ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

Факультет *Информационных Технологий и Программирования*

Направление (специальность) *Прикладная математика и информатика*

Квалификация (степень) *Магистр прикладной математики и информатики*

Кафедра *Компьютерных технологий* Группа *6536*

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

Генерация покрывающих наборов тестов для систем, представленных в стандарте IEC 61499, на основе эволюционных алгоритмов

Автор магистерской диссертации *Бужинский И.П.*

Научный руководитель *Шалыто А.А.*

Руководитель магистерской программы *Васильев В.Н.*

К защите допустить

Заведующий кафедрой *Васильев В.Н.*

“____”____________________2015 г.

Санкт-Петербург, 2015 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ **5**

ГЛАВА 1. ОБЗОР ЛИТЕРАТУРЫ **7**

1.1. Стандарт IEC 61499 7

1.1.1. Функциональные блоки 8

1.1.2. Базовые функциональные блоки 9

1.1.3. Составные функциональные блоки 11

1.1.4. Функциональные блоки интерфейсов служб 13

1.2. Автоматизация тестирования ПО 15

1.2.1. Тестирование на основе модели 16

1.2.2. Критерии покрытия на основе модели 17

1.2.5. Тесты, наборы тестов и критерии покрытия в применении к рассматриваемой задаче 19

1.2.3. Автоматизация тестирования на основе удовлетворения ограничений 21

1.2.4. Автоматизация тестирования на основе эволюционных вычислений 23

1.3. Эволюционные вычисления 23

1.3.1. Эволюционные операторы 24

1.3.2. Эволюционные алгоритмы 27

Выводы по главе 1 28

ГЛАВА 2. МЕТОД, ОСНОВАННЫЙ НА СТОРОННИХ ПРОГРАММНЫХ СРЕДСТВАХ **29**

2.1. Общая схема метода и обоснование принятых решений 29

2.2. Первый этап 32

2.3. Второй этап 32

2.4. Третий этап 33

2.5. Недостатки и ограничения подхода 34

Выводы по главе 2 36

ГЛАВА 3. МЕТОД, ОСНОВАННЫЙ НА СОБСТВЕННОМ ПРЕДСТАВЛЕНИИ ТЕСТОВ И ФУНКЦИОНАЛЬНЫХ БЛОКОВ **37**

3.1. Трансляция функциональных блоков в код на языке Java 37

3.2. Реализация эволюционного алгоритма 39

3.2.1. Функции приспособленности 40

3.2.2. Оператор мутации ... 40
Выводы по главе 3 ... 41
ГЛАВА 4. ИССЛЕДОВАНИЕ ПРЕДЛОЖЕННЫХ МЕТОДОВ ... 42
4.1. Тестируемые системы ... 42
4.2. Описание экспериментов ... 45
4.3. Результаты ... 46
4.3.1. Обзор результатов ... 47
4.3.2. Детальное изучение сгенерированных наборов тестов ... 49
4.4. Сравнение предложенных подходов с тестированием на основе модели ... 51
Выводы по главе 4 ... 53
ЗАКЛЮЧЕНИЕ ... 55
СПИСОК ЛИТЕРАТУРЫ ... 57
ПРИЛОЖЕНИЕ А. ПРИМЕР КОДА НА ЯЗЫКЕ JAVA, ПОДГОТОВЛЕННОГО К ЗАПУСКУ EVOSUITE ... 61
ПРИЛОЖЕНИЕ Б. ПРИМЕР НАБОРА ТЕСТОВ, СОЗДАННОГО EVOSUITE ... 67

ВВЕДЕНИЕ

Управляющее программное обеспечение (ПО) – это важный элемент современных систем промышленной автоматизации [35], примерами которых являются системы производства и обработки материалов. Оно отвечает за безопасность и корректность их работы. Это означает, что такие системы должны быть в достаточной степени протестированы или верифицированы. Один из современных стандартов для разработки подобных систем управления – это IEC 61499 [20], основанный на концепции функциональных блоков – программных единиц систем управления. Стандарт нацелен на увеличение гибкости и адаптивности таких систем [35], а также на полноценную поддержку распределенного управления.

Один из способов удостовериться в том, что управляющие системы работают корректно – это тестирование [19]. Во время тестирования для проверяемой системы готовится *набор тестов*. Каждый тест является последовательность вызовов к элементам системы (например, при тестировании объектно-ориентированной программы тест может состоять из вызовов методов определенного класса). Системы, рассматриваемые в настоящей диссертации, имеют интерфейсы, характеризуемые событиями и переменными, поэтому тесты для них будут состоять из посылок событий с привязанными к ним значениями переменных. Такие посылки так же могут быть представлены как вызовы методов.

В отличие от формальной верификации ПО, тестирование не может полностью обеспечить корректность систем на практике, но все же может выявить большое число ошибок. Кроме того, выполнение тестов обычно занимает меньше времени, чем проведение верификации. Более того, создание тестов можно автоматизировать. Простой формальной мерой качества набора тестов является *покрытие*, которые можно определить рядом способов.

В рамках предыдущих исследований многое было сделано в области тестирования на основе модели (model-based testing) [3]. Во-первых, были определены различные критерии покрытия, включая явно использующие конечные автоматы [7] – сущности, широко задействованные в стандарте IEC 61499. Во-

вторых, были разработаны методы генерации тестов, нацеленные на покрытие моделей ПО, основанных на спецификации. Тем не менее, настоящая магистерская диссертация ставит своей целью разработку метода генерации тестов, покрывающих *реализацию* ПО, представленного в стандарте IEC 61499. Генерация тестов для покрытия реализации ПО – это тоже известная задача, и одно из ее недавних решений [13] использует эволюционный подход [17]. В рамках этого подхода в процессе симулируемой эволюции исследуются возможные решения задачи (в нашем случае, наборы тестов). В процессе поиска решения изменяются (подвергаются мутациям) и комбинируются друг с другом.

Насколько известно автору диссертации, эволюционный подход, а также другие подходы тестирования «черного ящика» ранее не применялись к ПО для систем промышленной автоматизации и, в частности, к системам управления, представленным согласно стандарту IEC 61499. В настоящей магистерской диссертации как раз и рассматривается такое применение. В ней предложены два метода генерации наборов тестов. Первый метод использует сторонние программные средства и основан на разделении задачи на две части: трансляцию тестируемого функционального блока в исходный код на языке программирования общего назначения и оптимизацию покрытия этого кода. Во втором подходе используется похожая схема, но он гораздо менее привязан к сторонним программным средствам, что делает его более гибким. Показывается, как применить предложенные методы, которые различаются своими достоинствами и недостатками, на практике: методы оказываются способны обнаружить реальные дефекты в управляющих приложениях.

Диссертация имеет следующую структуру. В главе 1 описывается предметная область исследования: стандарт IEC 61499, автоматизация тестирования ПО и эволюционные вычисления. Основной вклад диссертации, а именно два метода генерации наборов тестов, представлен в главах 2 и 3. Наконец, исследование и сравнение предложенных методов проводится в главе 4, после чего заключение подводит итог диссертации.

ГЛАВА 1. ОБЗОР ЛИТЕРАТУРЫ

В настоящей главе будут рассмотрены три области, имеющие большое значение в контексте настоящей диссертации. Во-первых, будет рассмотрен стандарт IEC 61499, включая его особенности, существенные для постановки задачи диссертации. Далее будут рассмотрены существующие технологии автоматизации тестирования ПО, и в особенности те, которые могут быть использованы для решения задачи. В заключение, будут более подробно рассмотрены эволюционные вычисления – подход, которым было решено воспользоваться по результатам исследования существующих решений.

1.1. Стандарт IEC 61499

IEC 61499 [20] – это открытый стандарт для разработки распределенных управляющих приложений для систем промышленной автоматизации, предложенный Международной электротехнической комиссией (International Electrotechnical Commission, IEC) в 2005 году. Цель создания стандарта заключалась в предоставлении разработчикам возможности работать с распределенными системами управления, которые могут быть размещены на множестве программируемых логических контроллеров (ПЛК) и состоят из повторно используемых модулей. В настоящее время осуществляются попытки применения стандарта в промышленности. Примером такого применения является производство обуви [6]. Однако применение этого стандарта испытывает несколько сложностей [31, 15], связанных с непривычностью семантики стандарта для инженеров и невозможностью охвата стандартом всех фаз процесса разработки (например, стандарт не применим к выделению требований к системе). Несколько программных средств поддерживают разработку систем, представленных в стандарте IEC 61499. Среди них *ISaGRAF* [33], *NxtStudio* [26] и *FBDK* [33].

Стандарт IEC 61499 предлагает рассматривать управляющее приложение как набор функциональных блоков (ФБ), базовых и составных, которые связаны друг с другом в сеть. Концепция функционального блока будет более детально изучена в

следующих подразделах настоящего раздела. Принятым форматом представления ФБ является формат XML.

1.1.1. Функциональные блоки

ФБ – это сущность с определенным интерфейсом, которая совмещает в себе поведение и состояние. Таким образом, типы ФБ и их экземпляры близки к классам и объектам из объектно-ориентированного программирования соответственно, однако они не поддерживают ни наследование, ни полиморфизм.

Для начала определим *интерфейс ФБ*, который является частью ФБ обеих разновидностей. В настоящей диссертации под интерфейсом ФБ будет подразумеваться восьмерка $(E_I, V_I, D_I, E_O, V_O, D_O, M_I, M_O)$, где E_I – это множество *входных событий*, V_I – множество *входных переменных*, $D_I = \{ D_1, \ldots, D_{|V_I|} \}$ множество областей значений переменных (все области значений конечны и прямо соответствуют типам переменных, типичным для языков программирования общего назначения: BOOL, INT, REAL, TIME, ARRAY и т.д., и которые описаны в стандарте), E_O – множество *выходных событий*, также называемых *выходными воздействиями*, а V_O и D_O – множества *выходных переменных* и их областей значений соответственно.

Наконец, M_I и M_O – это булевы (то есть состоящие из нулей и единиц) матрицы размеров $|E_I|\times|V_I|$ и $|E_O|\times|V_O|$ соответственно, которые определяют, между какими событиями и переменными есть *ассоциации*. Каждый шаг выполнения ФБ вызывается передачей ему одного из входных событий. События всегда обрабатываются последовательно. Наличие ассоциации между входным событием и входной переменной означает, что когда событие приходит к ФБ, значение входной переменной обновляется значением, выработанным другим ФБ. Далее, в результате шага выполнения ФБ может быть выработаны выходные события и обновлены значения выходных переменных. Если выходное событие связано с некоторой выходной переменной, то тогда обновленное значение выходной переменной становится доступно другим ФБ в момент выработки события. Более точное определение ассоциации будет дано в подразделе 1.1.3.

На рис. 1 приведена общая схема ФБ, а пример конкретного ФБ показан на рис. 2. У этого ФБ три входных события (E1, E2 и E3), две входные переменные (булева переменная BOOL_VAR и целочисленная переменная INT_VAR), одно выходное воздействие (O1) и одна булева выходная переменная OV. Входное событие E2 ассоциировано с обеими входными переменными, входное событие E3 ассоциировано с BOOL_VAR, а выходное событие O1 ассоциировано с выходной переменной OV. Из рисунков видно, что ФБ обычно представляют в графической нотации «головы и тела», где события находятся в «голове», а переменные – в «теле».

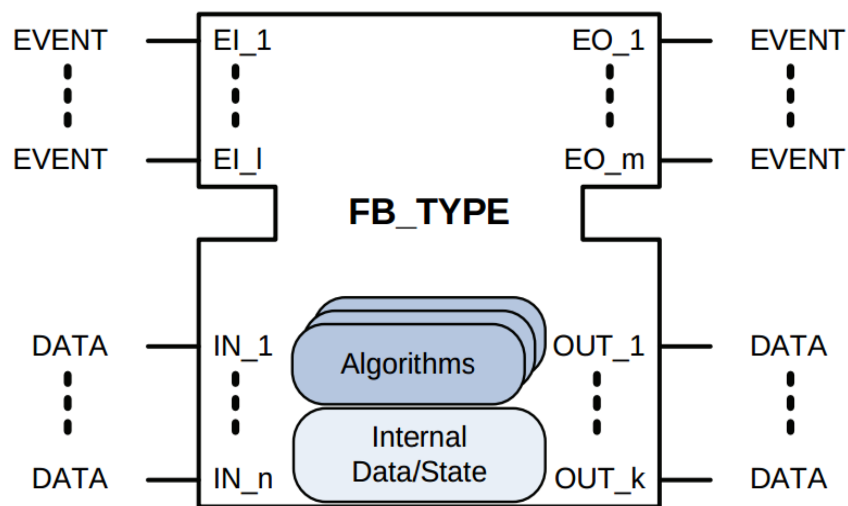

Рис. 1. Общая схема ФБ

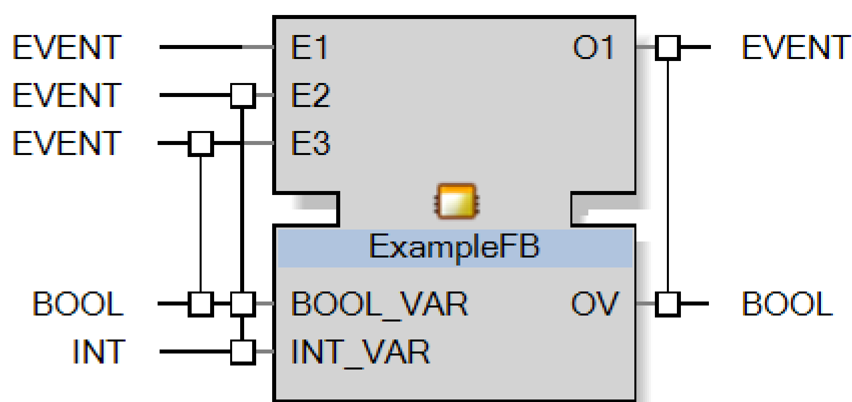

Рис. 2. Пример конкретного ФБ

1.1.2. Базовые функциональные блоки

В соответствии со стандартом, базовые ФБ реализованы с использованием концепции *управляющих диаграмм выполнения* (execution control charts, ECCs),

которые являются управляющими конечными автоматами. Формально, базовый ФБ – это восьмерка $(I, V, D, S, s_0, \delta, \lambda, \alpha)$, где I – это интерфейс ФБ, V и D – множества *внутренних переменных* и их областей значений, S – это множество *состояний* (в каждый момент времени только одно из них активно), s_0 – *начальное состояние*, $\delta : S \times E \times D_1 \times \ldots \times D_{|VI|} \to S$ – это *функция переходов*, которая определяет новое состояние диаграммы при получении ей события, $\lambda : S \to 2^{E_O}$ – это *функция выходов*, определяющая выходные события для каждого состояния, а $\alpha : S \to L$ – это *функция алгоритмов*, которая определяет для каждого состояния алгоритм (на языке L, которым обычно является язык *Structured Text*), который будет выполнен при активации состояния. Алгоритмы могут оперировать со всеми тремя разновидностями переменных: с входными, выходными и внутренними. В частности, только они способны обновлять значения внутренних переменных. Также предполагается, что у всех переменных внутри базового ФБ есть значения по умолчанию, специфичные для типа переменной (например, «ложь» для типа BOOL).

Управляющие диаграммы – это графическое представление базовых ФБ. В этих диаграммах состояния связаны друг с другом *переходами.* Переходы обычно вызываются событиями, что возможно только тогда, когда выполнены *охранные условия* перехода. Эти условия определены над множеством входных переменных V_I, что отражено в области определения функции переходов δ. Выбор переходов, который будет выполнен в каждый момент времени, всегда детерминирован: ситуации, когда могут выполниться сразу несколько переходов из текущего состояния, разрешаются приоритетами переходов. Возможно также, что при возникновении события не может выполниться ни один переход. Более того, одно событие может вызвать сразу несколько изменений состояния, что происходит из-за *спонтанных переходов*, которым не нужны события для выполнения. Если из текущего состояния есть переход с удовлетворенным охранным условием, то он всегда будет выполнен (за исключением ситуации, когда может выполниться другой переход с более высоким приоритетом).

Приход события к ФБ может вызвать его реакцию: выходное событие (возможно, несколько событий или даже бесконечную последовательность

событий), изменение состояния и изменение значений переменных. Отсутствие реакции может быть объяснено отсутствием выработки события в текущем состоянии ФБ, отсутствием ассоциаций между событиями и переменными (даже если событие выработано, новые данные в этом случае не будут видны другим ФБ), или бесконечным циклом в управляющей диаграмме. Когда управляющая диаграмма находится в состоянии ожидания (то есть, нет переходов, которые могут выполниться в настоящий момент), состояние ФБ полностью определяется значениями его переменных и состоянием управляющей диаграммы.

Пример управляющей диаграммы ФБ показан на рис. 3. Эта диаграмма совместима с интерфейсом, приведенным на рис. 2, и поэтому вместе с ним определяет полноценный базовый ФБ. У диаграммы три состояния, два из которых (S1 и S2) связаны с алгоритмами (ALG_T и ALG_F), и одно из которых (S1) имеет выходное воздействие (O1). Алгоритмы ALG_T и ALG_F изменяют значение булевой выходной переменной OV.

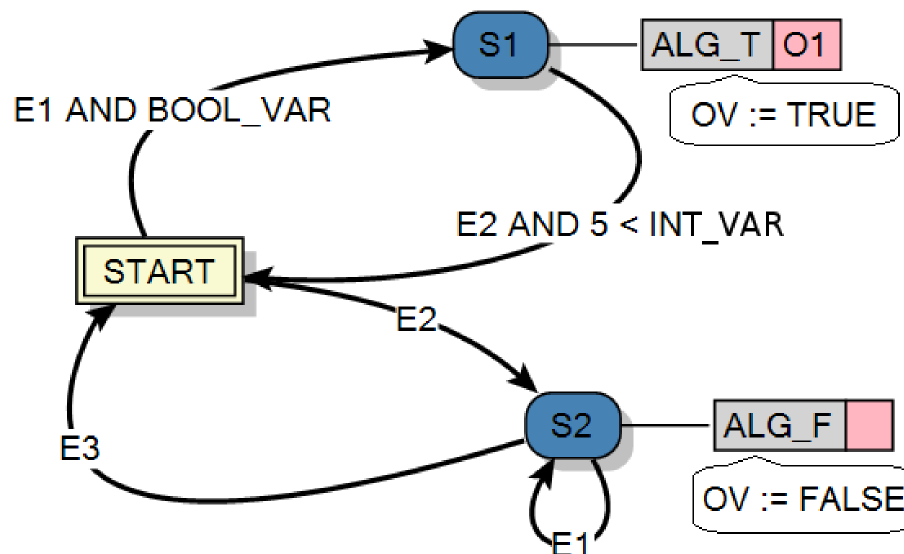

Рис. 3. Пример управляющей диаграммы базового ФБ

1.1.3. Составные функциональные блоки

Внутри составного ФБ находится сеть ФБ других типов, соединенных по событиям и по данным. Составной ФБ – это пятерка (I, B, C_E, C_D, P), где I – это интерфейс ФБ, B – множество *вложенных ФБ*, C_E и C_D – множества *соединений по событиям и по данным*, а P – множество предопределенных значений входных

переменных для вложенных ФБ. Каждое соединение связывает выходы и входы (по событиям или переменным) вложенных ФБ друг с другом, или, что также возможно, с входами и выходами самого составного ФБ. Предопределенные значения переменных полезны в случае отсутствия соединения по данным для некоторой входной переменной некоторого вложенного ФБ. Пример составного ФБ приведен на рис. 4 (этот снимок экрана сделан в *NxtStudio*). Интерфейс этого ФБ изображен слева и справа от вложенных ФБ.

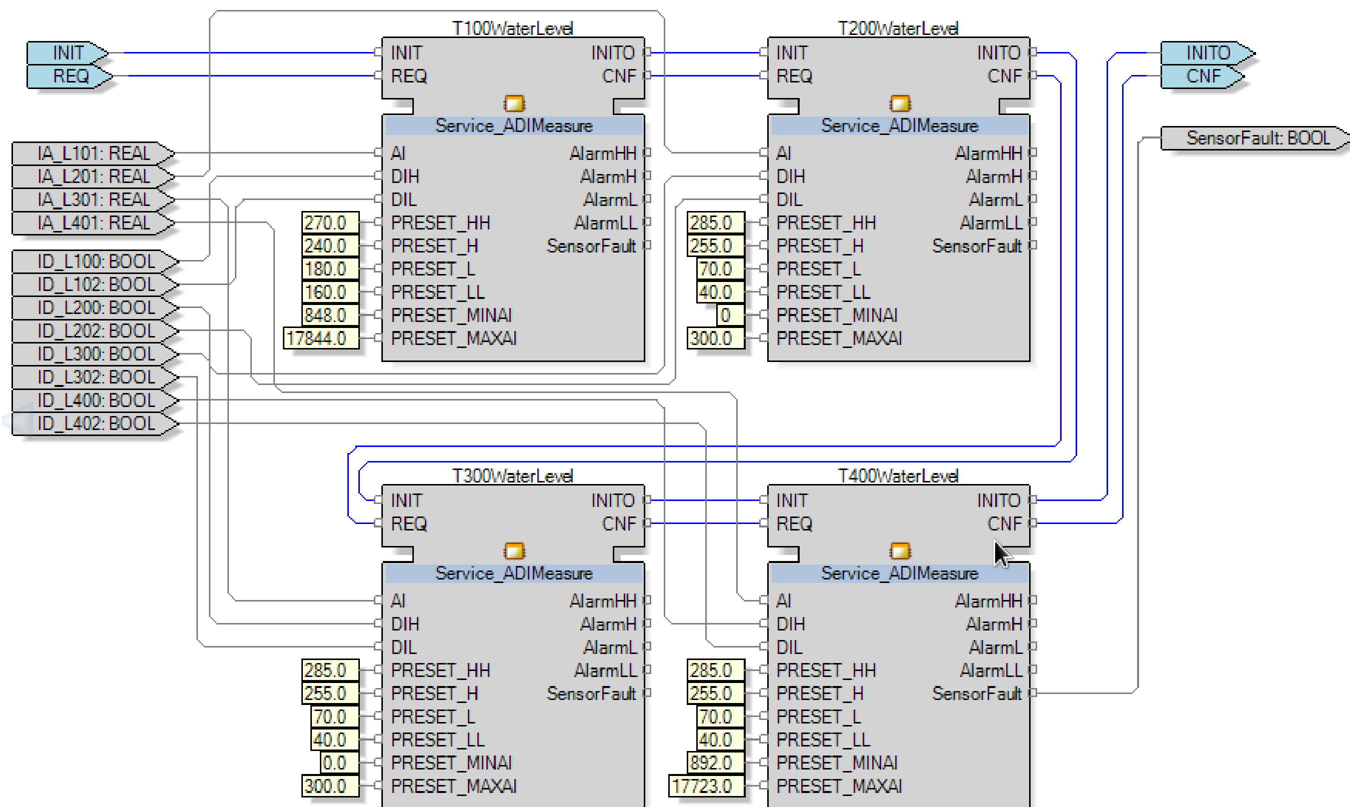

Рис. 4. Пример составного ФБ

Составные ФБ являются способом многократного использования ФБ более низкого уровня. Кроме того, нет необходимости размещать все вложенные ФБ внутри составного ФБ на одном устройстве – то есть, составные ФБ могут представлять собой распределенные управляющие системы. Они являются универсальным способом представления управляющих систем – как распределенных, так и централизованных.

Когда некоторое событие приходит к составному ФБ, оно распространяется до вложенных ФБ, с которыми оно связано. Это событие вызывает шаг выполнения каждого из этих ФБ, которые, в свою очередь, могут выработать новые события.

Стандарт предполагает, что события внутри составного ФБ распространяются в порядке обхода в ширину [7].

В заключение, более точно опишем семантику ассоциаций между событиями и переменными. Пусть выходное событие e_1 ассоциировано с выходной переменной v_1 в ФБ fb_1, а входное событие e_2 ассоциировано с входной переменной v_2 в ФБ fb_2. Если e_1 вырабатывается fb_1, а затем сразу же или через некоторое время e_2 поступает на вход к fb_2 (e_2 не обязано происходить из fb_1 и даже быть такого же типа: возможно, что ФБ принимает событие из одного ФБ и читает значения переменных, пришедшие из другого ФБ), то v_2 будет обновлена значением v_1 в момент выработки e_1. Если хотя бы одна из упомянутых ассоциаций отсутствует, обновление не произойдет. Однако некоторые реализации стандарта нарушают описанные принципы. Например, реализация стандарта в средстве разработки *FBDK* предполагает, что события внутри составных ФБ распространяются в порядке обхода в глубину [7] и что все изменения выходных переменных становятся незамедлительно доступными тем ФБ, которые получают их значения как входные, вне зависимости от существующих ассоциаций между событиями и переменными.

1.1.4. Функциональные блоки интерфейсов служб

До текущего момента мы не рассматривали взаимодействие ФБ с физическими устройствами. *ФБ интерфейсов служб* отвечают за подобную деятельность. Они представляют собой низкоуровневые службы, реализуемые программным или аппаратным обеспечением устройств.

Поведение ФБ интерфейсов служб сложно учитывать при автоматической генерации тестов: для этого необходимо иметь формальные модели устройств. Однако, поскольку ФБ интерфейсов служб часто ответственны за операции ввода/вывода, они могут быть опущены и заменены входами и выходами составного ФБ, в котором они находились. Это может быть сделано следующим образом. Предположим, что в составном ФБ самого верхнего уровня находится некоторый ФБ интерфейса службы. Каждый входной и выходной элемент его интерфейса (событие или переменная) будет преобразован в выходной или входной элемент интерфейса составного ФБ соответственно. После этого ФБ интерфейса службы

может быть удален. В дальнейшем будет предполагаться, что базовые и составные ФБ – это единственные разновидности ФБ, имеющиеся в тестируемой системе. Тем не менее, ФБ интерфейсов служб тоже могут содержать ошибки, поэтому исключение их из рассмотрения в настоящей диссертации является одним из ограничений исследования.

1.2. Автоматизация тестирования ПО

Как было упомянуто во введении, тестирование ПО – это один из способов обеспечения его надежности. Известно множество видов тестирования: модульное, функциональное, интеграционное тестирование, нагрузочное и стресс-тестирование, и так далее [19]. Ручное создание тестов часто является тяжелым трудом, поскольку разработчик тестов должен вручную проверить различные пути выполнения приложения. Поэтому, чтобы уменьшить стоимость тестирования и увеличить его надежность, появилась автоматизация тестирования, и в частности, автоматизация генерации входных тестовых данных [11].

В настоящей диссертации предметами интереса являются функциональное и модульное тестирование управляющих приложений для систем промышленной автоматизации. Для таких систем особенно важным этапом тестирования является заводское приемочное тестирование (factory acceptance testing), являющееся первым интеграционным тестом для приложения [28]. Другой подход, имеющий значение для подобных систем – это тестирование управляющих контуров, которое «проверяет связность входов и выходов, стратегию управления и аспекты безопасности контура управления на соответствие со спецификацией» [28]. В работе [28] также говорится, что анализ покрытия и генерация тестов на основе исходного кода – это одна из целевых областей функционального тестирования ПО. Эта область весьма близка к той, которая рассматривается в настоящей диссертации.

В настоящем разделе будут изучены разнообразные подходы к автоматизации тестирования ПО. Сначала будет рассмотрена широкая область тестирования на основе модели [3], сильно связанная с применением конечных автоматов. Обзор этой области позволит дать формальные определения тестам и критериям покрытия, которые будут использоваться далее. После этого будут рассмотрены несколько других подходов, среди которых подходы, основанные на удовлетворении ограничений, а также эволюционный подход. Последний из них и будет применен в настоящей диссертации.

1.2.1. Тестирование на основе модели

Общая идея *тестирования на основе модели* (*model-based testing*, MBT) [3] заключается в использовании формальных моделей ПО, вручную создаваемых на основе требований к нему, для анализа систем и генерации наборов тестов, которые способны проверить ПО на соответствие его спецификации. Для реактивных систем, на которых акцентирована настоящая диссертация, такими моделями могут служить конечные автоматы. В MBT важной целью при решении задачи генерации тестов является, как правило, *покрытие моделей*. Тем не менее, в настоящей диссертации рассматривается *покрытие* тестами *реализации* приложения. Однако в силу того, что в стандарте IEC 61499 конечные автоматы играют ключевую роль, во время разработки метода генерации тестов могут быть применены многие идеи из области MBT. В настоящем подразделе будут рассмотрены несколько определений критериев покрытия и несколько методов генерации покрывающих наборов тестов.

Согласно [3], известно три основных подхода к генерации тестов: автоматическое доказательство теорем, символьное выполнение и проверка моделей (model checking). *Автоматическое доказательство теорем* связано с разбиением модели ПО на несколько классов эквивалентности, таких что каждый класс ответственен за наличие или отсутствие конкретной ошибки. После определения классов эквивалентности каждый из них рассматривается как отдельный тест. Разбиение модели осуществляется на основе спецификации, записанной на формальном языке. Чисто классов эквивалентности может быть разным и, в частности, может зависеть от размера спецификации. Например, в [18] предикаты из спецификации преобразуются в дизъюнктивную нормальную форму (ДНФ) и число классов эквивалентности оказывается равным числу полученных дизъюнктов.

Вторая техника – *символьное выполнение* – тоже применяется при формальной верификации. В рамках этой техники входы системы заменяются символами (переменными и ограничениями на их значения). Символьное выполнение применимо как для моделей, так и для программного кода. Примером применения этого подхода является работа [32].

Последний подход – это *проверка моделей* (*model checking*) [5], и это снова один из методов формальной верификации. Один из вариантов проверки моделей заключается в сопоставлении модели системы ее темпоральной спецификации (например, представленной в логиках LTL или CTL). В случае несоответствия алгоритмы верификации генерируют контрпримеры, объясняющие, почему модель не удовлетворяет спецификации. Чтобы применить проверку моделей для генерации тестов, спецификацию для тестов нужно выразить в виде темпоральных свойств, и тогда проблема будет сведена к нахождению контрпримеров. Например, этот подход применяется в [12].

1.2.2. Критерии покрытия на основе модели

Критерии покрытия наборов тестов используют для оценки их качества. Многие критерии можно рассматривать как в бинарном (есть покрытие или нет), так и в процентом значении. Согласно [3], выделяют три типа критериев покрытия: структурные, функциональные и стохастические. *Структурные критерии* основаны на структуре модели – в нашем случае, на структуре конечных автоматов:

- *Покрытие состояний* требует, чтобы все состояния автомата были посещены в процессе выполнения тестов.
- *Покрытие переходов* предполагает, что покрыты все переходы конечно-автоматной спецификации.
- *Покрытие внутренности границы* (*boundary interior coverage*) означает покрытие всех циклов не менее определенного числа раз.
- *Покрытие путей* – сильнейший и наиболее трудный для достижения критерий – предполагает, что каждый путь в спецификации покрыт как минимум одним тестом.
- Для обеспечения *покрытия турне* (*round-trip coverage*) [1] необходимо, чтобы была покрыта каждая последовательность переходов, начинающаяся и заканчивающаяся в одном и том же состоянии (например, в начальном). Более точно, каждый переход и цикл на каждом таком пути должен быть посещен как минимум один раз.

Функциональные критерии покрытия предполагают, что в распоряжении инженера по тестированию помимо спецификации имеется некоторая модель внешней для приложения среды. Эта модель задает несколько возможных сценариев поведения системы и таким образом ограничивает ими возможные тесты. Такие сценарии могут служить источниками ожидаемых выходных воздействий реализации приложения, в то время как входные части теста можно получить на основе спецификации.

Последняя разновидность критериев покрытия представлена *стохастическими критериями*. Эти критерии основаны на вероятностях посещения различных частей спецификации, которые вычисляются на основе действий пользователя. В простейшем случае, когда выполнение всех переходов из каждого состояния модели равновероятно, тесты выбираются случайным и равномерным образом.

Выделяют также критерии, *ориентированные на поток управления и на поток данных*. Критерии, ориентированные на поток управления, задаются в терминах решений и условий, которые обычно выражают условным оператором:

- Для *покрытия решений*, или *покрытия ветвей*, необходимо, чтобы был покрыт каждый исход каждого решения в спецификации. Например, при наличии условного оператора как минимум одним тестом должна быть покрыта и then, и else-ветвь этого оператора.
- *Покрытие условий требует* покрытия всех исходов каждого условия внутри каждого решения. Например, если решение имеет вид A and B and C, то должен быть покрыт каждый исход всех трех условий A, B и C.
- Достижение *покрытия решений и условий* означает, что одним и тем же набором тестов достигнуты и покрытие решений, и покрытие условий.
- Наконец, *покрытие множественных условий* определяется как покрытие каждой комбинации условий. Например, в составном решении (A and B) or (C and D) внутри условного оператора или оператора цикла должны быть протестированы все 16 комбинаций исходов условий A, B, C и D.

Критерии покрытия, основанные на потоке данных, в отличие от тех, что основаны на потоке управления, определяются в терминах графа потока управления (control flow graph, CFG) и путей в этом графе. Граф потока управления строится на основе программы (например, на основе скомпилированной версии модели) как набор линейных вычислений (вершины графа) и решений, которые передают управление между узлами. Основная идея, на которой основаны критерии – рассмотреть пути в графе, начинающиеся с определения некоторой переменной и ведущие в места, где она используется. В настоящем обзоре мы опустим рассмотрение конкретных критериев покрытия, основанных на потоке данных, поскольку их описание требует большого теоретического введения.

Другая известная техника – это *тестирование, основанное на разбиении* (partition based testing) [14]. В ее рамках предлагается разделить область входных значений (например, набор возможных значений некоторой входной переменной) на несколько подобластей. Такие разбиения могут быть получены из условий. Требование на набор тестов, которые нужно выполнить, заключается в том, чтобы для каждой подобласти в нем был хотя бы один тест с входным значением из этой области. Это делает критерии, основанные на разбиении, похожими на структурные критерии, рассмотренные нами ранее.

1.2.5. Тесты, наборы тестов и критерии покрытия в применении к рассматриваемой задаче

В этом подразделе будут приведены некоторые определения, которые в дальнейшем будут использоваться в диссертации, и будет формально поставлена ее задача. Чтобы определить тест, для начала зафиксируем тестируемый ФБ. Пусть его входные события – $E_1, \ldots, E_n$, а его входные переменные – $V_1, \ldots, V_m$ с областями значений $D_1, \ldots, D_m$ соответственно. Далее, булевы значения W_{ij} будут означать, ассоциировано ли событие E_i с переменной V_j. Мы также будем рассматривать специальное значение $\bot$, которое не принадлежит ни одному из множеств D_j, $j = 1..m$. Это значение служит для представления отсутствующих значений переменных для случая, когда входное событие не ассоциировано с некоторой входной переменной.

Входной кортеж – это кортеж (E_i, α_1, …, α_m), где α_j, $j = 1..m$ либо принадлежит D_j (в случае истинности W_{ij}), либо равно ⊥ (в противном случае). Таким образом, входной кортеж содержит только значения переменных, с которыми ассоциировано событие. Входной кортеж может быть отправлен на вход к ФБ и вызвать шаг его выполнения. Отметим также, что поскольку события не могут приходить на вход ФБ одновременно, в кортеже только одно событие.

Тестом назовем конечную последовательность входных кортежей. Выходные воздействия не включаются в содержимое теста, поскольку они не имеют значения для определения критериев покрытия и их максимизации. Тест может определять серию шагов выполнения ФБ (по шагу на каждый входной кортеж), в промежутках между которыми ФБ сохраняет свое состояние. Также предполагается, что перед выполнением теста ФБ находится в начальном состоянии: все управляющие диаграммы находятся в своих начальных состояниях, а переменные инициализированы значениями по умолчанию. Пример теста для ФБ с интерфейсом, приведенным на рис. 2, показан в табл. 1.

Табл. 1. Пример теста с длиной, равной четырем

Номер кортежа	E_i	α_1 **(BOOL_VAR)**	α_1 **(INT_VAR)**
1	E3	true	⊥
2	E1	⊥	⊥
3	E2	false	–100
4	E2	false	42

Наконец, определим *набор тестов* как множество тестов. Смысл рассматривать множество тестов как объект оптимизации в том, что для полного покрытия программной системы одного теста может быть недостаточно.

Теперь настал момент, когда можно дать определения мерам покрытия базовых ФБ, которые будут основаны на более ранних определениях, рассмотренных в процессе обзора тестирования на основе модели.

- *Покрытие переходов* – это доля всех переходов управляющей диаграммы базового ФБ, которые были выполнены хотя бы один раз при выполнении всех тестов в наборе.

- *Покрытие n-кортежей переходов* – это доля всех выполнившихся *n*-кортежей последовательных переходов управляющей диаграммы. Например, для $n = 2$ этот критерий равен доле покрытых пар следующих друг за другом переходов.
- *Покрытие ветвей* базового ФБ определим как покрытие ветвей исходного кода, представляющего этот ФБ и полученного из него на основе некоторого детерминированного преобразования.

Что касается мер покрытия для составных ФБ, их можно вычислять на основе значений этих мер для базовых ФБ, находящихся внутри них (на произвольно глубоких уровнях вложенности). Например, можно суммировать числа покрытых элементов для каждого типа внутренних ФБ (далее предполагается именно такое определение) или делать то же самое для каждого экземпляра внутреннего ФБ. Кроме того, можно измерять чисто посещенных соединений по событиям и по данным.

На основе представленного формализма можно определить задачу, которую нужно решить в диссертации. Она заключается в разработки метода генерации наборов тестов, который максимизирует один из только что определенных критериев покрытия заданного ФБ (базового или составного). Среди этих критериев покрытие переходов часто применяется в области MBT, а также использует специфику нашей предметной области – структуру базовых ФБ, основанную на состояниях. Покрытие *n*-кортежей переходов не так популярно, но это пример более сложной меры покрытия. Наконец, покрытие ветвей широко применяется в разработке ПО и, в отличие от других рассмотренных критериев покрытия, требует посещения всех фрагментов алгоритмов внутри базовых ФБ.

1.2.3. Автоматизация тестирования на основе удовлетворения ограничений

Один из первых подходов к автоматической генерации тестов на основе удовлетворения ограничений был предложен в работе [10]. Тестовые данные в ней генерируются на основе критерия относительной, или мутационной, адекватности: тест удовлетворяет этому критерию, когда он обнаруживает ошибки в определенном числе некорректных программ. Некорректные программы, в свою очередь,

генерируются как мутации, или небольшие изменения, тестируемой программы. Для того чтобы создать тесты, обеспечивающие нахождение ошибок в программах-мутантах, генерируются и затем решаются алгебраические ограничения.

Более широкий класс методов генерации тестов, использующих идею решения задачи удовлетворения ограничений, совмещают ее с ранее упомянутым символьным выполнением или с объединением символьного и конкретного выполнения. Примером применения последней идеи является работа [25]. Идея символьного подхода заключается в том, чтобы добиться покрытия программы за счет обхода ее графа потока управления, при этом в процессе обхода графа вместо выполнения программы поддерживается набор ограничений на использующиеся в ней переменные. Так, например, посещение одной из ветвей оператора ветвления добавляет в множество ограничений либо условие этого оператора, либо его отрицание. Чтобы убедиться, что текущий путь достижим, на текущем множестве ограничений запускается сторонняя программа – CSP или SMT-решатель (солвер), которая при достижимости пути возвращает значения переменных, обеспечивающих выполнение этого пути. Поскольку среди таких переменных можно рассматривать входные переменные программы, для выполнимого пути можно сгенерировать тест (точнее, входные тестовые данные), который заставит программу выполняться по этому пути.

Существенной проблемой описанного подхода является экспоненциальный рост числа путей в графе потока управления в зависимости от размера программы. Множество эвристик (некоторые из них предложены в [25]) направлено на решение этой проблемы. Среди других недостатков символьного выполнения – сложности с поддержкой вещественных переменных и нелинейных ограничений. Обзор современных подходов к символьной генерации тестов можно найти в работе [4]. Кроме того, в последнее время появился ряд применимых на практике программных средств, основанных на этих подходах. Примеры таких программных средств – CATG (https://github.com/ksen007/janala2), Palus (http://code.google.com/p/tpalus/), CREST (https://github.com/jburnim/crest), LCT (http://www.tcs.hut.fi/Software/lime/).

1.2.4. Автоматизация тестирования на основе эволюционных вычислений

Комбинированный подход к задаче генерации тестов применяется в [13]. В этой работе описывается программное средство *EvoSuite*, которое позволяет автоматически генерировать модульные тесты для кода на языке *Java*. Генерируемые наборы тестов совместимы с библиотекой *JUnit*. Описываемый подход основан на эволюционном поиске (см. раздел 1.3) и оптимизирует покрытие наборов тестов. Другие используемые в подходе техники включают в себя гибридный поиск, динамическое символьное выполнение, рассмотренное в предыдущем подразделе, и преобразование тестируемости (testability transformation). В дополнение к этому подход способен автоматически внедрять в код тестовые оракулы – проверяющие утверждения, построенные на основе обобщения поведения программы. Эффективность этих утверждений оценивается на основе мутационного тестирования, уже упомянутого в этом разделе во время описания подхода, основанного на ограничениях [10]. Корректность сгенерированных утверждений может быть вручную проверена разработчиком.

1.3. Эволюционные вычисления

В настоящем разделе будет рассмотрена концепция эволюционных вычислений. Будут представлены несколько простых (например, метод спуска со случайными мутациями [24]) и более сложных (например, генетический алгоритм [21]) эволюционных алгоритмов. Представленные алгоритмы можно будет использовать в методах генерации наборов тестов, которые будут представлены в следующих главах диссертации.

Эволюционные и генетические алгоритмы – это общие методы оптимизации, которые применимы к различным дискретным и непрерывным задачам. Для задач, для которых их применяют, обычно не существует точных алгоритмов, работающих за полиномиальное время (в случае, если $P \neq NP$). К таким задачам, например, относится задача коммивояжера [22] и задача рабочего цеха (job shop) [9]. Эволюционные алгоритмы обычно не гарантируют нахождения оптимального

решения задачи за разумное время, а иногда не гарантируют его нахождения в принципе. Тем не менее, на практике они часто эффективны.

Основная идея эволюционных вычислений заключается в следующем. Эволюционные алгоритмы используют некоторое определенное представление возможных решений задачи (решения, представленные таким образом, называют *особями*) и обычно получают новые решения, применяя небольшие изменения к имеющимся (такие изменения называют *мутациями*) или комбинируя их между собой (эта операция известна как *кроссовер*). Функция приспособленности – мера качества, сопоставляющая особям числа на вещественной оси, – направляет эволюционный поиск в сторону лучших в ее терминах решений (в дальнейшем мы будем предполагать, что функцию приспособленности необходимо максимизировать): такие решения оставляются, в то время как худшие решения удаляются. Эта процедура называется *отбором*, или *селекцией*.

Множество конкретных техник используют эволюционные идеи. Далее мы рассмотрим основные эволюционные операторы (мутацию, кроссовер и отбор) и несколько конкретных эволюционных алгоритмов.

1.3.1. Эволюционные операторы

На протяжении настоящего обзора мы будем рассматривать две оптимизационные задачи. Первая из них, задача OneMax [29], очень проста и широко известна в литературе. В этой задаче требуется угадать определенную битовую строку длины n, на которой функция приспособленности, равная числу совпадающих битов строки и искомой строки, достигает своего максимума. Для простоты часто предполагается, что искомая строка состоит из n единиц. В этом случае функция приспособленности битовой строки равна числу единиц в ней. При этом предполагается, что эволюционный алгоритм по-прежнему не знает, какая строка является искомой. Вторая рассматриваемая задача – эта задача, поставленная в настоящем исследовании: найти набор тестов с высоким значением выбранного критерия покрытия, который используется в качестве функции приспособленности.

Идея *оператора мутации* – применить небольшое изменение к особи. Эта операция принимает на вход особь и источник случайных чисел и возвращает новую особь. Следующие мутации типичны для задачи OneMax:

- Изменить бит в случайной позиции строки.
- Для каждой позиции изменить бит в этой позиции с вероятностью $1/n$.

Что качается задачи генерации тестов, примеры возможных мутаций в этой задаче таковы:

- Выбрать случайный тест в наборе тестов, выбрать случайную позицию в этом тесте, заменить событие в этой позиции на случайное и случайным образом сгенерировать новые входные данные для этого события.
- Выбрать случайный тест в наборе тестов, выбрать случайное значение входной переменной в этом тесте и заменить его на случайно сгенерированное.
- Выбрать случайный тест в наборе тестов, выбрать случайное значение входной переменной и мутировать это значение (например, для целочисленного значения мутация может заключаться в прибавлении или вычитании небольшого числа).

Оператор кроссовера использует две особи, чтобы сгенерировать новую особь или две новых особи. Как и оператору мутации, ему нужен источник случайных чисел. Для задачи OneMax и для задач оптимизации строк в целом известны следующие операторы кроссовера:

- *Одноточечный кроссовер* выбирает случайную позицию и меняет местами суффиксы строк после этой позиции.
- *Двухточечный кроссовер* выбирает две случайные позиции и меняет местами подстроки, заключенные между этими позициями.
- *Равномерные кроссовер* независимо для каждой позиции с некоторой вероятностью (часто ½) обменивает символы, находящиеся в этой позиции.

Описанные операторы кроссовера проиллюстрированы на рис. 5, где обмениваемые части битовых строк до и после применения кроссовера выделены голубым цветом.

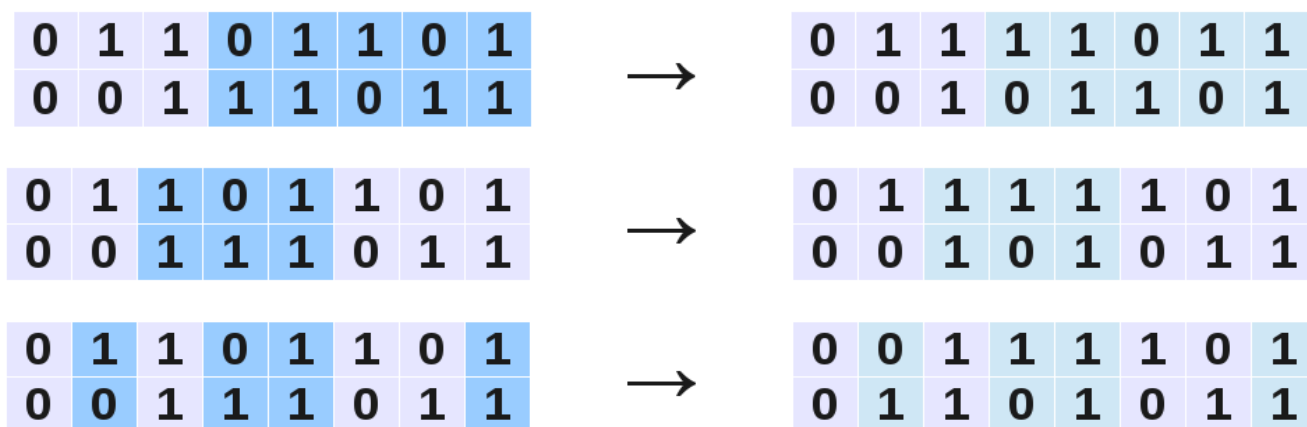

Рис. 5. Примеры применения трех операторов кроссовера: одноточечного (сверху), двухточечного (посередине) и равномерного (снизу).

Похожие идеи могут быть применены и к задаче оптимизации наборов тестов, где особью является набор тестов. Возможны следующие идеи кроссовера двух наборов тестов:

- Применить к наборам тестов один из трех строковых операторов кроссовера, считая тесты символами.
- Выбрать два случайных теста (каждый в своем наборе) и применить к ним один из трех строковых операторов кроссовера.

Оператор отбора обычно применяется в алгоритмах, оперирующих сразу с набором особей, таких как генетический алгоритм, который мы рассмотрим в дальнейшем. В каждом случае задача заключается в том, чтобы оставить определенное число особей, удалив остальные. Среди возможных операторов кроссовера есть следующие:

- Отсортировать особи в соответствии со значениями функции приспособленности и оставить заданное число лучших особей. Эта техника является самой простой.
- *Турнирный отбор*: для каждой особи, которую требуется включить в набор остающихся, случайно выбираются несколько особей (часто две) из текущего поколения, и оставляется та, у которой наибольшее значение функции приспособленности.
- *Метод «рулетки»*: особи сопоставляются частям отрезка [0, 1], длины которых пропорциональны значениям функции приспособленности особей,

после чего выбирается случайное число на отрезке. Особь, на часть отрезка которой выпало число, останется в следующем поколении. Описанный процесс показан на рис. 6.

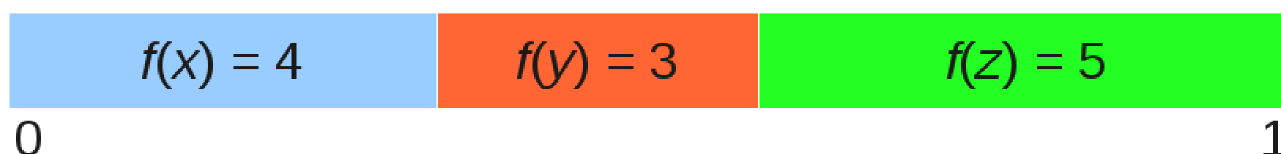

Рис. 6. Отрезок [0, 1], случайной выбор числа на котором определяет выбранную особь (x, y или z) в отборе методом «рулетки»

1.3.2. Эволюционные алгоритмы

В настоящем разделе будут представлены два эволюционных алгоритма. Мы начнем с простейшего метода спуска, после чего перейдем к более сложному генетическому алгоритму (ГА). Более подробный обзор эволюционных алгоритмов, а также метаэвристик вообще (это более широкий класс методов оптимизации) может быть найден в [2].

Метод спуска со случайными мутациями [24], также известный как *random mutation hill climber (RMHC)*, – это очень простой эволюционный алгоритм. Он хранит в памяти только одну особь – текущее решение задачи. В начале работы алгоритма случайным образом генерируется начальное решение задачи x (например, в случае задачи OneMax генерируется случайная битовая строка). После этого, пока не достигнуто условие останова, выполняются следующие действия: как результат мутации x создается новая особь y, и если $f(y) \geq f(x)$, где f – функция приспособленности, то x заменяется на y. Часто используются следующие критерии останова:

- Выполнено заданное число итераций.
- Достигнуто заданное значение функции приспособленности.
- Значение функции приспособленности не было улучшено за определенное число последних итераций (такую ситуацию называют *стагнацией*).

Генетический алгоритм (ГА) [21] – это алгоритм, который хранит множество особей в одно и то же время. Такой набор особей называют *поколением*. В начале работы алгоритма поколение заполняется случайно сгенерированными особями,

после чего на каждой итерации каждая особь подвергается кроссоверу и мутации, а затем из получившихся (и, возможно, старых) особей путем отбора собирается новое поколение того же размера. Эта схема приведена на рис. 7.

Рис. 7. Общая схема итерации генетического алгоритма

Существует множества вариаций ГА. Например, среди них есть островной ГА, в котором развиваются сразу несколько поколений на различных вычислительных узлах, и steady-state ГА, где на каждой итерации эволюционные операторы применяются только к двум особям.

Выводы по главе 1

1. В начале главе были рассмотрены особенности стандарта IEC 61499. Понятия функционального блока (ФБ) и его входного интерфейса должны оказаться важны для формального определения тестирования систем, представленных согласно стандарту IEC 61499. Внутренняя структура базовых ФБ, основанная на управляющих диаграммах, может оказаться полезна при определении критериев покрытия.
2. Далее в главе были рассмотрены различные подходы к автоматизации тестирования, включая методы на основе удовлетворения ограничений и эволюционные методы, а также различные определения критериев покрытия. На основе проведенного обзора выполнена формальная постановка задачи генерации покрывающих наборов тестов.
3. В конце главы более подробно рассмотрен эволюционный подход к оптимизации. В качестве алгоритмов решения поставленной задачи могут использоваться метод спуска со случайными мутациями и генетический алгоритм.

ГЛАВА 2. МЕТОД, ОСНОВАННЫЙ НА СТОРОННИХ ПРОГРАММНЫХ СРЕДСТВАХ

Завершив обзор литературы, мы можем перейти к разработке методов генерации наборов тестов. Один из них будет представлен в настоящей главе, а другой – в следующей. Сначала мы рассмотрим общую схему метода и обоснуем выбранный способ решения задачи, а затем поэтапно разберем метод и проанализируем его недостатки и ограничения.

2.1. Общая схема метода и обоснование принятых решений

Первый предлагаемый метод генерации наборов тестов объединяет преобразование ФБ в код на языке *Java* и эволюционный поиск набора тестов, максимизирующего покрытие полученного кода на языке *Java*. Подход использует два сторонних программных средства, *FBDK* (http://www.holobloc.com/doc/fbdk/) и *EvoSuite* [13], и поддерживает оптимизацию покрытия ветвей и покрытия переходов. Общая схема подхода приведена на рис. 8. На вход метод генерации тестов принимает XML-файл с расширением .fbt, который описывает тестируемый ФБ. Подобные файлы могут быть созданы с помощью таких средств разработки, как *FBDK* или *NxtStudio* [26]. Если ФБ является составным, то методам также должны быть доступны XML-описания всех вложенных ФБ. Метод состоит из трех этапов, описанных ниже. Первые два этапа были реализованы на языке *Java*, а третий этап – в виде скрипта на языке *bash*.

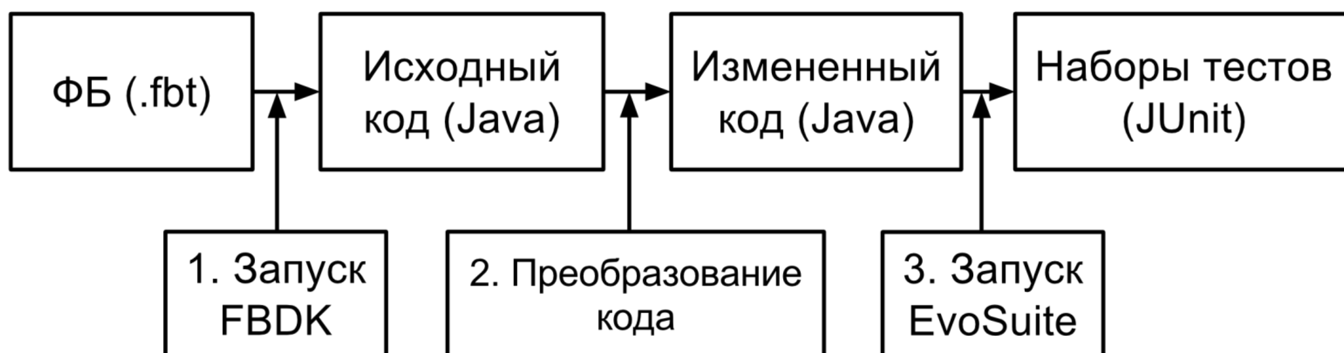

Рис. 8. Схема метода, основанного на сторонних программных средствах

Прежде чем рассматривать метод подробно, ответим на следующие вопросы, которые прояснят решения, принятые на этапе разработки общей схемы метода:

1. Почему в качестве средства поиска тестов используются эволюционные вычисления? Согласно главе 1, для решения задачи также существуют методы, основанные на удовлетворении ограничений и символьном исполнении. Есть и другие альтернативные подходы, среди которых можно было бы рассмотреть методы, связанные с формальной верификацией.
2. Почему задача решается не непосредственно для ФБ, а для программного кода на языке общего назначения, полученного на их основе?
3. Почему в качестве языка программирования для представления кода, соответствующего ФБ, выбран язык *Java*? Альтернативами являются другие императивные языки – как императивные (например, *C*), так и функциональные (например, *Haskell*).

Отвечая на первый вопрос, сначала проясним вопрос с формальной верификацией. В качестве формального средства для представления систем ФБ обычно используют модели, именуемые net condition/event systems (NCES) [34], которые, в отличие от конечных автоматов, достаточно выразительны для представления модульности. Имея такие модели, тесты для программной системы можно генерировать как решения задачи достижимости определенных элементов NCES на основе формальной верификации. Однако существенным недостатком NCES является полноценная поддержка только булевых переменных: поддержка вещественных переменных возможна только при дискретизации их значений, а реализация целочисленных переменных усложняет модель как в вычислительном смысле, так и в степени ее наглядности для человека. Кроме того, автору диссертации не известны открытые программные средства, позволяющие преобразовывать программы, представленные согласно стандарту IEC 61499, в эквивалентные им модели NCES.

Применение для решения задачи генерации тестов символьных подходов, основанных на решении задачи удовлетворения ограничений [4], является более перспективной областью. Тем не менее, недостатком этих подходов является

сложность реализации техник, которые позволяют применять их на программах реалистичного размера. Это ограничивает возможность их применения существующими открытыми программными средствами, которые их реализуют. Автором диссертации были рассмотрены несколько таких средств, включая средства CATG (https://github.com/ksen007/janala2) и LCT (http://www.tcs.hut.fi/Software/lime/). Как оказалось, эти средства могут быть применены для генерации тестов для несложных базовых ФБ, однако их использование требует больших временных затрат при большом числе входных и внутренних переменных ФБ, что типично для составных ФБ. Кроме того, существующие программные средства не всегда поддерживают нелинейные операции в программе и операции над вещественными числами, а также не уделяют внимание вопросу минимизации получающихся наборов тестов. Эволюционный подход не обладает описанными недостатками.

Далее рассмотрим вопрос использования промежуточного языка общего назначения для представления ФБ. В случае неиспользования промежуточного языка в качестве одного из шагов решения задачи пришлось бы реализовать программу, которая осуществляла бы запуск ФБ на заданном тесте и измеряла бы его покрытие. Таким образом, помимо задачи разбора синтаксиса ФБ пришлось бы решить задачу их исполнения, которая не требует решения при использовании промежуточного языка. Кроме того, для многих языков программирования общего назначения существуют библиотеки (такие как *JUnit* для языка *Java*), способные измерять покрытие кода на них тестами.

Наконец, обоснуем выбор конкретного языка общего назначения в качестве промежуточного – языка *Java*. Сравнительная простота этого языка среди императивных и его платформонезависимость привели к тому, что для него существует множество программных средств, которые могут в том числе быть применены и в процессе решения поставленной задачи – это среда выполнения функциональных блоков *FBDK*, библиотека для измерения покрытия кода *JaCoCo* (http://www.eclemma.org/jacoco/), библиотека для тестирования *JUnit*, средство генерации покрывающих наборов тестов *EvoSuite*. Если же рассматривать не

императивные языки программирования, а, например, функциональные и логические, то для таких языков число доступных программных средств существенно ниже. Кроме того, пришлось бы потратить дополнительные усилия на преобразование императивно заданной семантики выполнения ФБ и алгоритмов внутри базовых ФБ в программный код на функциональном или логическом языке.

2.2. Первый этап

Перейдем к детальному описанию предлагаемого метода генерации наборов тестов. На первом этапе работы метода стороннее программное средство *FBDK* преобразует описание тестируемого ФБ, заданное XML-файлом с расширением .fbt, в файл с исходным кодом на языке *Java*, состоящий из одного класса. Если тестируемый ФБ базовый, то описывающий его класс содержит описания состояний, событий, переменных, а также методы для обработки событий и методы, представляющие алгоритмы базового ФБ. В противном случае, если ФБ составной, то в конструкторе класса создаются экземпляры вложенных в него ФБ и соединения между ними. Описанное преобразование полностью автоматизировано и реализовано с использованием библиотеки на языке *Java*, поставляемой вместе с *FBDK*.

2.3. Второй этап

На втором этапе полученный исходный код подвергается модификации, чтобы подготовить его к эволюционной генерации тестов, которая будет далее осуществлена другим программным средством. Опишем эту модификацию. Для начала, создается новый *Java*-класс, который включает в себя класс, сгенерированный *FBDK*, как вложенный. Если тестируется составной ФБ, то в новый класс в качестве вложенных также включаются все прямые или косвенные зависимости (непосредственно и косвенно вложенные ФБ) тестируемого ФБ. Вложенные классы помечаются как приватные, чтобы предотвратить генерацию тестов, которые обращаются к ним напрямую. Далее, для каждого входного события тестируемого ФБ создается публичный метод во внешнем классе, упомянутом выше. Таким образом, только эти событийные методы оказываются доступными

снаружи внешнего класса. Каждый такой метод в качестве аргументов принимает значения переменных, с которыми ассоциировано событие, обновляет соответствующие значения внутреннего класса, сгенерированного *FBDK*, и вызывает его метод обработки события.

В дополнение к этому, для каждого перехода в каждом вложенном классе, соответствующем базовому ФБ, во внешнем классе создается пустой приватный метод-заглушка. Вызов этого метода добавляется в то место вложенного класса, где совершается соответствующий переход. Назначение таких методов-заглушек в том, чтобы оптимизировать покрытие переходов (см. следующий этап): все эти методы покрываются набором тестов тогда и только тогда, когда им покрываются все переходы. Если в качестве критерия покрытия используется покрытие ветвей, то описанные методы не генерируются.

Пример кода, сгенерированного *FBDK* и преобразованного в соответствии с описанной процедурой (включая генерацию методов-заглушек для переходов), приведен в Приложении А. Этот код реализует составной ФБ `my_sensor2` из системы PnP, которая будет описана в главе 4. Интерфейс этого ФБ и сеть ФБ внутри него приведены в начале приложения.

2.4. Третий этап

На третьем этапе модифицированный исходный код подается на вход *EvoSuite* – программному средству, которое генерирует наборы тестов для кода на языке *Java*, в качестве функции приспособленности используя покрытие ветвей. Оно реализует несколько эволюционных алгоритмов, среди которых в настоящей диссертации используется алгоритм по умолчанию – steady-state ГА. В зависимости от требуемого критерия покрытия, *EvoSuite* настраивается так, чтобы генерировать тесты покрытия для всего класса (в случае покрытия ветвей), или только для методов-заглушек, соответствующих переходам и созданных в конце предыдущего этапа (в случае покрытия переходов). Поиск осуществляется в течение фиксированного промежутка времени. Результатом работы *EvoSuite* является набор тестов, представленный в формате *JUnit*. Так как на предыдущем этапе только

событийные методы были оставлены публичными, все сгенерированные тесты являются последовательностями их вызовов со значениями переменных. Приведем пример теста из табл. 1 в виде, в котором он мог бы появиться внутри набора тестов в формате *JUnit*:

```
@Test
public void test_0() {
    ExampleFB fb = new ExampleFB();
    fb.service_E3(true);
    fb.service_E1();
    fb.service_E2(false, -100);
    fb.service_E2(false, 42);
}
```

В свою очередь, набор тестов в формате *JUnit*, состоящий только из приведенного теста, является обыкновенным *Java*-классом, включающим в себя единственный метод `test_0()`. Более сложный пример набора тестов для составного ФБ приведен в Приложении Б. Это набор состоит из двух тестов и был сгенерирован программным средством *EvoSuite*, которому на вход было передано описание ФБ из Приложения А.

2.5. Недостатки и ограничения подхода

Главное ограничение подхода заключается в малом числе поддерживаемых критериев покрытия. Он поддерживает покрытие ветвей, поскольку эту меру покрытия оптимизирует *EvoSuite*, а также покрытие переходов, которое было сведено к покрытию ветвей путем добавления в код на языке *Java* методов-заглушек для всех переходов во всех управляющих диаграммах внутри тестируемого ФБ. Применив указанный трюк, можно также добиться поддержки методом покрытия состояний автоматов. Тем не менее, существуют меры покрытия, для которых такой способ решения проблемы не подойдет.

Примером такой меры является покрытие n-кортежей переходов при $n > 1$: в этом случае объекты, которые требуется покрыть, представляют собой кортежи следующих друг за другом событий, и покрытие каждого кортежа требует, чтобы различные места кода были выполнены в нужной последовательности.

Следовательно, решение с добавление методов-заглушек в этом случае не сработает. Другие примеры критериев покрытия, не поддерживаемых описанным подходом, – это покрытие внутренности границы и покрытие путей. Возможный путь к поддержке таких мер покрытия – учитывать порядок выполнения различных элементов кода во время выполнения тестов. Однако применение *FBDK* для трансляции ФБ в код на языке *Java* не допускает такого решения проблемы. В следующей главе использование трансляции, реализованной специально для настоящей диссертации, позволит оптимизировать и такие критерии покрытия, как покрытие *n*-кортежей переходов.

Другое ограничение, проявляющееся при оптимизации покрытия ветвей, связано с наличием в коде ветвей, которые либо оказываются покрыты всегда, либо не могут быть покрыты в принципе из-за технических артефактов преобразования ФБ в код на языке *Java*. Пример фрагмента кода, который оказывается покрытым любым тестом – это конструктор класса, соответствующего ФБ. Далее, рассмотрим пример ветви в коде, сгенерированном *FBDK*, которую невозможно покрыть:

```
public void service_INIT() {
    if ((eccState == index_START)) {
        state_INIT();
        transition_OR_2();
    }
}
```

Этот метод определяет переход, который будет выполнен в случае получения ФБ события INIT. Если START – единственное состояние управляющей диаграммы, то тогда условие `eccState == index_START` выполняется всегда, в то время как покрытие ветвей требует наличия теста, где это условия приняло бы ложное значение. Из невозможности покрытия ветви, соответствующей ложному значению этого условия, не следует, что в ФБ есть ошибка, однако покрытие ветвей кода для этого ФБ, сгенерированного *FBDK*, никогда не достигнет 100 %.

Наконец, во время преобразования ФБ в код на языке *Java*, *FBDK* предполагает, что ФБ, вложенные в составной ФБ, выполняются в порядке обхода в глубину, в то время как стандарт IEC 61499 утверждает, что выполнение должно

производиться в порядке обхода в ширину. Это означает, что поведение сгенерированного *FBDK* кода, представляющего ФБ, не полностью эквивалентно поведению, заданному стандартом. Описанная проблема не встает при рассмотрении только базовых ФБ. Более того, если разработчик ФБ использует *FBDK*, то проблема не возникает и для составных ФБ, потому что поведение, видное в *FBDK* при разработке ФБ, вызвано как раз выполнением кода на языке *Java*, полученного этим программным средством из ФБ. Однако есть разработчик применяет другую среду разработки (например, *NxtStudio*), рекомендуется проверять, что генерируемые методом тесты выполняются так же, как они выполняются в среде разработки. Одна из причин, по которой существуют программные средства, реализующие стандарт IEC 61499 различным образом, – это неточность некоторых мест стандарта. Сказанное означает, что нельзя говорить об эквивалентности поведения сгенерированного кода поведению ФБ, не ссылаясь на конкретную реализацию стандарта, согласно которой ФБ должен выполняться.

Выводы по главе 2

1. В главе была обоснована общая схема построения методов генерации наборов тестов для покрытия ФБ, а также был представлен первый из таких методов. Он основан на использовании сторонних программных средств.
2. Представленный метод имеет ряд недостатков и ограничений, часть из которых возможно исправить.

ГЛАВА 3. МЕТОД, ОСНОВАННЫЙ НА СОБСТВЕННОМ ПРЕДСТАВЛЕНИИ ТЕСТОВ И ФУНКЦИОНАЛЬНЫХ БЛОКОВ

Второй подход к генерации тестов, который будет описан в настоящей главе, нацелен на преодоление некоторых ограничений подхода, основанного на сторонних программных средствах, и на его ускорение путем использования информации о задаче во время эволюционного поиска. Схема подхода приведена на рис. 9. Первый этап метода включает в себя трансляцию ФБ в код на языке *Java*, похожую на ту, что выполняется *FBDK*, однако получающийся в результате трансляции код более гибок и позволяет обрабатывать посещенные элементы ФБ (такие как переходы и состояния) во время выполнения, что расширяет множество поддерживаемых методом критериев покрытия. Второй этап метода заключается в выполнении простого эволюционного алгоритма – метода спуска, упомянутого в подразделе 1.3.2 и используемого для нахождения набора тестов с соответствии с заданным критерием покрытия. Оба этапа метода будут описаны в оставшейся части главы.

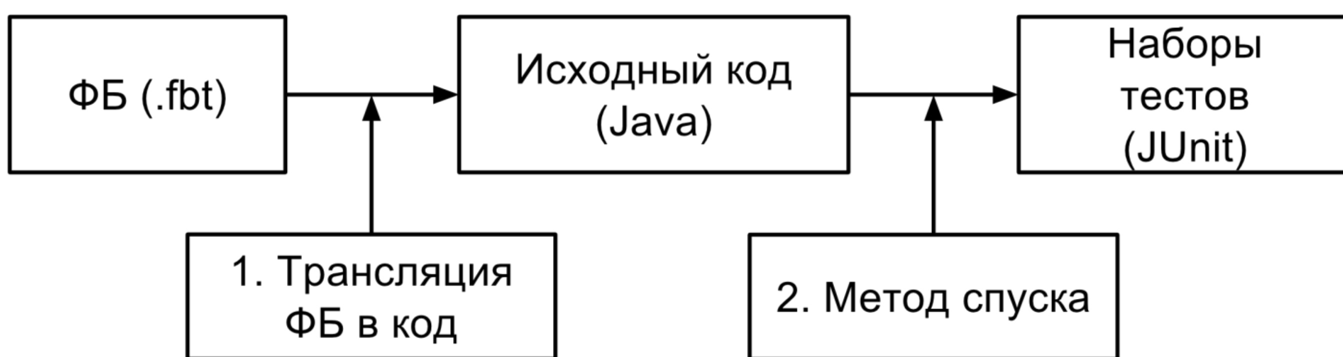

Рис. 9. Схема подхода, основанного на внутреннем представлении тестов и функциональных блоков

3.1. Трансляция функциональных блоков в код на языке Java

В этом разделе, вместо того, чтобы давать полное описание реализованного преобразование ФБ в код на языке *Java*, которое изобилует техническими деталями, мы остановимся лишь на ключевых особенностях преобразования, важных в контексте настоящей диссертации. Эти особенности приведены ниже.

- Преобразование явно учитывает порядок выполнения базовых ФБ внутри составного. Порядок выполнения внутренних составных ФБ исключается из рассмотрения: все границы составных блоков, за исключением границы тестируемого ФБ, удаляются. То есть, если ФБ fb_2 был вложен внутрь fb_1, а fb_3 был вложен в fb_2, то fb_2 будет удален, а fb_3 окажется напрямую вложен в fb_1. Если в сети присутствовало соединение, входящее в fb_2 и далее следующее к fb_3, то после удаления границ это соединение будет сразу же входить в fb_3. Похожая операция будет проведена с соединением, начинающимся в fb_3 и выходящим из fb_2.
- Возможно выполнять ФБ как в порядке обхода в ширину, так и в порядке обхода в глубину. Первая стратегия выполнения задается стандартом IEC 61499 и используется, например, в среде разработки *NxtStudio*, а вторая стратегия предполагается в *FBDK*. Далее, если есть некоторая среда разработки ФБ, реализующая стандарт по-своему (например, нарушая его в некоторых местах, как это делает средство *FBDK*), предлагаемый подход несложно модифицировать для генерации кода, представляющего поведение, характерное для упомянутой реализации стандарта.
- Для базовых ФБ преобразование генерирует класс с описаниями состояний и переменных, а также методами, представляющими алгоритмы, и методами, отвечающими за обработку событий, которые принимают как параметры значения переменных, ассоциированных с событиями. Таким образом, получающийся код визуально похож на тот, что генерируется *FBDK*. Более того, для упрощения трансляции алгоритмов было решено использовать для них код, как раз и генерируемый *FBDK*. Классы для составных ФБ включают в себя классы для всех базовых ФБ, от которых они зависят, в качестве вложенных.
- Во время выполнения каждого перехода вызывается специальный метод, который принимает информацию о совершаемом переходе (его начальное, конечно состояние и охранное условие). Этот метод, реализация которого находится за пределами автоматически генерируемого кода, вызывает

другой метод, который специфичен для выбранного критерия покрытия. Например, в случае покрытия переходов этот метод считает число уникальных выполнившихся переходов, а в случае покрытия ветвей этот метод не делает ничего, потому что этот критерий покрытия зависит только от покрытых участков кода.

3.2. Реализация эволюционного алгоритма

Если множество вариантов эволюционных алгоритмов, которые можно реализовать для решения задачи. Число имеющихся вариантов сильно превосходит число алгоритмов, рассмотренных в подразделе 1.3.2. В итоге выбор пал на метод спуска со случайными мутациями, который отличается своей простотой. Возможности реализации алгоритмов, которые используют в одно и то же время больше одной особи (например, генетический алгоритм) не рассматривались, поскольку из набора решений поставленной задачи всегда можно сконструировать новое, которое будет не хуже их всех. Чтобы это сделать, достаточно объединить все тесты из всех наборов в один большой набор тестов: это обеспечит покрытие полученным набором тестов всех фрагментов системы, которые были покрыты изначальными наборами тестов. Полученный таким способом набор тестов разумно минимизировать (например, жадно удаляя тесты и входные кортежи, пока это не приводит к уменьшению покрытия): небольшие наборы тестов обычно предпочтительны, поскольку они требуют меньше времени для запуска и более просты для понимания человеком. Таким образом, для выбранной задачи достаточно одной особи в поколении.

Эволюция начинается с единственного теста, состоящего из одного случайным образом сгенерированного входного кортежа, и заканчивается в случае, если за последние 1000 итераций функция приспособленности не была улучшена. Ниже приведены описания выбранных функций приспособленности и реализации оператора мутации.

3.2.1. Функции приспособленности

Используемые функции приспособленности прямо соответствуют выбранным критериям покрытия: покрытию ветвей, переходов и 2-кортежей переходов. Они также учитывают размер наборов тестов: если у двух наборов тестов одинаковое значение критерия покрытия, то среди них лучше набор тестов меньшего размера.

Значение функции приспособленности, основанной на покрытии ветвей, вычисляется при помощи библиотеки *JaCoCo* (http://www.eclemma.org/jacoco/) для покрытия кода на языке *Java*. Перед каждым вычислением функции приспособленности она обрабатывает .class-файл, соответствующий тестируемому ФБ, чтобы вставить в него информацию, используемую для определения покрытых ветвей. В отличие от покрытия ветвей, вычисление покрытия переходов и 2-кортежей переходов существенно проще: информация об уникальных выполнившихся переходах или парах переходов поддерживается во время выполнения тестов (см. конец раздела 3.1).

Наконец, каждый набор тестов выполняется с использованием механизма reflection, присутствующего в языке *Java* и позволяющего, в числе прочего, вызывать методы, зная их сигнатуры только во время выполнения программы. Альтернативный, но существенно более медленный подход заключается в компиляции каждого набора тестов при каждом вычислении функции приспособленности.

3.2.2. Оператор мутации

Оператор мутации имеет три параметра: p_{rem} – вероятность уменьшить набор тестов, p_{adj} – вероятность изменить набор тестов путем добавления или модификации входных кортежей, и N_{op} – максимальное число операций, которое можно выполнить в процессе одной мутации. Последний параметр управляет силой мутации. В настоящем исследовании были использованы значения $p_{\mathrm{rem}} = 0{,}3$, $p_{\mathrm{adj}} = 0{,}4$ и $N_{\mathrm{op}} = 3$. Детальное описание оператора мутации набора тестов приведено ниже.

- **Мутация удаления.** С вероятностью p_{rem} либо случайный тест, либо случайный входной кортеж в случайном тесте (оба варианта с

вероятностью ½) удаляется из набора тестов. В этом случае новая особь будет принята методом спуска, если покрытие не уменьшилось в результате мутации. Эта мутация нацелена на уменьшение размера набора тестов, который обычно увеличивается в процессе эволюции.

- **Мутация копирования.** Если не была применена мутация удаления, то с вероятностью p_{adj} происходит мутация копирования: выбирается и копируется случайный тест, после чего в нем делается случайное число изменений, число которых равномерно распределено между 1 и N_{op}. Каждое изменение заключается либо в замещении случайного выбранного кортежа на сгенерированный случайным образом, либо во вставке случайно сгенерированного входного кортежа в случайно выбранную позицию в тесте (каждый вариант с вероятностью ½). Для этой мутации существенно, что тест копируется перед модификацией, поскольку это означает, что новый набор тестов не может иметь меньшее покрытие, чем старый. Однако недостатком мутаций копирования является быстрое увеличение размера набора тестов, которое должно быть компенсировано мутациями удаления.
- **Мутация создания.** Если не были применены другие мутации, то применяется мутация создания: генерируется новый тест и добавляется к текущему набору тестов. Длина нового теста равномерно распределена между 1 и N_{op}.

Выводы по главе 3

1. В главе был представлен метод генерации наборов тестов, основанный на использовании внутреннего представления тестов и функциональных блоков.
2. Представленный метод исправляет некоторые недостатки метода, приведенного в предыдущей главе.

ГЛАВА 4. ИССЛЕДОВАНИЕ ПРЕДЛОЖЕННЫХ МЕТОДОВ

В настоящей главе описываются проведенное на двух наборах ФБ экспериментальное исследование предложенных подходов к генерации наборов тестов, а также его результаты. Первая цель экспериментального исследования – оценить предложенные методы в терминах значений критериев покрытия, времени работы и размера получаемых наборов тестов. Дополнительная его цель – сравнить оба подхода. В конце главы приводится методологическое сравнение предлагаемого подхода и подходов MBT.

4.1. Тестируемые системы

В качестве тестируемых систем используются две программные системы, предназначенные для управления двумя несложными лабораторными установками. Первая система, или, более точно, набор похожих систем, представляет собой программу управления pick-and-place манипулятором (PnP), ранее использованным в [27] для исследования подхода к решению другой проблемы. Эта система состоит их 31 базового и 17 составных ФБ, реализованных в *FBDK*.

Одна из аппаратных реализаций этого устройства приведена на рис. 10 в виде снимка экрана, сделанного в *FBDK* (это программное средство может использоваться не только для разработки ФБ, но и для моделирования взаимодействия управляющей системы с объектом управления). На снимке экрана показаны два горизонтальных цилиндра и один вертикальный. Цилиндры соединены друг с другом. Система цилиндров должна подбирать объекты с трех платформ и помещать их в корзину слева. Другой снимок экрана из *FBDK*, приведенный в рис. 11, изображает сеть внутренних ФБ составного ФБ `PnpCylinders`, который моделирует связи между цилиндрами.

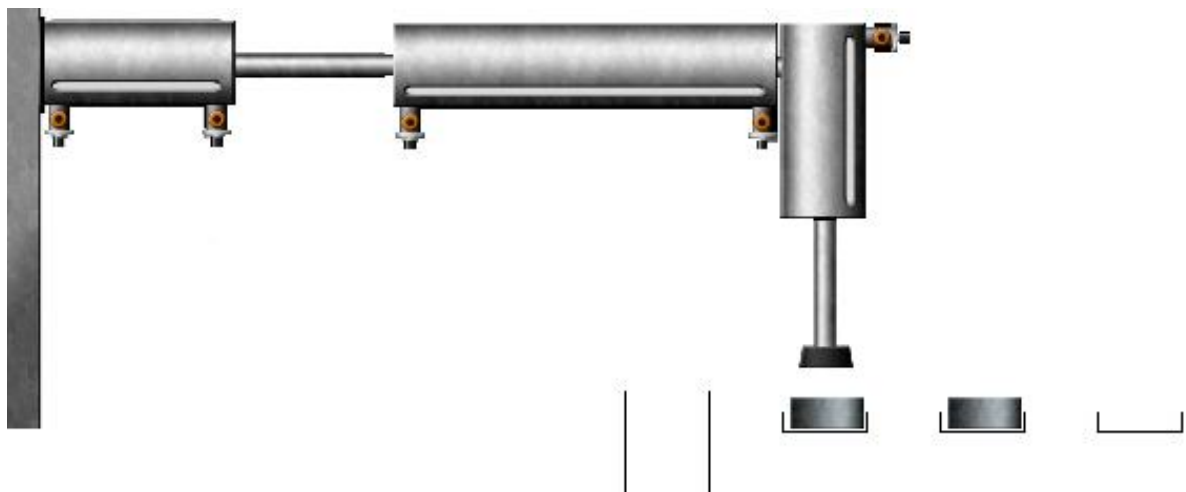

Рис. 10. Схема одной из реализаций pick-and-place манипулятора

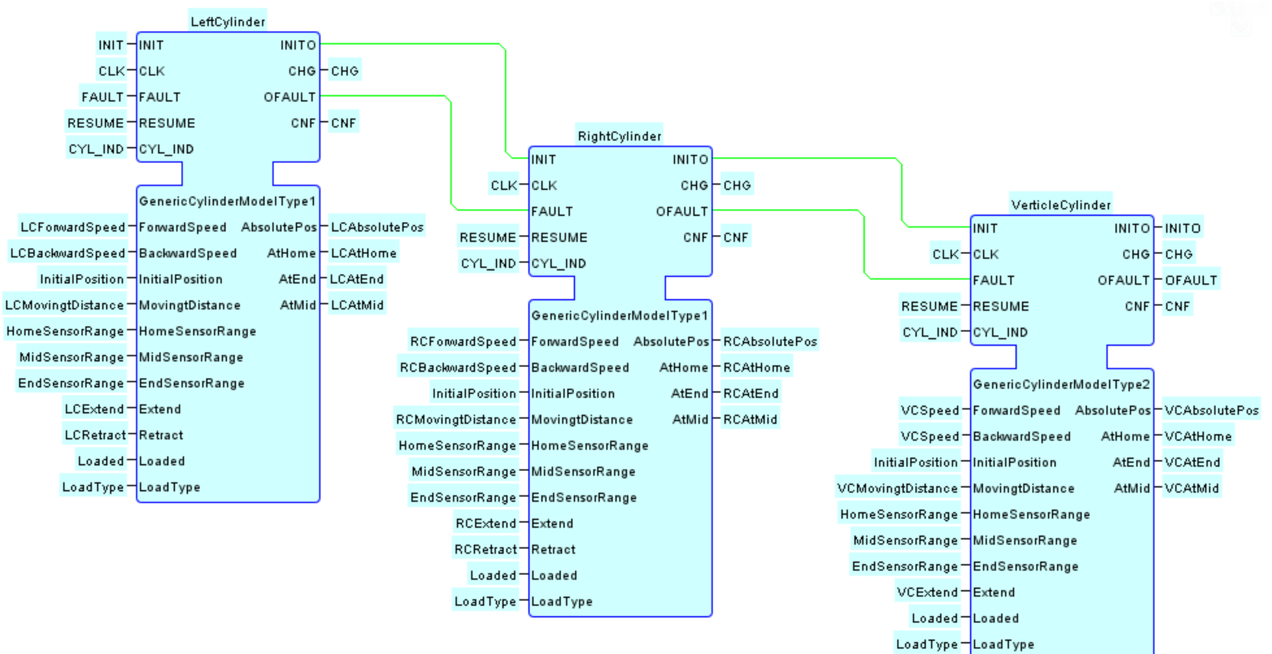

Рис. 11. Сеть ФБ, соответствующая модели из трех связанных друг с другом цилиндров, приведенных на рис. 10

Вторая программная система представляет собой управляющее приложение для генератора тепла (heat production plant, HPP), показанного на рис. 12. В работе [28] упоминается управляющая система для этого объекта управления, представленная в стандарте IEC 61131, а система, которую мы будем использовать, является результатом повторной разработки этой системы для соответствия стандарту IEC 61499. Для повторной разработки использовалось программное средство *NxtStudio* [26]. ФБ, разработанные в этой системе, после небольших

автоматизируемых изменений могут обрабатываться и *FBDK*. Однако получившаяся система оказалась не такой модульной, как система PnP: в ней всего один составной ФБ, представляющий собой все приложение. Двенадцать других ФБ являются базовыми.

Рис. 12. Генератор тепла

Число входных событий среди ФБ из только что описанных систем варьируется от 1 до 7 (медиана равна двум). Число входных переменных среди этих ФБ в целом выше: оно варьируется от 0 до 34 (медиана равна шести). У базовых ФБ имеются от 2 до 15 состояний и от 2 до 21 перехода (медианы – 3 и 4 соответственно). Наконец, размер ФБ, измеренный в строках кода на языке *Java*, составляет от 92 до 4725 (медиана равна 320). Большие файлы с исходным кодом (то есть около 1000 строк кода и больше) характерны для составных ФБ, поскольку код для них включает код всех их зависимостей.

4.2. Описание экспериментов

Подход на основе сторонних программных средств (описанный в главе 2, далее будет именоваться как Подход 1) и подход, основанный на внутреннем представлении тестов и ФБ (описанный в главе 3, далее будет называться Подходом 2), были исследованы отдельно. Все вычисления проводились на персональном компьютере с процессором *Intel Core i*7-2670*QM* (2.2 ГГц). Для каждого ФБ и каждого рассмотренного критерия покрытия был проведен один запуск каждого из подходов. Покрытие ветвей и переходов рассматривались для обоих методом, а дополнительные эксперименты с покрытием 2-кортежей переходов были проведены только для Подхода 2, поскольку Подход 1 не поддерживает этот критерий покрытия.

Для различных подходов были использованы различные критерии останова. Подходу 1 для нахождения решения задачи всегда давалось фиксированное время, что связано с особенностями работы *EvoSuite*. Отметим, что время выполнения преобразования ФБ в код и время модификации этого кода здесь не учитываются в силу его незначительности (меньше секунды). Десять минут давались *EvoSuite* для генерации тестов для базовых ФБ, и двадцать минут – для составных. Время, отпущенное для генерации набора тестов для составных ФБ, было больше, поскольку код, сгенерированный для них, также был больше. Что касается Подхода 2, он был реализован так, чтобы была возможность использовать более разумный критерий, основанный на стагнации: запуск метод спуска со случайными мутациями останавливался, если значение функции приспособленности не увеличивалось в течение 1000 итераций. Оба подхода также могли остановиться преждевременно, если они добились полного покрытия.

После запусков каждый набор тестов был подвергнут минимизации. Для Подхода 1 эта операция выполнялась встроенной возможностью *EvoSuite*, а для Подхода 2 применялась жадная процедура: входные кортежи удалялись из набора тестов до тех пор, пока ни один из них не мог быть удален без уменьшения покрытия набора тестов.

4.3. Результаты

Результаты проведенных экспериментов приведены в табл. 2–5, где для всех групп экспериментов показаны основные статистические данные. Статистики значений критериев покрытия для обоих подходов представлены в табл. 2 и 3:

Табл. 2. Статистики значений критериев покрытия для Подхода 1

Тип ФБ, критерий покрытия	Минимум	Первый квартиль	Медиана	Третий квартиль	Максимум
Базовые, покрытие ветвей	54,1%	86,4%	91,7%	94,4%	98,7%
Составные, покрытие ветвей	32,0%	77,0%	83,0%	89,7%	94,3%
Базовые, покрытие переходов	55,6%	100,0%	100,0%	100,0%	100,0%
Составные, покрытие переходов	5,7%	92,0%	100,0%	100,0%	100,0%

Табл. 3. Статистики значений критериев покрытия для Подхода 2

Тип ФБ, критерий покрытия	Минимум	Первый квартиль	Медиана	Третий квартиль	Максимум
Базовые, покрытие ветвей	42,9%	71,4%	82,7%	93,0%	98,8%
Составные, покрытие ветвей	7,7%	56,6%	66,7%	86,6%	91,2%
Базовые, покрытие переходов	55,6%	100,0%	100,0%	100,0%	100,0%
Составные, покрытие переходов	5,7%	70,0%	92,2%	100,0%	100,0%
Базовые, покрытие 2-кортежей переходов	41,2%	100,0%	100,0%	100,0%	100,0%
Составные, покрытие 2-кортежей переходов	0,4%	64,6%	86,4%	99,7%	100,0%

Статистика времени выполнения представлена только для Подхода 2, поскольку для Подхода 1 это время было задано заранее (10 минут для базовых ФБ, 20 минут для составных ФБ). Она содержится в табл. 4:

Табл. 4. Статистики времени работы Подхода 2 (время приведено в секундах)

Тип ФБ, критерий покрытия	Минимум	Первый квартиль	Медиана	Третий квартиль	Максимум

Базовые, покрытие ветвей	2,4	3,3	4,8	7,7	17,4
Составные, покрытие ветвей	3,2	18,2	38,0	65,0	298,9
Базовые, покрытие переходов	0,0	0,0	0,1	0,3	2,5
Составные, покрытие переходов	0,0	1,2	5,4	15,2	160,6
Базовые, покрытие 2-кортежей переходов	0,0	0,1	0,2	1,1	29,4
Составные, покрытие 2-кортежей переходов	0,0	3,9	37,8	185,4	1646,9

В дополнение к приведенным данным были изучены размеры сгенерированных наборов тестов для обоих подходов. Статистики размера представлены в табл. 5:

Табл. 5. Статистики размера наборов тестов для обоих подходов (размер приведен в числе входных кортежей)

Тип ФБ, подход	Минимум	Первый квартиль	Медиана	Третий квартиль	Максимум
Базовые, Подход 1	1	4	12	27	77
Базовые, Подход 2	1	4	11	17,5	52
Составные, Подход 1	1	8	32	64	95
Составные, Подход 2	1	5	39	47,75	89

4.3.1. Обзор результатов

Начнем анализ результатов с обзора данных, приведенных в таблицах. Во-первых, значения покрытия оказались лучше для базовых ФБ независимо от критериев покрытия, что может быть объяснено тем, что они меньше составных, а также тем, что полное покрытие не является требованием к составным ФБ, если они не представляют собой сами программные системы.

Далее, покрытие переходов в целом было более простым для достижения, чем покрытие ветвей. Полное (100 %) покрытие было достигнуто Подходом 1 более чем для 75 % базовых ФБ (для 42 из 43) и более чем для 50 % (11 из 18) составных ФБ. Такой результат может быть объяснен тем, что достижение покрытия переходов –

более простая цель, поскольку оно не требует посещения всевозможных путей выполнения алгоритмов внутри управляющих диаграмм. Что касается Подхода 2, его результаты хуже, однако он способен оптимизировать покрытие 2-кортежей переходов, в отличие от Подхода 1. Значения покрытия 2-кортежей переходов оказались близки к значениям обыкновенного покрытия переходов, достигнутых тем же Подходом 2.

Значения покрытия ветвей были в целом ниже соответствующих значений покрытия переходов, и опять же, они оказались лучше у Подхода 1. Однако наличие всегда покрываемых и недостижимых ветвей, отмеченное в разделе 2.4, не позволяет полноценно сравнить значения покрытия ветвей для двух подходов. Поэтому сгенерированные наборы тестов были исследованы вручную, и превосходство Подхода 1 в терминах покрытия ветвей было подтверждено. Более подробно результаты ручного исследования наборов тестов описаны в следующем подразделе.

Как видно, Подход 1 превосходит Подход 2 по значениям критериев качества. Были проведены несколько попыток улучшить Подход 2 путем настройки его параметров p_{rem}, p_{adj} и N_{op} программным средством *irace* [23], а также проведением двух запусков подхода и слиянием их результатов, но эти изменения почти не повлияли на его производительность. Попытка применить покрытие ветвей в качестве вспомогательного критерия оптимизации для покрытия переходов (то есть, если у двух наборов тестов одинаковое покрытие переходов, оставлять из них тот, у кого больше покрытие ветвей), тоже не увенчалась успехом. Тем не менее, как видно из табл. 4, Подход 2 существенно быстрее: он работает меньше минуты в более чем половине случаев, в то время как Подходу 1 нужно 10 или 20 минут в зависимости от разновидности тестируемого ФБ. Кроме того, Подход 2 поддерживает больше критериев покрытия.

В заключение сравним размеры наборов тестов, получаемыми двумя методами. Эти размеры приведены в табл. 5. Медианные размеры наборов тестов, полученных различными подходами, очень близки друг к другу, однако третий квартиль размера оказался больше для Подхода 1. Впрочем, это не удивительно,

поскольку Подход 1 обеспечивает лучшее покрытие, а для его достижения могут потребоваться наборы тестов большего размера

4.3.2. Детальное изучение сгенерированных наборов тестов

После того, как методами были сгенерированы все наборы тестов, они были запущены в среде разработки the *Eclipse IDE* (https://eclipse.org/) с использованием плагина *EclEmma* (http://www.eclemma.org/), который встраивает в *Eclipse IDE* поддержку *JUnit*. Непокрытые фрагменты ФБ были изучены, и на основе этого изучения были сделаны несколько предварительных заключений, которые приведены ниже.

- Если при оценке покрытия не учитывать недостижимые ветви, то тогда 18 из 43 базовых ФБ и 4 из 18 составных ФБ оказываются полностью покрыты наборами тестов, сгенерированными Подходом 1. Для Подхода 2 ситуация похожая: покрытыми полностью оказываются 16 базовых и 4 составных.
- В *EvoSuite* и *JaCoCo* покрытие ветвей подразумевает покрытие всех комбинаций значений условий внутри решений условных операторов. Если это определение ослабить до покрытия ветвей ‘then’ и ‘else’, соответствующих решению целиком, то тогда еще 6 базовых и 1 составной ФБ оказываются полностью покрытыми.
- Некоторые базовые ФБ, особенно ФБ из системы PnP, содержали в себе алгоритмы, не привязанные ни к одному состоянию и поэтому недостижимые. Эту особенность можно считать ошибкой, однако обнаружить такие алгоритмы не составляет труда при помощи наивного статического анализа кода, который, например, осуществляется *Eclipse IDE*.

Далее более детально были изучены наборы тестов, сгенерированные Подходом 1. Поскольку эволюционный поиск не гарантирует оптимальности решения задачи, также была осуществлена попытка покрыть непокрытые фрагменты ФБ вручную. Пробелы в покрытии ветвей двух базовых ФБ были заполнены тестами из соответствующих наборов для покрытия переходов. Также

удалось незначительно изменить один из сгенерированных наборов тестов, чтобы улучшить покрытие ветвей другого базового ФБ.

Более того, удалось найти недостижимые ветви кода, которые свидетельствуют об ошибках при разработке ФБ. В одном из базовых ФБ из системы НРР были обнаружены несколько недостижимых состояний. Причина их недостижимости заключалась в ошибке при написании составного условия (`AI.value < PRESET_H.value & AI.value >= PRESET_H.value`), которое, очевидно, является невыполнимым. Эта ситуация показана на рис. 13, где непокрытый строки кода обозначены красным. Другие цвета соответствуют частично покрытому коду (желтый) и полностью покрытому коду (зеленый). Раскраска была осуществлена плагином *EclEmma*.

```
5809        public void alg_REQ() {
5810            ;
5811            if (((DIH.value & DIL.value)
5812                    | (AI.value >= PRESET_HH.value & !DIH.value)
5813                    | (AI.value <= PRESET_LL.value & !DIL.value)
5814                    | (AI.value > PRESET_MAXAI.value) | (AI.value <
5815                SensorFault.value = true;
5816                AlarmHH.value = false;
5817                AlarmH.value = false;
5818                AlarmL.value = false;
5819                AlarmLL.value = false;
5820            } else {
5821                SensorFault.value = false;
5822                if (AI.value >= PRESET_HH.value | DIH.value) {
5823                    AlarmHH.value = true;
5824                    AlarmH.value = false;
5825                    AlarmL.value = false;
5826                    AlarmLL.value = false;
5827                } else if (AI.value < PRESET_H.value
5828                        & AI.value >= PRESET_H.value) {
5829                    AlarmHH.value = false;
5830                    AlarmH.value = true;
5831                    AlarmL.value = false;
5832                    AlarmLL.value = false;
5833                } else if (AI.value <= PRESET_L.value
```

Рис. 13. Пример кода, не достижимого из-за ошибки в условии

Кроме того, удалось объяснить низкие значения покрытия двух составных ФБ. Первый составной ФБ принадлежал системе PnP и имел значения покрытия ветвей и переходов 32,1 % и 5,7 % соответственно. У этого ФБ отсутствовали некоторые соединения по событиям между элементами входной части его интерфейса и внутренними ФБ. Другие части второго ФБ – единственного составного ФБ в

системе НРР, значения обоих критериев покрытия которого были равны 64,4 %, – были недостижимы в связи с фиксированными значениями некоторых входных переменных для вложенных ФБ, а также из-за того, что этот ФБ включал в себя базовые ФБ с недостижимыми частями.

Все ошибки в тестируемых системах, найденные Подходом 1, можно было также обнаружить при помощи Подхода 2, однако использование наборов тестов, сгенерированных Подходом 2, обычно требует большего количества ручной работы, поскольку этот подход оставляет больше непокрытых сегментов кода, которые на самом деле достижимы, что демонстрируется Подходом 1.

4.4. Сравнение предложенных подходов с тестированием на основе модели

Ранее, в разделе 1.2, упоминалась популярная методология автоматизации тестирования – тестирование на основе модели (МВТ). Эту методологию используют в том числе для тестирования ПО для систем промышленной автоматизации, поэтому осмысленно сравнить с ней предложенные в диссертации подходы. Различие предлагаемых методов и МВТ по используемым в процессе тестирования артефактам и по свойствам создаваемых наборов тестов затрудняет их численное сравнение. Вместо этого мы покажем, что предлагаемые методы являются альтернативой МВТ и имеют по сравнению с МВТ как достоинства, так и недостатки.

Начнем сравнение с более подробного рассмотрения процесса применения МВТ. Поскольку МВТ использует для генерации тестов абстрактную модель, независимую от реализации программы (то есть речь идет о тестировании «черного ящика»), генерируемые тесты также являются абстрактными. Следовательно, помимо ручного создания модели спецификации для применения МВТ нужно также создать программный компонент, трансформирующий абстрактные тесты в исполняемые тестовые скрипты, либо адаптирующий интерфейс реализации к интерфейсу модели. Тесты, получающиеся в результате применения МВТ,

обеспечивают покрытие модели спецификации, а значит – покрытие требований к программной системе. Схема применения MBT представлена на рис. 14.

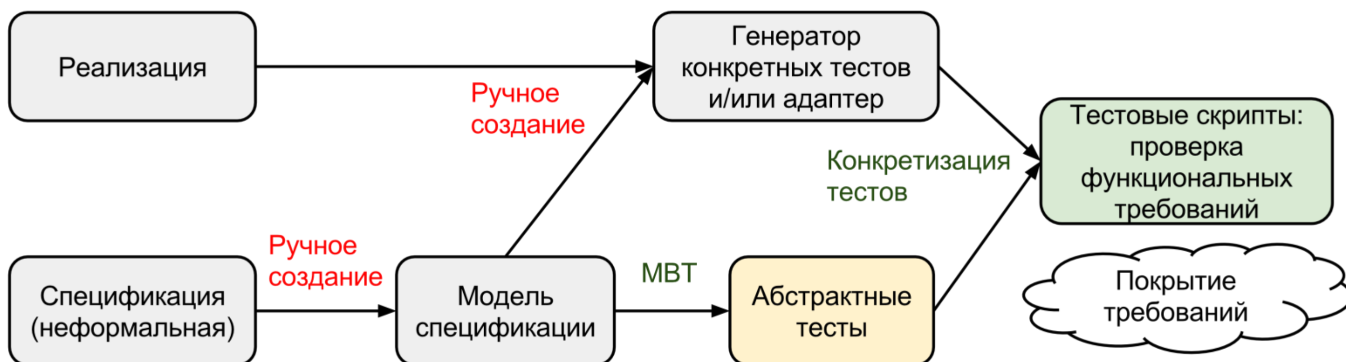

Рис. 14. Схема применения MBT

Теперь перейдем к применению предлагаемых подходов. Непосредственное применение эволюционной генерации тестов позволяет получить из реализации ФБ тесты, состоящие только из входных данных (без проверок выходных воздействий на корректность, или *оракулов*), однако этот процесс полностью автоматизирован и не требует ни создания модели спецификации, ни написания генератора конкретных тестов из абстрактных. Полученные входные тестовые данные можно использовать для обнаружения недостижимых частей системы, что позволяет идентифицировать в ней ошибки. Однако чтобы сгенерированные тесты оказались способны проверять функциональные требования к системе, их нужно вручную на основе спецификации дополнить выходными воздействиями. Схема описанной цепочки действий приведена на рис. 15.

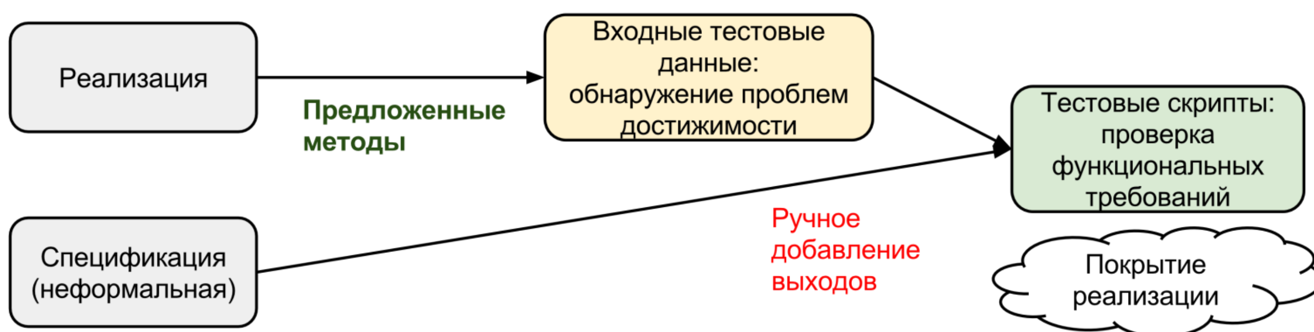

Рис. 15. Схема возможных применений предлагаемого подхода

Мы видим, что и эволюционная генерация тестов, и MBT позволяют частично автоматизировать процесс создания полноценных исполняемых тестовых скриптов,

при этом каждый из подходов имеет части, которые необходимо выполнить вручную. Критерии, на основе которых оба подхода создают тесты, не идентичны. С одной стороны, из полного покрытия тестами реализации ФБ не следует покрытие его требований, поскольку в некорректной реализации некоторые требования могут быть не отражены. С другой стороны, из полного покрытия модели требований не следует покрытие реализации, поскольку эта модель часто гораздо менее подробна, чем реализация. Далее, эволюционная генерация тестов, в отличие от MBT, полностью автоматизирует процесс получения входных тестовых данных.

Таким образом, обе методологии обладают достоинствами и недостатками, при этом ни одна из них не превосходит другую по всем критериям. Сравнение методологий обобщено в табл. 6. Возможным также является совместное применение методологий – это позволило бы получать наборы тестов, покрывающих как требования к ФБ, так и его реализацию.

Табл. 6. Сравнение эволюционного подхода к генерации тестов и MBT

Критерий	**Эволюционная генерация тестов**	**MBT**
Разновидность тестирования	Тестирование «белого ящика»	Тестирование «черного ящика»
Тип покрытия тестов	Покрытие реализации	Покрытие требований
Полностью автоматизированное получение входных тестовых данных	Да	Нет
Возможность получения полноценных тестов	При ручном добавлении оракулов	Непосредственная (при достаточной детальности модели)
Полностью автоматизированное получение полноценных тестов	Нет	Нет
Работа, выполняемая вручную	Добавление тестовых оракулов	Создание модели и решение задачи конкретизации тестов

Выводы по главе 4

1. В главе было проведено экспериментальное исследование и сравнение подходов, предложенных в предыдущих двух главах.

2. Первый из предложенных подходов генерирует наборы тестов с большим покрытием, но второй подход более быстрый и гибкий.
3. Предложенные методы удалось применить на практике: с их помощью были обнаружены несколько ошибок в тестируемых системах.
4. Было проведено методологическое сравнение предложенных подходов и тестирования на основе модели, которое показало, что предлагаемое в диссертации решение является альтернативной существующему.

ЗАКЛЮЧЕНИЕ

В начале диссертации были рассмотрены несколько областей, существенных для разработки метода генерации покрывающих наборов тестов для программных систем, представленных в стандарте IEC 61499. Полученные знания позволили формально поставить задачу, а затем разработать методы, ее решающие.

Далее в диссертации были представлены два метода генерации входных тестовых данных для ФБ, представленных в стандарте IEC 61499. Методы оптимизируют покрытие заданных ФБ генерируемыми наборами тестов. Первый метод основан на применении сторонних программных средств, в то время как второй метод реализован независимым от них. Полученные результаты и их детальное рассмотрение свидетельствуют о том, что разработанные методы применимы на практике. В частности, они помогли обнаружить несколько ошибок в тестируемых системах, которые делали некоторые их части недостижимыми. Методы, однако, отличаются по качеству создаваемых наборов тестов (лучшим оказался первый), по времени их работы (лучшим оказался второй) и по множеству поддерживаемых критериев покрытия (лучшим снова оказался второй). Таким образом, выбор метода при необходимости применить один из них должен зависеть от того, какой параметр важнее. Превосходство метода, использующего сторонние программные средства, в терминах значений критериев покрытия, может быть связано с тем, что *EvoSuite* использует для генерации тестов не только эволюционные вычисления, но и другие техники, упомянутые в подразделе 1.2.2.

Проведенное исследование имеет несколько ограничений, которые могут быть преодолены в рамках дальнейшей научной работы. Первое его ограничение связано с природой эволюционных алгоритмов, которые не всегда генерируют точное решение задачи. Возможный способ преодоления этого ограничения – замена эволюционного поиска на один из методов, основанных на решении ограничений и символьном исполнении [4], однако эта замена может повлечь за собой существенное увеличение времени работы метода в некоторых случаях. Далее, в диссертации не шла речь о выходных данных в тесте, которые могут быть получены на основе достаточно детальной формальной модели требований к приложению.

Наконец, тестируемые системы, на которых исследовались предложенные методы, не в полной степени отражают сложность ПО для систем промышленной автоматизации.

Еще одно замечание касается связи предложенных подходов и методологии MBT. В рамках MBT генерация тестов основана на формальных моделях требований к ПО, которые подготавливаются независимо от реализации системы. Напротив, предлагаемые методы используют только реализацию. Таким образом, они не попадают в рамки методологии MBT, хотя эта область и оказалось очень полезна определениями критериев покрытия.

По теме настоящей магистерской диссертации сделан доклад на IV Всероссийском конгрессе молодых ученых (НИУ ИТМО, 2015), а также принят доклад на конференцию INDIN 2015 IEEE International Conference on Industrial Informatics.

СПИСОК ЛИТЕРАТУРЫ

1. *Binder R.* Testing object-oriented systems: models, patterns, and tools. 2000. Addison-Wesley Professional.
2. *Boussaïd I., Lepagnot J., Siarry P.* A survey on optimization metaheuristics. Information Sciences, 2013, vol. 237, pp. 82–117. Elsevier.
3. *Broy M., Jonsson B., Katoen J.-P., Leucker M., Pretschner A. (eds.).* Model-Based Testing of Reactive Systems; Advanced Lectures. Lecture Notes in Computer Science, 2005, vol. 3472. Berlin-Heidelberg: Springer Verlag.
4. *Cadar C., Sen K.* Symbolic execution for software testing: three decades later. Communications of the ACM, 2013, vol. 56, no. 2, pp. 82–90. ACM.
5. *Clarke E.M., Grumberg O., Peled D.* Model checking. 1999. Cambridge: MIT press.
6. *Colla M., Brusaferri A., Carpanzano E.* Applying the IEC-61499 model to the shoe manufacturing sector. 11th IEEE Conference on Emerging Technologies and Factory Automation, ETFA'06, 2006, pp. 1301–1308. IEEE.
7. *Cormen T.H., Leiserson C.E., Rivest R.L, Stein C.* Introduction to algorithms. Vol. 2. 2001. Cambridge: MIT press.
8. *Creswell J.* Review of the Literature, Chapter 2 of Research Design: Qualitative, Quantitative, and Mixed Method Approaches. 2007. Thousand Oaks: Sage Publications.
9. *Della Croce F., Tadei R., Volta G.* A genetic algorithm for the job shop problem. Computers & Operations Research, 1995, vol. 22, no. 1, pp. 15–24. Elsevier.
10. *DeMilli R.A., Offutt A.J.* Constraint-based automatic test data generation. IEEE Transactions on Software Engineering, 1991, vol. 17, no. 9, pp. 900–910.
11. *Edvardsson J.* A survey on automatic test data generation. 2nd Conference on Computer Science and Engineering, 1999, pp. 21–28. Linkoping. ECSEL.
12. *Enoiu E.P., Sundmark D., Pettersson P.* Model-based test suite generation for function block diagrams using the UPPAAL model checker. 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2013, pp. 158–167. IEEE.

13. *Fraser G., Arcuri A*. Evosuite: automatic test suite generation for object-oriented software. Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 2011. New York, NY, USA, pp. 416–419. ACM.
14. *Gutjahr W*. Partition testing versus random testing: The influence of uncertainty. IEEE Transactions on Software Engineering, 1999, vol. 25, no. 5, pp. 661–674. IEEE.
15. *Hall K.H., Staron R.J., Zoitl A*. Challenges to industry adoption of the IEC 61499 standard on event-based function blocks. 5th IEEE International Conference on Industrial Informatics (INDIN'07), 2007, Vienna, Austria, vol. 2, pp. 823–828. IEEE.
16. *Harel D., Politi M*. Modeling reactive systems with statecharts: the STATEMATE approach. 1998. New York, NY, USA. McGraw-Hill, Inc.
17. *Harman M*. Software Engineering Meets Evolutionary Computation. Computer, 2011, vol. 44, no. 11, pp. 31–39. IEEE.
18. *Helke S., Neustupny T., Santen T*. Automating test case generation from Z specifications with Isabelle. Jonathan P. Bowen, Michael G. Hinchey, and David Till (eds.), Proceedings of the 10th International Conference of Z Users: The Z Formal Specification Notation (ZUM 1997). Lecture Notes in Computer Science, 1997, vol. 1212, pp. 52–71. Springer Berlin Heidelberg.
19. IEEE. P29119-1-FDIS, Apr 2013 – IEEE Approved Draft Standard for Software and Systems Engineering – Software Testing – Part 1: Concepts and Definitions. 2013. IEEE.
20. International Electrotechnical Commission. IEC 61499-1: Function blocks – Part 1: Architecture. International standard, second edition. 2012. Geneva: International Electrotechnical Commission.
21. *Koza J.R*. Genetic programming: on the programming of computers by means of natural selection. Vol. 1. 1992. Cambridge: MIT press.
22. *Larranaga P., Kuijpers C.M.H., Murga R.H., Inza I., Dizdarevic S*. Genetic algorithms for the travelling salesman problem: A review of representations and

operators. Artificial Intelligence Review, 1999, vol. 13, no. 2, pp. 129–170. Springer.

23. *López-Ibáñez M., Dubois-Lacoste J., Stützle T., Birattari M*. The irace package, Iterated Race for Automatic Algorithm Configuration. Technical Report TR/IRIDIA/2011-004, 2011, IRIDIA, Université libre de Bruxelles, Belgium.
24. *Mitchell M., Holland J., Forrest S*. When will a genetic algorithm outperform hill climbing. Advances in Neural Information Processing Systems, pp. 51–58. 1994. Morgan Kaufmann, San Mateo, CA.
25. *Müller R.A., Lembeck C., Kuchen H.* A symbolic Java virtual machine for test case generation. IASTED Conference on Software Engineering, vol. 4. 2004. ACTA Press.
26. nxtControl – nxtSTUDIO [Электронный ресурс]. Режим доступа http://www.nxtcontrol.com/en/engineering/ свободный. Яз. англ. (дата обращения 01.04.2015).
27. *Patil S., Vyatkin V., Sorouri M*. Formal verification of intelligent mechatronic systems with decentralized control logic. 17th IEEE Conference on Emerging Technologies & Factory Automation (ETFA'12), 2012, Krakow, Poland, pp. 1–7. IEEE.
28. *Peltola J., Sierla S., Aarnio P., Koskinen K*. Industrial evaluation of functional Model-Based Testing for process control applications using CAEX. 18th IEEE Conference on Emerging Technologies & Factory Automation (ETFA'13), 2013, Cagliary, Italy, pp. 1–8. IEEE.
29. *Schaffer J.D., Eshelman L.J*. On crossover as an evolutionary viable strategy. 4th International Conference on Genetic Algorithms, 1991, pp. 61–68. San Mateo, CA, Morgan Kaufmann.
30. *Srinivas N., Deb K.*. Muiltiobjective optimization using nondominated sorting in genetic algorithms. Evolutionary computation, 1994, vol. 2, no. 3, pp. 221–248.
31. *Thramboulidis K*. IEC 61499 in factory automation. Advances in Computer, Information, and Systems Sciences, and Engineering, pp. 115–124. 2006. Springer Netherlands.

32. *von Styp S., Yu L.* Symbolic Model-Based Testing for Industrial Automation Software. Hardware and Software: Verification and Testing, pp. 78–94. 2013. Springer International Publishing.
33. *Vyatkin V., Chouinard J*. On Comparisons of the ISaGRAF implementation of IEC 61499 with FBDK and other implementations. 6th IEEE International Conference on Industrial Informatics (INDIN'08), 2008, Daejeon, Korea, pp. 289–294.
34. *Vyatkin V., Hanisch H.-M.* Formal modeling and verification in the software engineering framework of IEC 61499: a way to self-verifying systems. 8th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2001, vol. 2, pp. 113–118. IEEE.
35. *Zoitl A., Vyatkin V.* IEC 61499 Architecture for Distributed Automation: the 'Glass Half Full' View. IEEE Industrial Electronics Magazine, 2009, vol. 3, no. 4, pp. 7–23. IEEE.

ПРИЛОЖЕНИЕ А. ПРИМЕР КОДА НА ЯЗЫКЕ JAVA, ПОДГОТОВЛЕННОГО К ЗАПУСКУ EVOSUITE

В настоящем приложении приведен код, сгенерированный по составному ФБ `my_sensor2` из системы PnP. Интерфейс ФБ и сеть из ФБ внутри него показаны на рис. 16 (снимок экрана был сделан в *FBDK*).

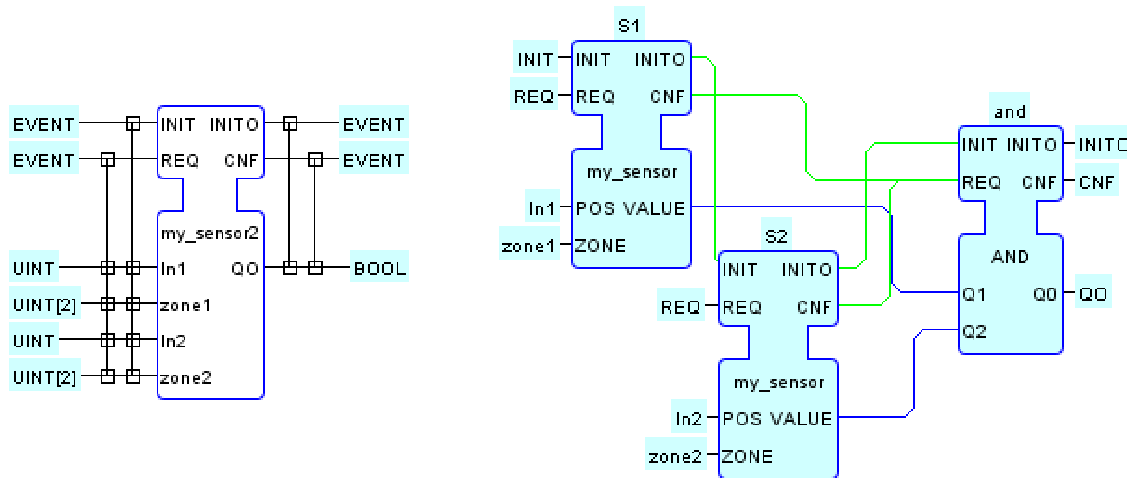

Рис. 16. Интерфейс (слева) и сеть внутренних ФБ (справа) составного ФБ `my_sensor2`.

```
package fb.rt.pnp;

import fb.rt.*;
import fb.rt.net.*;
import fb.datatype.*;

public class my_sensor2__Composite {
    private final my_sensor2 instance = new my_sensor2();

    public void event_INIT(int In1_, int zone1_0, int zone1_1, int
In2_, int zone2_0, int zone2_1) {
        instance.In1.value = Math.abs(In1_);
        ((UINT) instance.zone1.value[0]).value = Math.abs(zone1_0);
        ((UINT) instance.zone1.value[1]).value = Math.abs(zone1_1);
        instance.In2.value = Math.abs(In2_);
        ((UINT) instance.zone2.value[0]).value = Math.abs(zone2_0);
        ((UINT) instance.zone2.value[1]).value = Math.abs(zone2_1);
        instance.INIT.serviceEvent(instance);
    }

    public void event_REQ(int In1_, int zone1_0, int zone1_1, int
In2_, int zone2_0, int zone2_1) {
        instance.In1.value = Math.abs(In1_);
        ((UINT) instance.zone1.value[0]).value = Math.abs(zone1_0);
        ((UINT) instance.zone1.value[1]).value = Math.abs(zone1_1);
        instance.In2.value = Math.abs(In2_);
        ((UINT) instance.zone2.value[0]).value = Math.abs(zone2_0);
        ((UINT) instance.zone2.value[1]).value = Math.abs(zone2_1);
```

```
        instance.REQ.serviceEvent(instance);
    }

    private class my_sensor2 extends FBInstance {
        public UINT In1 = new UINT();
        public ARRAY zone1 = new ARRAY(new UINT(),2);
        public UINT In2 = new UINT();
        public ARRAY zone2 = new ARRAY(new UINT(),2);
        public BOOL QO = new BOOL();
        public EventOutput INIT = new EventOutput();
        public EventOutput REQ = new EventOutput();
        public EventOutput INITO = new EventOutput();
        public EventOutput CNF = new EventOutput();
        protected my_sensor S1 = new my_sensor() ;
        protected my_sensor S2 = new my_sensor() ;
        protected AND and = new AND() ;

        public my_sensor2() {
            super();
            INIT.connectTo(S1.INIT);
            S1.INITO.connectTo(S2.INIT);
            REQ.connectTo(S1.REQ);
            S2.INITO.connectTo(and.INIT);
            S1.CNF.connectTo(and.REQ);
            S2.CNF.connectTo(and.REQ);
            and.INITO.connectTo(INITO);
            and.CNF.connectTo(CNF);
            REQ.connectTo(S2.REQ);
            S1.connectIVNoException("POS",In1);
            S1.connectIVNoException("ZONE",zone1);
            S2.connectIVNoException("POS",In2);
            S2.connectIVNoException("ZONE",zone2);

and.connectIVNoException("Q1",S1.ovNamedNoException("VALUE"));

and.connectIVNoException("Q2",S2.ovNamedNoException("VALUE"));
            QO = (BOOL) and.ovNamedNoException("Q0");
        }
    }

    private class my_sensor extends FBInstance {
        public UINT POS = new UINT();
        public ARRAY ZONE = new ARRAY(new UINT(),2);
        public BOOL VALUE = new BOOL();
        public EventServer INIT = new EventInput(this);
        public EventServer REQ = new EventInput(this);
        public EventOutput INITO = new EventOutput();
        public EventOutput CNF = new EventOutput();

        public ANY ovNamed(String s) throws FBRManagementException {
            if ("VALUE".equals(s)) return VALUE;
            return super.ovNamed(s);
        }
```

```
        public void connectIV(String ivName, ANY newIV)
            throws FBRManagementException {
            if ("POS".equals(ivName)) connect_POS((UINT) newIV);
            else if ("ZONE".equals(ivName)) connect_ZONE((ARRAY)
newIV);
            else super.connectIV(ivName, newIV);
        }

        public void connect_POS(UINT newIV) {
            POS = newIV;
        }

        public void connect_ZONE(ARRAY newIV) {
            ZONE = newIV;
        }

        private static final int index_START = 0;

        private void state_START() {
            eccState = index_START;
        }

        private static final int index_INIT = 1;

        private void state_INIT() {
            eccState = index_INIT;
            alg_INIT();
            INITO.serviceEvent(this);
            state_START();
            transition_my_sensor_0();
        }

        private static final int index_REQ = 2;

        private void state_REQ() {
            eccState = index_REQ;
            alg_REQ();
            CNF.serviceEvent(this);
            state_START();
            transition_my_sensor_1();
        }

        public my_sensor() {
            super();
        }

        public void serviceEvent(EventServer e) {
            if (e == INIT) service_INIT();
            else if (e == REQ) service_REQ();
        }

        public void service_INIT() {
```