и действует обычным образом со всеми остальными классами.

Запуск модифицированных решений представляется отдельной проблемой. Запущенное измененное решение может не только корректно завершать свою работу, но и завершаться с ошибкой, работать очень долго или вообще никогда не завершаться. Именно поэтому запуск модифицированных решений происходит в отдельном потоке, что позволяет контролировать появление ошибок времени исполнения, а также следить за временем выполнения отдельного решения на отдельном тесте.

Проверка модифицированного решения производится на предоставленном наборе тестов. Для каждого решения производится последовательный запуск его на каждом из тестов. Если решение работает достаточно долго на одном из тестов, то принимается решение о прекращении тестирования данного решения, так как велика вероятность появления «бесконечного» цикла в коде программы, что заставит систему выжидать установленное время на каждом тесте и замедлит процесс тестирования. На последующих тестах решение также не тестируется.

Одним из основных вопросов является вопрос о том, какие именно модифицированные решения тестировать. Очевидно, что множество программ, которые можно получить с помощью перечисленных модификаций, очень ве-

лико. В данной работе было принято проверять все решения, отличающиеся от правильного единичной мутацией. Таким образом, в каждом месте исходного кода программы будет допущена какая-нибудь ошибка.

На каждом предоставленном тесте после запуска модифицированного решения, в случае его удачного и своевременного завершения, производится запуск проверяющей программы. В качестве входных данных проверяющая программа получает входной файл теста, выходной файл, полученный запуском решения жюри, и выходной файл, полученный в результате запуска модифицированного решения.

Если проверяющая программа проверяет корректность ответа жюри или же ответ участника сопровождается сертификатом и может оказаться лучше ответа жюри и проверяющая программа это проверяет, то такие ветки проверяющей программы никогда покрыты не будут. С целью покрытия ветвей, ведущих к результату исполнения проверяющей программы *FAIL*, производится запуск проверяющей программы с поменянными местами выходными файлами. Как будет показано далее, такой запуск позволяет не только добиваться более полного покрытия исходного кода проверяющей программы, но и делать дополнительные выводы о правильности компонент задачи, анализируя результат, выдаваемый проверяющей программой в случае такого запуска.

Запуск проверяющей программы также осуществляется в отдельном потоке. Это связано с возможностью чрезвычайно долгой работы проверяющей программы на некоторых входных данных, что свидетельствует о наличии ошибок.

3.1.2. Анализ покрытия кода проверяющей программы

Анализ покрытия исходного кода проверяющей программы, после тестирования модифицированных решений, позволяет делать некоторые выводы о качестве и правильности данной проверяющей программы. Прежде всего, достижение полного покрытия по всем метрикам, свидетельствует о высоком качестве проверяющей программы и предоставленного набора тестов, так как были разобраны все ветви, отвечающие всем крайним случаям.

В данной работе для анализа покрытия рассматриваются два критерия покрытия кода. Первым из них является покрытие строк. Если в результате запуска проверяющей программы на выходных файлах модифицированного решения какие-либо строки исходного кода проверяющей программы не были покрыты, то это может указывать на целый ряд неполадок и неисправностей. Непокрытые строки могут оказаться избыточными в проверяющей программе. Возможно, указанные строки не были покрыты из-за неправильности структуры и условий в проверяющей программе. Еще одной возможной причиной отсутствия покрытия может являться неполнота предоставленного набора тестов, не покрывающего все крайние случаи, возможные в задаче.

Схожие выводы можно делать и относительно отсутствия покрытия решений. Отсутствие покрытия решений позволяет более глубоко понять проблему, а также отследить наличие всегда выполняющихся условий, которые, тем не менее, оставляют покрытия строк полным.

Стоит заметить, что полное покрытие исходного кода проверяющей программы не может гарантировать ее правильность. Например, проверяющая программа, содержащая только операцию возврата результата *OK*, будет иметь полное покрытие, но она не является верной для практически любой задачи. Никакое тестирование никогда не сможет показать правильность программы, но, наоборот, может указать на неправильность. Как сказал Эдсгер Дейкстра, — «Тестирование программы может весьма эффективно продемонстрировать наличие ошибок, но безнадежно неадекватно для демонстрации их отсутствия».

3.1.3. Анализ результаты работа проверяющей программы

Сделать выводы об ошибочности той или иной части задачи можно, анализируя не только покрытие исходного кода проверяющей программы, но и изучая результаты ее запуска на полученных выходных файлах.

Прежде всего, в результате исполнения проверяющей программы может произойти ошибка, что точно указывает на ее неправильность. Помимо ошибки времени исполнения, проверяющая программа может работать недопустимо долго. Такой случай также будет зафиксирован тестирующей системой.

Если же работа проверяющей программы успешно завершилась, то анализу может быть подвергнут выдаваемый ею результат. При нормальной работе проверяющей программы никак нельзя ожидать результата *FAIL*. Такой результат обычно имеет одно из следующих значений: исполнение проверяющей программы завершилось с ошибкой, «правильный» ответ жюри не является корректным с точки зрения поставленной задачи, ответ участника является корректным и лучшим, чем ответ жюри относительно условия данной задачи. Таким образом, в случае получения результата *FAIL* появляется определенная область для поиска неисправности. Более того, система предоставит входной файл, ответ жюри и ответ участника, на которых достигается неверное поведение проверяющей программы.

При запуске проверяющей программы, с ответом участника, полностью совпадающим с ответом жюри, можно ожидать только один результат исполнения проверяющей программы *OK*. Любой другой результат, очевидно, является ошибочным. Причина ошибки может заключаться в неправильности проверяющей программы или же решения жюри.

Результаты тестирования *WA* и *PE* не представляют собой большого интереса, так как модифицированные решения могут выдавать выходные файлы с практически случайным содержимым, являющимся неправильным или же не имеющим смысла в условиях поставленной задачи.

Отдельно стоит рассмотреть результат в случае «обратного» запуска проверяющей программы — запуска с поменянными места выходными файлами жюри и участника. Однако, стоит заметить, что глубокий анализ результатов такого запуска не всегда возможен. Сделать какие-либо выводы можно в случаях, когда проверяющая программа проверяет корректность решения жюри или участника и сравнивает, какое из них лучше. В таком случае от проверяющей программы нельзя ожидать результатов проверки отличных от *OK* и *FAIL*.

Более того, в режиме «обратной» проверки, тестируемая проверяющая программа не должна никогда выдавать в качестве результата проверки *PE*, так

как выходной файл, получаемый при запуске решения жюри, обязан быть корректным с точки зрения условия задачи.

Как можно понять из сказанного выше, тестирование проверяющих программ, просто сравнивающих выходные файлы, не будет являться эффективным, так как такие проверяющие программы не несут в себе какой-либо логики. Проверяющие программы такого рода должны входить в специальную библиотеку проверяющих программ, откуда они могут быть доступны составителям задач, чтобы не возникало необходимости написания таких программ заново.

Удобство разработанной системы состоит в том, что в случае обнаружения неправильного или некорректного поведения проверяющей программы, пользователю будут сообщены входной файл и выходные файлы жюри и участника, при запуске с которыми наблюдалось неправильное поведение. Это предоставляет возможность для дальнейшей отладки проверяющей программы.

3.2. Результаты тестирования задач

3.2.1. Результаты тестирования

Разработанная программа была запущенна для тестирования проверяющих программ, использованных на полуфинале чемпионата мира по программированию *NEERC2010* [9]. Данное соревнование крайне удобно для тестирования, так как каждая задача сопровождается набором тестов, решениями и проверяющей программой на языке *Java*. Результаты тестирования приведены в табл. 6.

Таблица 6. Результаты тестирования проверяющих программ соревнования *NEERC2010*. В последнем столбце без скобок обозначена метрика покрытия строк, в скобках — метрика покрытия решений

Задача	Число строк в решении	Число единичных модификаций	Число компилирующихся решений	Число правильных решений	Время работы (мин.)	Число непокрытых строк проверяющей программы
alignment	79	1994	1143 (57%)	335 (17%)	9	0 (0) из 12 (4)
binary	208	5597	3349 (60%)	629 (11%)	8	0 (0) из 7 (2)
cactus	280	7811	4738 (61%)	1340 (17%)	500	0 (0) из 58 (19)
dome	191	3482	2237 (64%)	513 (15%)	175	0 (0) из 25 (7)
evacuation	133	6446	4009 (62%)	1750 (27%)	700	0 (0) из 36 (12)
factorial	156	5770	3623 (63%)	331 (6%)	700	0 (0) из 14 (4)
hands	308	7919	4200 (53%)	757 (10%)	245	0 (0) из 7 (2)
ideal	201	6502	3919 (60%)	2214 (34%)	900	0 (0) из 10 (3)
jungle	161	2991	1610 (54%)	357 (12%)	100	0 (0) из 7 (2)
kgraph	206	2941	1840 (63%)	556 (19%)	300	0 (0) из 27 (7)

Также разработанный продукт использовался при подготовке соревнования *Russian Code Cup 2011* [10]. В процессе тестирования проверяющих программ этого соревнования выявлялись многочисленные ошибки проверяющих программ. Ошибки были устранены. Заметим, что наличие ошибок серьезно бы повлияло на ход соревнования. Результаты тестирования проверяющих программ приведены в табл. 7.

Таблица 7. Результаты тестирования проверяющих программ соревнования *Russian Code Cup 2011*. В последнем столбце без скобок обозначена метрика покрытия строк, в скобках — метрика покрытия решений

Задача	Число строк в решении	Число единичных модификаций	Число компилирующихся решений	Число правильных решений	Время работы (мин.)	Число непокрытых строк проверяющей программы
guess	108	4100	2213 (54%)	926 (23%)	20	0 (0) из 53 (15)
birds	93	3791	2679 (71%)	903 (24%)	3	0 (0) из 40 (7)
queuesort	113	4518	2922 (65%)	1114 (25%)	10	0 (0) из 45 (15)
square	105	2192	1214 (55%)	316 (14%)	3	0 (0) из 37 (13)

Большинство рассмотренных проверяющих программ имеют достаточно простую структуру. Поэтому тестированию была подвергнута еще одна проверяющая программа, которая обладает сложной структурой – программа к задаче *quest*. К данной задаче уже имелось решение на языке *Java*, однако, проверяющая программа была написана на языке *Pascal*. Для тестирования программа была переписана на язык *Java* с сохранением поведения. Результаты тестирования этой проверяющей программы приведены в табл. 8.

Таблица 8. Результаты тестирования проверяющей программы задачи *quest*. В последнем столбце без скобок обозначена метрика покрытия строк, в скобках — метрика покрытия решений

Задача	Число строк в решении	Число единичных модификаций	Число компилирующихся решений	Число правильных решений	Время работы (мин.)	Число непокрытых строк проверяющей программы
quest	337	9652	5911 (61%)	1410 (15%)	60	9 (15) из 280 (92)

3.2.2. Оценка полученных результатов

Как показывают результаты тестирования проверяющих программ полуфинала чемпионата мира по программированию *NEERC2010* (табл. 6), проверяющие программы выполнены на качественном уровне. Разработанный метод достигает полного покрытия исходного кода проверяющей программы и не выявляет ошибок в процессе исполнения проверяющих программ, а также анализа результата их работы. Заметим, что для некоторых задач тестирование занимает значительное время. Это связано с большим временем исполнения правильного решения, которое составляет для этих задач порядка секунды, а также большим размером тестового набора, составляющего более 70 тестов.

Выявленная проблема может иметь несколько решений. Первое из решений состоит в улучшении используемого решения жюри, с целью уменьшения времени его работы. Другим решением может являться исключение некоторых тестов из тестового набора задачи, так как обычно при подготовке задачи тестовый набор дополняется случайно сгенерированными тестами, не проверяющими каких-либо крайних случаев поставленной задачи. От части таких тестов часто можно отказаться. Еще одним решением выявленной проблемы является совершенствование реализации разработанного метода, а именно внедрение возможности параллельного тестирования сразу нескольких модифицированных решений.

При тестировании задач соревнования по программированию *Russian Code Cup 2011* проблем с временем работы выявлено не было. Тем не менее, были выявлены ошибки в проверяющих программах к задачам *guess* и *birds*. Выявленные ошибки были устранены до проведения указанного соревнования. Заметим, что не все ошибки были найдены и устранены с первого раза. Повторное тестирование выявляло новые ошибки в проверяющей программе задачи *birds*, которые также были устранены.

Рассмотрим более подробно ошибку, выявленную в проверяющей программе задачи *birds*. Исходный код проверяющей программы, содержащей ошибку, приведен в Приложении 1. Ошибка проявилась при подаче пустого выходного файла на вход проверяющей программе. Это приводило к ошибке исполнения в 45 строке исходного кода, так как строка *O* являлась пустой. Данная ошибка была устранена с помощью окружения этой строки блоком *try-catch*, сообщающим о некорректном формате выходного файла в случае возникновения ошибки. Исправленный код проверяющей программы приведен на сайте соревнования [10] и в Приложении 2.

Интересными представляются результаты запуска разработанной программы для автоматизированного тестирования проверяющей программы задачи *quest*. В результате тестирования проверяющей программы в ней был выявлен ряд ошибок. Одна из ошибок возникала при генерации сообщения о неправильности ответа участника. Данная ошибка не была ранее обнаружена при использовании данной задачи на соревнованиях. Ошибка была успешно устранена, и проверяющая программа была перетестирована.

Как видно из табл. 8, не все строки проверяющей программы были покрыты. Интересно отметить, что первые три непокрытых строки отвечают созданию сообщения об ошибке в случае некорректного входного файла. Так как разработанный метод не производит каких-либо изменения входных файлов, то данные строки не буду покрыты никогда. Проверка корректности входного файла и частей кода, ее проверяющих, остается на совести программиста, занимающегося подготовкой задачи к соревнованию. Остальные непокрытые стро-

ки соответствую нарушению формата выходного файла. Такие строки наиболее сложны для покрытия, так как иногда требуют от программы вывода совсем странных выходных данных.

Также в проверяющей программе задачи *quest* наблюдается большое число непокрытых решений. Девять из них являются операторами *if*, находящимися непосредственно перед непокрытыми строками. Именно тот факт, что в процессе работы условия операторов всегда не выполнялись, и привел к неполноте покрытия строк исходного кода проверяющей программы. Оставшиеся непокрытые решения соответствуют имеющимся в коде конструкциям *while-true*, условие которых очевидным образом всегда выполняется.

Помимо перечисленных в таблицах задач, на этапе разработки тестированию были подвергнуты проверяющие программы задач из цикла интернет олимпиад сезона 2010 – 2011 гг. [11]. Стоит отметить, что в одной из задач была выявлена часть кода, которая никогда не могла быть исполнена. Данная часть кода была устранена, тем самым был уменьшен объем проверяющей программы.

В процессе тестирования было обнаружено, что наиболее сложными для покрытия строками исходного кода проверяющей программы являются строки, проверяющие некоторые специфические ограничения по формату или содержанию выходного файла решения участника, а также строки соответствующие генерации сообщения в случае невыполнения этих ограничений. Например, блок кода, проверяющий, что выходной файл не содержит более никакой информации, чаще всего будет являться последней покрытой частью проверяющей программы. Стоит отметить, что необходимость именно в таком блоке чаще всего отсутствует, так как данное условие проверяется автоматически.

Результаты показывают, что описанный в работе метод позволяет находить ошибки, не найденные даже при использовании задачи при проведении соревнований. Это свидетельствует об эффективности разработанного в данной работе метода. Однако, стоит отметить, что тестирование проверяющих программ может происходить в течение длительного времени. Для решения этой

проблемы требуется модифицировать метод для параллельного тестирования сразу нескольких мутированных решений.

3.3. Рекомендации по внедрению

Ошибки в проверяющих программах — распространенная проблема при подготовке соревнований по информатике и программированию. Разработанная программа позволяет выявлять многие ошибки в проверяющих программах до проведения соревнования. Использование программы позитивно влияет на качество разрабатываемых проверяющих программ. Также программа для тестирования проверяющих программ упрощает работу программиста, находя крайние случаи, на которых проверяющая программа ведет себя некорректно, или же указывая на проблемные и подозрительные места исходного кода проверяющей программы.

Таким образом, видно, что данный продукт следует применять при подготовке задач для соревнований по информатике и программированию, так как использование данной программы позволяет выявить множество ошибок в проверяющих программах.

3.4. Рекомендацию по улучшению

Разработанный продукт может быть улучшен сразу по ряду позиций. В основном улучшения носят технический характер. Компиляция решений является очень долгим процессом. В связи с этим рекомендуется вести работу напрямую с байт-кодом и применять мутационные операторы именно к нему.

Запуск решений и проверяющей программы не стоит производить в отдельном потоке, так как остановка потока снаружи является неподдерживаемой и скоро может исчезнуть из библиотек. Более того, любая попытка остановки потока является небезопасной. Для тестирования решений и запуска проверяющих программ необходимо создавать новую виртуальную машину и производить запуск в ней. Запуск новой виртуальной машины — долгая операция, поэтому имеет смысл поддерживать пул уже созданных машин, готовых приступить к исполнению кода.

Для более глубокого анализа покрытия исходного кода проверяющей программы возможно добавление и учет других метрик покрытия кода, например покрытия условий.

Такие улучшения позволят более эффективно тестировать проверяющие программы с меньшими временными затратами, что позволит создавать больше мутированных решений и соответственно больше выходных файлов, что позволит добиться более полного покрытия кода, даже в случае затруднительного покрытия.

Выводы

В данной работе была разработана программа, позволяющая выявлять ошибки в проверяющих программах для задач соревнований по информатике и программированию. Данная программа применяет различные мутационные операторы к правильным решениям, чтобы получить большой и разнообразный набор выходных файлов, которые вместе с входными файлами и ответами жюри образуют тестовый набор для проверяющей программы.

Проверяющая программа запускается на полученном тестовом наборе, при этом идет сбор статистических данных об исполнении проверяющей программы, с целью последующего анализа покрытия кода. Также происходит анализ результата работы проверяющей программы. Анализ позволяет обнаружить неправильное или странное поведение проверяющей программы, тем самым указав на возможную ошибку в ее исходном коде.

Как показали испытания, разработанная программа помогает найти сложные ошибки в проверяющих программах, не найденные человеком при просмотре исходного кода и при тестировании. Это указывает на перспективность использования разработанного метода для автоматизированного обнаружения ошибок в проверяющих программах. Указанный метод рекомендуется использовать при подготовке новых соревнований по программированию и информатике.

Список используемых источников

1. *Dijkstra E.* Notes On Structured Programming. 1970. http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.
2. *Miller J.C., Maloney C.J.* Systematic mistake analysis of digital computer programs. Communications of the ACM. 1963. 6(2): 58 – 63.
3. *Clover*: Java code coverage & test optimization. http://www.atlassian.com/software/clover/.
4. *EMMA*: a free Java code coverage tool. http://emma.sourceforge.net/.
5. *Serenity Plugin.* http://wiki.hudson-ci.org/display/HUDSON/Serenity+Plugin.
6. *Ma Y., Offutt J., Kwon Y.* MuJava : An Automated Class Mutation System, Software Testing, Verification and Reliability. 2005. 15(2): 97 – 133. http://cs.gmu.edu/~offutt/rsrch/papers/mujava.pdf.
7. *Ma Y., Kwon Y., Offutt J.* Inter-class mutation operators for Java / 13th International Symposium on Software Reliability Engineering. Annapolis, MD. 2002, pp. 352-363.
8. *Java SE Development Kit (JDK).* http://www.oracle.com/technetwork/java/index.html
9. *North-Eastern European Regional Contest*. http://neerc.ifmo.ru/
10. *Russian Code Cup.* http://russiancodecup.ru/
11. *Интернет-олимпиады по информатике.* http://neerc.ifmo.ru/school/io/
12. *Code Coverage*. http://en.wikipedia.org/wiki/Code_coverage
13. *Mutation Testing*. http://en.wikipedia.org/wiki/Mutation_testing
14. *Jenkov J.* Dynamic Class Loading and Reloading in Java. http://tutorials.jenkov.com/java-reflection/dynamic-class-loading-reloading.html
15. *Budd T., DeMillo R., Lipton R., Sayward F.* The design of a prototype mutation system for program testing / AFIPS Conference Record. 1978, pp. 623-627.
16. *Programming tutorials and source code examples.* http://java2s.com/

Приложение 1

Код проверяющей программы задачи *birds* второго квалификационного раунда соревнования *Russian Code Cup 2011*, содержащий логическую ошибку. Желтым выделена строка, на которой происходит ошибка времени исполнения проверяющей программы при пустом выходном файле участника.

```
[1]    import ru.ifmo.testlib.InStream;
[2]    import ru.ifmo.testlib.Outcome;
[3]    import ru.ifmo.testlib.Outcome.Type;
[4]
[5]    public class Check implements ru.ifmo.testlib.Checker {
[6]
[7]        String nosol = "impossible";
[8]        String ok = "ok";
[9]        double EPS = 1e-4;
[10]
[11]       double getDist(double al, double v, double a, double x0,
                                              double y0, double t) {
[12]           double vx = v * Math.cos(al);
[13]           double vy = v * Math.sin(al);
[14]           double x = vx * t;
[15]           double y = vy * t - a * t * t * .5;
[16]           double dx = vx;
[17]           double dy = vy - a * t;
[18]           double A = dy;
[19]           double B = -dx;
[20]           double C = -A * x - B * y;
[21]           double yA = (-C - A * x0) / B;
[22]           return Math.abs(yA - y0);
[23]       }
[24]
[25]       @Override
[26]       public Outcome test(InStream inf, InStream ouf, InStream ans){
[27]           String O = ouf.nextLine();
[28]           String A = ans.nextLine();
[29]           O = O.toLowerCase();
[30]           A = A.toLowerCase();
[31]
[32]           if (O.equals(nosol) && A.equals(nosol))
[33]               return new Outcome(Type.OK, "Impossible");
[34]
[35]           if (O.equals(nosol) && !A.equals(nosol))
[36]               return new Outcome(Type.WA, "Solution exists,
                                  but participant hasn't found one");
[37]
[38]           double al = Math.toRadians(inf.nextInt());
[39]           double v = inf.nextInt();
[40]           double a = inf.nextInt();
[41]           double x0 = inf.nextInt();
[42]           double y0 = inf.nextInt();
[43]
[44]           if (!A.equals(ok)) {
[45]               double t = Double.parseDouble(O);
[46]               if (t < -EPS) {
```

```
[47]                 return new Outcome(Type.WA,"Negative time: " + t);
[48]             }
[49]             double d = getDist(al, v, a, x0, y0, t);
[50]             if (d >= EPS) {
[51]                 return new Outcome(Type.WA, "The bird is too far
                                                  from the pig: " + d);
[52]             }
[53]             if (x0 + EPS > v * Math.cos(al) * t) {
[54]                 return new Outcome(Type.OK, "Ok, distance: " + d);
[55]             } else {
[56]                 return new Outcome(Type.WA, "The bird needs to fly
                                        to the left to hit the pig.");
[57]             }
[58]         } else {
[59]             if (A.equals(ok))
[60]                 return new Outcome(Type.OK, "Ok");
[61]             double t = x0 / (v * Math.cos(al));
[62]             double y = v * Math.sin(al) * t - a * t * t * .5;
[63]             if (Math.abs(y - y0) > EPS) {
[64]                 return new Outcome(Type.WA, "Expected: " + A + ",
                                                    found: " + O);
[65]             }
[66]             return new Outcome(Type.OK, "Ok");
[67]         }
[68]     }
[69] }
```

Приложение 2

Код проверяющей программы задачи *birds* второго квалификационного раунда соревнования *Russian Code Cup 2011*, не содержащий ошибок.

```
[1]    import ru.ifmo.testlib.InStream;
[2]    import ru.ifmo.testlib.Outcome;
[3]    import ru.ifmo.testlib.Outcome.Type;
[4]
[5]    public class Check implements ru.ifmo.testlib.Checker {
[6]
[7]        String nosol = "impossible";
[8]        String ok = "ok";
[9]        double EPS = 1e-4;
[10]
[11]       double getDist(double al, double v, double a, double x0,
                                                 double y0, double t) {
[12]           double vx = v * Math.cos(al);
[13]           double vy = v * Math.sin(al);
[14]           double x = vx * t;
[15]           double y = vy * t - a * t * t * .5;
[16]           double dx = vx;
[17]           double dy = vy - a * t;
[18]           double A = dy;
[19]           double B = -dx;
[20]           double C = -A * x - B * y;
[21]           double yA = (-C - A * x0) / B;
[22]           return Math.abs(yA - y0);
[23]       }
[24]
[25]       @Override
[26]       public Outcome test(InStream inf, InStream ouf, InStream ans){
[27]           String O = ouf.nextLine().toUpperCase();
[28]           String A = ans.nextLine().toUpperCase();
[29]           O = O.toLowerCase();
[30]           A = A.toLowerCase();
[31]
[32]           if (O.equals(nosol) && A.equals(nosol))
[33]               return new Outcome(Type.OK, "Impossible");
[34]
[35]           if (O.equals(nosol) && !A.equals(nosol))
[36]               return new Outcome(Type.WA, "Solution exists,
                                 but participant hasn't found one");
[37]
[38]           double al = Math.toRadians(inf.nextInt());
[39]           double v = inf.nextInt();
[40]           double a = inf.nextInt();
[41]           double x0 = inf.nextInt();
[42]           double y0 = inf.nextInt();
[43]
[44]           if (!O.equals(ok)) {
[45]               double t;
[46]               try {
[47]                   t = Double.parseDouble(O);
[48]               } catch (NumberFormatException e) {
```

```
[49]                 return new Outcome(Type.PE, "Unexpected output: "
                                                              + O);
[50]             }
[51]             if (t < -1e-9) {
[52]                 return new Outcome(Type.WA, "Negative time: "+t);
[53]             }
[54]             double d = getDist(al, v, a, x0, y0, t);
[55]             if (d > EPS) {
[56]                 return new Outcome(Type.WA, "The bird is too far
                                                 from the pig: " + d);
[57]             }
[58]             if (x0 + EPS > v * Math.cos(al) * t) {
[59]                 return new Outcome(Type.OK, "Ok, distance: " + d);
[60]             } else {
[61]                 return new Outcome(Type.WA, "The bird needs to fly
                                        to the left to hit the pig.");
[62]             }
[63]         } else {
[64]             double t = x0 / (v * Math.cos(al));
[65]             double y = v * Math.sin(al) * t - a * t * t * .5;
[66]             if (Math.abs(y - y0) > EPS) {
[67]                 return new Outcome(Type.WA, "Expected: " + A +
                                                  ", found: " + O);
[68]             }
[69]             return new Outcome(Type.OK, "Ok");
[70]         }
[71]     }
[72] }
```