

Министерство образования и науки Российской Федерации
Государственное образовательное учреждение высшего
профессионального образования
«Алтайский государственный технический университет
им. И. И. Ползунова»

С. М. Старолетов, Е. Н. Крючкова

**Автоматное моделирование многокомпонентных
программных систем, их тестирование и верификация**

Учебно-методическое пособие для студентов направления «Программная
инженерия»

Барнаул — 2015

УДК 004.054

ББК 32.817

Старолетов С. М. Автоматное моделирование многокомпонентных программных систем, их тестирование и верификация: учебно-методическое пособие/ С. М. Старолетов, Е. Н. Крючкова. – Барнаул : Изд-во АлтГТУ, 2015. – 156 с.

В настоящем пособии рассматриваются способы построения автоматных моделей и инструментальных средства для проектирования и тестирования программных систем согласно принципам MDD (разработка, управляемая моделями), MBT (тестирование на основе моделей), MBC (верификация на основе моделей).

© Старолетов С. М., Крючкова Е. Н., 2015

© Алтайский государственный технический
университет им. Ползунова, 2015

Содержание

Список использованных сокращений.....	5
Введение.....	6
1 Применение моделей в процессе разработки программ. Тестирование на основе моделей. Модели распределенных взаимодействующих систем.....	7
1.1 Разработка, управляемая моделями.....	7
1.2 Тестирование ПО сегодня.....	8
1.3 Краткий обзор технологий тестирования.....	10
1.3.1 Тестирование параллельного и распределенного программного обеспечения.....	13
1.3.2 Методы проверки правильности параллельных и распределенных приложений.....	16
1.3.3 Проверка моделей.....	19
1.3.4 Тестирование распределенных приложений на основе моделей.....	20
1.4 Обзор средств тестирования на основе моделей.....	21
1.4.1 UniTesK.....	21
1.4.2 Microsoft Spec Explorer.....	22
1.4.3 IBM Rational Test RealTime.....	23
1.4.4 Conformiq Test Generator.....	24
1.4.5 AGEDIS.....	24
1.4.6 Продукты фирмы Compuware.....	26
1.4.7 AsmL Test Tool.....	26
1.5 Модели распределенных многопоточных взаимодействующих систем.....	28
1.5.1 Сети Петри.....	28
1.5.2 Язык SDL.....	30
1.5.3 Язык UML.....	31
Задание к главе 1.....	33
2 Пример построения математической модели современных распределенных недетерминированных систем на основе автоматов.....	34
2.1 Модели в исследовании свойств алгоритма. Проблема тестирования.....	34
2.2 Распределенные недетерминированные взаимодействующие системы. Постановка задачи на тестирование.....	37
2.3 Модель распределенных недетерминированных систем.....	39
2.3.1 Конечный автомат	39
2.3.2 Конечный автомат с вероятностными переходами.....	47
2.3.3 Конечный автомат с вероятностными переходами и обработкой событий и исключений.....	50
2.3.4 Расширение автомата для поддержки моделирования многопоточных приложений.....	57
2.3.4.1 Параллелизм.....	58
2.3.4.2 Операция создания потока и понятие кратности.....	60
2.3.4.3 Операция ожидания потоков.....	65

2.3.4.4 Обмен сообщениями.....	66
2.3.4.5 Блокировка ресурсов.....	70
2.3.5. Модель многокомпонентной системы высокого уровня.....	72
2.3.6 Моделирование межкомпонентной связности.....	77
2.3.7 Итоговая модель системы в виде взаимодействующих компонентов.....	82
2.4 Предварительная оценка модели.....	85
Задание к разделу 2.....	86
3 Практическая реализация процессов разработки и тестирования на основе моделей	87
3.1 Способ представления автомата в виде состояний, переходов и операций	87
3.2 Способы описания модели и ее соотношение с исходным кодом системы.....	89
3.2.1 Пример описания модели при отсутствии кода.....	89
3.2.2 Пример описание модели при наличии кода	91
3.3 О практической значимости и адекватности модели	94
3.4 Процесс разработки и тестирования на основе моделей.....	104
3.5 Статическая верификация модели.....	109
3.6 Динамическое тестирование по модели.....	114
Задание к главе 3.....	120
Список использованной литературы.....	121
Приложение А. Пример моделирования АОП функциональности.....	133
Приложение Б. Объектная мета-модель.....	136
Приложение В. Пример использования объектно-ориентированной реализации многокомпонентной системы на языке Java.....	142
Приложение Г. Контекстно-свободная грамматика языка описания модели.....	145
Г.1 Грамматика, описывающая модель высокого уровня (структурный автомат).....	146
Г.2 Грамматика, описывающая модель на уровне компонента системы.....	147
Приложение Д. Способы получения эквивалентной модели на языке Promela в целях верификации.....	150
Д.1 Система переходов и недетерминированный выбор состояния.....	150
Д.2 Создание и ожидание потоков.....	152
Д.3 Отправка и получение сообщений.....	155
Д.4 Блокировка и разблокировка ресурсов.....	156

Список использованных сокращений

АОП – аспектно-ориентированное программирование

ПО – программное обеспечение

ASM – abstract state machine (машина абстрактных состояний)

MBT – model based testing (тестирование на основе моделей)

MDD – model driven development (разработка, управляемая моделями)

UML – unified modeling language (унифицированный язык моделирования)

XML – extensible markup language (расширяемый язык разметки)

Введение

С ростом сложности программных систем возникает проблема обеспечения достаточного уровня надежности разрабатываемого ПО, ошибки в котором могут нанести серьезный экономический ущерб и привести к жизненно-опасным ситуациям. Современные технологии программирования не могут обеспечить эффективных методов безошибочного проектирования ПО. В настоящее время на рынке нет понятных и эффективных продуктов для тестирования программ с использованием математических моделей, как нет и общепризнанных математических моделей для описания многокомпонентных распределенных систем. Большая работа проделана в исследовательском центре корпорации Microsoft профессором Ю. Гуревичем, ведутся исследования в научных центрах NASA и Bell labs, в России следует отметить работы института системного программирования РАН и кафедры технологий программирования СПбГУ ИТМО. Однако практическая применимость для тестирования предлагаемых моделей в компаниях по разработке ПО пока невелика, что является следствием сложности предлагаемых моделей и низкой степенью их вовлечения в реальные процессы разработки. В связи с этим актуальным является комплексное исследование предметной области в сфере сложных многокомпонентных взаимодействующих программ, их моделирование и практическое применение построенной модели для верификации и тестирования ПО.

1 Применение моделей в процессе разработки программ. Тестирование на основе моделей. Модели распределенных взаимодействующих систем

1.1 Разработка, управляемая моделями

Рассматривая пути развития современных программных систем, можно отметить следующие факторы:

1. Аппаратная часть постоянно усовершенствуется, усложняются задачи, которые ставятся перед компьютерными системами, программный код становится сложнее.
2. При разработке систем любого уровня сложности используются библиотеки и компоненты, написанные и протестированные ранее.
3. Задачи решаются распределенно и параллельно, все чаще используется сервисная архитектура.
4. Язык программирования уже не является определяющим фактором в разработке системы. Компоненты могут быть реализованы на различных языках и взаимодействовать по протоколам через сети передачи данных.
5. С прогрессом в области технологий разработки программного обеспечения повышается уровень абстракции программного кода – от низкоуровневых инструкций к использованию компонент со сложным поведением на основе заранее определенных интерфейсов, при этом при разработке, тестировании и анализе необходимо представлять программную систему на различных уровнях абстракции.

В последнее время в программной инженерии набирает популярность подход, при котором в процессе первичного проектирования программной системы разрабатывается некоторая модель, с нужной степенью абстракции описывающая данную систему, далее по модели генерируется программный код, документация, проводится тестирование. Такой подход называется «разработка, управляемая моделями» (model driven development – MDD), который по-

степенно вырос в отрасль разработки программ на основе моделей (Model driven engineering). Работы в этой области спонсируются, например, такими мировыми корпорациями, как IBM, понимающими, что для критически сложных систем для бизнеса (Enterprise-решения) нужно высокое качество процесса разработки и, следовательно, порождаемого в нем кода.

Как известно, в процессе разработки тестирование полученного продукта занимает большую часть времени. Модель программной системы, которая может быть использована в процессе ее разработки с применением принципа MDD, может быть использована для проверки ее правильности, если для описания модели путем декомпозиции реальной системы изначально были выбраны параметры, адекватно описывающие проверяемое состояние системы.

Пособие посвящено формализации современных взаимодействующих систем, основанная на построении модели и ее использовании для проведения тестирования, следуя принципам разработки, управляемой моделями. При этом использованы как математические методы, чтобы сделать описание модели более строгим, так и методы объектно-ориентированного анализа для обеспечения внедрения моделирования в реальный процесс разработки.

1.2 Тестирование ПО сегодня

Тестирование программного обеспечения — это инструмент повышения его качества с целью выявления возможных ошибок в нем. Далее в главе 2 будет показано, что для произвольной программы доказать ее безошибочность невозможно, поэтому предлагается выбрать это определение тестирования из других определений, акцентируя внимание именно на словах «выявление ошибок» вместо, например, «подтверждения правильности».

Тестирование программных продуктов может производиться разработчиками, узкими специалистами-тестировщиками, а также пользователями. В последнем случае, если, конечно, продукт не является специальным релизом для публичного тестирования (так называемой *бета-версией*), выявление серьезных ошибок в уже выведенном на рынок товаре - программном обеспече-

нии - говорит о низком качестве продукта и может привести к различным неприятным последствиям. Поэтому фирмы-производители программ и индивидуальные разработчики заинтересованы в минимизации ошибок, а следовательно, в повышении качества процесса разработки ПО, в том числе, улучшении способов тестирования.

Естественно, у разных групп по тестированию ПО будут различные цели, задачи и способы. Возможность использования исходного кода системы в процессе тестирования — это первый существенный признак классификации. Например, разработчики и высококвалифицированные тестировщики имеют доступ к исходному коду тестируемой системы на языке программирования и могут использовать эти знания в процессе поиска возможных ошибок. Тестировщикам же невысокой квалификации нет смысла просматривать исходный код, поскольку он ничего им не может сказать, в таком случае система рассматривается извне как черный ящик и анализируется ее поведение с точки зрения заранее описанных спецификаций и конечного результата.

Итак, согласно устоявшейся терминологии, тестирование с точки зрения использования исходного кода системы подразделяется на :

- тестирование *черного ящика* (black box testing), когда исходный код системы недоступен и производится проверка соответствия системы формальной спецификации или ожидаемому поведению;
- тестирование *белого ящика* (white box testing), когда для тестирования доступен ее исходный код.

Тестирование также можно классифицировать по функциональному назначению (например, *модульное тестирование* проверяет классы и модули программы; *регрессивное* — контролирует, что после изменений старый функционал системы работает корректно; *нагрузочное* — как тестируемая система будет вести себя под большой нагрузкой). Некоторые виды тестирования меняют даже сам способ разработки ПО (пример — *разработка через тестирование* — test driven development, с применением модульных тестов повсеместно).

В задачу исследования не входит подробная классификация всех видов

тестирования, в данной главе рассмотрим только те, которые представляют интерес с точки зрения тестирования рассматриваемых в данной работе современных распределенных систем.

Работа специалиста-тестировщика в компании по разработке ПО обычно представляет собой составление *плана тестирования* в соответствии со спецификацией (дизайном) системы, который включает *тестовые наборы* (test suites) и *тестовые образцы* (test cases), и далее проверку правильности ПО в соответствии с данным планом. Соответственно, тестирование может проводиться как вручную, так и автоматически.

Однако, как заметил Ю. Гуревич из Microsoft, при усложнении системы и приеме на работу все большего и большего количества специалистов по тестированию происходит насыщение и они уже не могут гарантировать качества безошибочной работы. Это означает, что для тестирования сложных взаимодействующих систем необходимо внедрять новые методологии.

С начала 2000-х годов набирает популярность применение в процессе тестирования *моделирования*, использование математических методов. Разработка модели взаимодействующей системы и использование математических методов в тестировании — это основная тема данной работы.

1.3 Краткий обзор технологий тестирования

Как уже было отмечено, в тестировании белым ящиком применяется доступ к исходному коду системы. *Модульное* тестирование (семейство xUnit для различных языков программирования — JUnit для языка программирования Java, NUnit для языка C# и проч.) является на сегодняшний день **наиболее популярным** методом тестирования (фактически, промышленным стандартом при разработке программного обеспечения) и представляет собой проверку правильности работы методов и классов тестируемого модуля, причем рекомендуется проверять каждый нетривиальный метод. Проверка методов заключается в создании экземпляров классов и вызовов методов из них, со сравнением результата с ожидаемым (результат — это возвращаемое значение метода, некоторое поле в классе и т.д.). Тесты обычно группируются в тестовые наборы.

Формально, при модульном тестировании происходит проверка истинности выражения

$$\bigwedge_{M_i \in TestCase} (M_i(x_1, x_2, \dots, x_n) = Ret_{expect}(M_i)), \quad (1.1)$$

где $M_i(x_1, x_2, \dots, x_n)$ - результат работы метода из тестового набора $TestCase$ с n входными параметрами, $Ret_{expect}(M_i)$ - ожидаемое возвращаемое значение метода. Операция конъюнкции означает то, что при несоответствии возвращаемого значения ожидаемого в одном из методов набора, работа всего тестового набора считается неправильной.

Поддерживаются также тестирование возникновения исключительных ситуаций, тестирование выполнения по времени, параметризация, а также группировка тестов в наборы. На рисунке 1.1 показано окно среды разработки Eclipse с результатами теста.

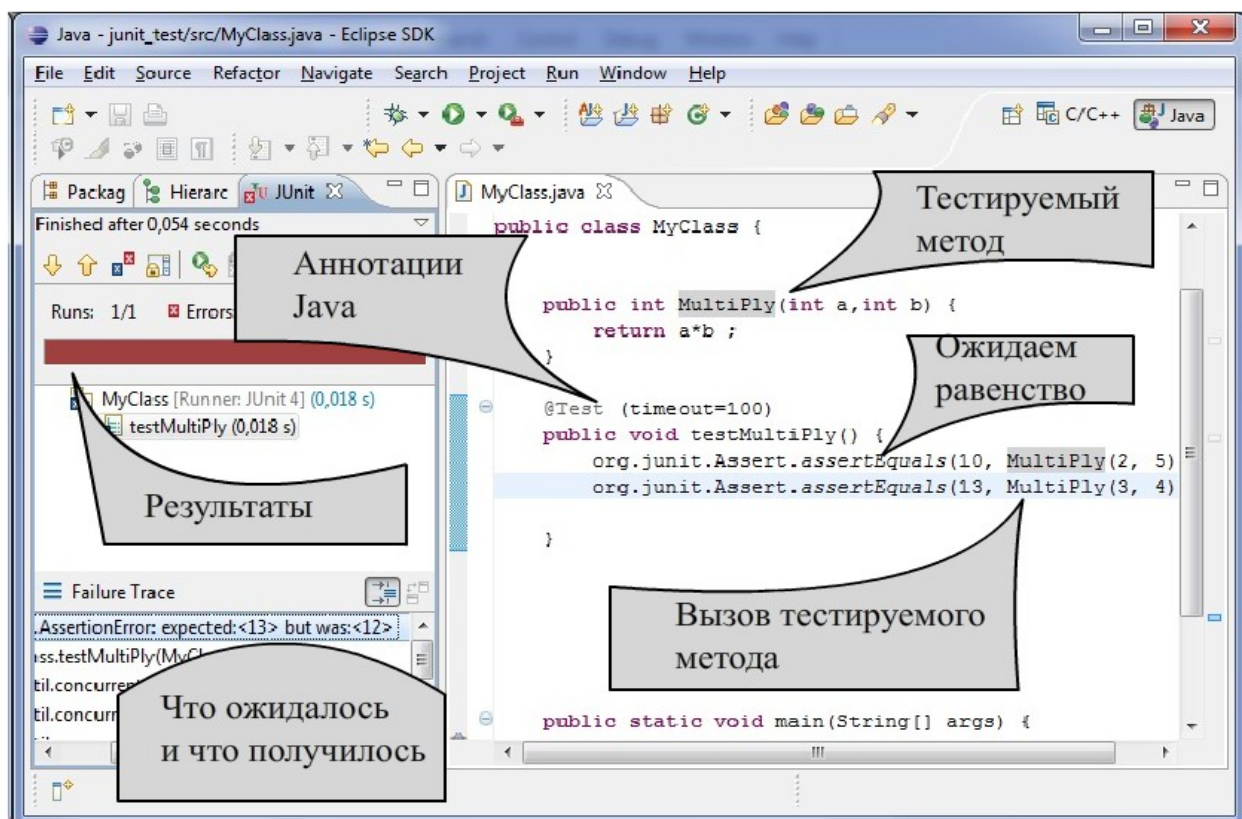


Рисунок 1.1 — Неуспешный тест, JUnit в среде разработки Eclipse

Использование xUnit тестов дало начало специальной технике программирова-

ния — *разработке через тестирование* или красно-зеленый рефакторинг, от цвета полосы с результатами: красный (рисунок 1.1) — ошибка в тесте и зеленый — тест пройден. Данная технология предполагает, что тесты пишутся до разработки кода системы и сама разработка представляет собой процесс написания исходного кода так, чтобы выполнялись все предварительно написанные тесты, и дальнейшего рефакторинга кода.

В настоящее время модульное тестирование применяется широко в индустрии разработки ПО, тесты обычно пишутся самими разработчиками системы. Модульное тестирование может использоваться в качестве *регрессивного*, когда при добавлении новой функциональности выполняются все старые тесты, и, если они проходят, новая возможность считается добавленной правильно.

Модульное тестирование тесно связано с покрытием кода. *Покрытие кода* — это метрика, показывающая, сколько процентов кода заданной структуры используется в тестах. Цель покрытия может быть, например, покрыть все операторы или все ветви программы, причем второе условие сильнее первого.

Однако даже 100% покрытие всех условий не дает гарантии, что система, являющаяся взаимодействующей, будет в итоге без ошибок, поскольку модульное тестирование в состоянии только проверить классы и методы по одиночке и тем более без учета событийно-ориентированной работы системы, а поэтому не гарантирует правильность распределенной системы в целом. Можно привести в качестве примера браузер компании Google (Chrome), представленный в 2008г. Продукт поставляется с тестами для системы тестирования Gtest (являющейся xUnit системой для языка C++), и, как известно, мировые корпорации - производители ПО требуют максимального покрытия кода тестами у своих разработчиков. Однако это не помешало данному браузеру иметь ошибки и заставило компанию выпускать заплатки, их исправляющие.

Обсуждается, что методика TDD (и, соответственно, модульное тестирование вообще) сталкивается с проблемами, которые в его рамках решить скорее всего нельзя. Это:

- автоматическое тестирование GUI (пользовательского интерфейса);
- тестирование распределенных объектов;
- схемы базы данных;
- разработка компиляторов/интерпретаторов.

Следовательно, нужно искать и развивать новые методы в сфере тестирования ПО.

1.3.1 Тестирование параллельного и распределенного программного обеспечения

Параллельное и распределенное программирование — это два базовых подхода к достижению параллельного выполнения составляющих программного обеспечения. Они представляют собой две различные парадигмы программирования, которые иногда пересекаются. Методы параллельного программирования позволяют распределить работу программы между двумя (или больше) процессорами в рамках одного физического или одного виртуального компьютера. Программа, содержащая параллелизм, выполняется на одном и том же физическом или виртуальном компьютере. Такую программу можно разбить на процессы или потоки. Поэтому параллельные программы также называют многопоточными.

Методы распределенного программирования позволяют распределить работу программы между двумя (или больше) процессами, причем процессы могут существовать на одном и том же компьютере или на разных, то есть части распределенной программы зачастую выполняются на разных компьютерах, связываемых по сети, или, по крайней мере, в различных процессах. Части распределенных приложений обычно реализуются как отдельные программы. Важнейшим аспектом параллельных и распределенных систем является взаимодействие между частями системы, так как именно взаимодействие порождает основные трудности, как при разработке, так и при тестировании параллельных и распределенных систем. В зависимости от архитектуры параллельной или распределенной системы, от способов её декомпозиции для

организации взаимодействий используются различные методы. Например, взаимодействие через разделяемую память (при этом требуется дополнительное взаимодействие по синхронизации операций над разделяемой памятью), взаимодействие путем вызовов удаленных процедур, путем организации транзакций (клиент-серверное взаимодействие), взаимодействие путем обмена сообщениями.

Важным аспектом функционирования распределенной системы является доступ частей системы к разделяемым ресурсам. При параллельном выполнении потоков управления может сложиться ситуация, в которой к одному ресурсу одновременно обращаются несколько потоков управления. Здесь одновременность означает, что за время, необходимое для выполнения запроса некоторого потока управления к ресурсу, могут поступить запросы от других потоков к тому же самому ресурсу. Тестирование распределенных и параллельных приложений имеет ряд особенностей. При тестировании последовательной программы разработчик может отследить ее логику в пошаговом режиме. Если он будет начинать тестирование с одних и тех же данных при условии, что система каждый раз будет пребывать в одном и том же состоянии, то результаты выполнения программы или ее логические цепочки будут вполне предсказуемыми. Программист может отыскивать ошибки в программе, используя соответствующие входные данные и исходное состояние программы, путем проверки ее логики в пошаговом режиме.

Тестирование и отладка в последовательной модели зависят от степени предсказуемости начального и текущего состояний программы, определяемых заданными входными данными. С параллельными и распределенными программами все обстоит иначе. Здесь трудно воспроизвести точный контекст параллельных или распределенных задач из-за разных стратегий планирования, применяемых в операционной системе, динамически меняющейся рабочей нагрузки, квантов процессорного времени, приоритетов процессов и потоков, временных задержек при их взаимодействии и собственно выполнении, а также различных случайных изменений ситуаций, характерных для параллельных или

распределенных контекстов. Чтобы воспроизвести точное состояние, в котором находилась среда при тестировании и отладке, необходимо воссоздать каждую задачу, выполнением которой была занята операционная система. При этом должен быть известен режим планирования процессорного времени и точно воспроизведены состояние виртуальной памяти и переключение контекстов. Кроме того, следует воссоздать условия возникновения прерываний и формирования сигналов, а в некоторых случаях — даже рабочую нагрузку сети. При этом нужно понимать, что и сами средства тестирования и отладки оказывают немалое влияние на состояние среды. Это означает, что создание одинаковой последовательности событий для тестирования и отладки зачастую невозможно. Необходимость воссоздания всех перечисленных выше условий обусловлено тем, что они позволяют определить, какие процессы или потоки следует выполнять и на каких именно процессорах. Смешанное выполнение процессов и потоков (в некоторой неудачной «пропорции») часто является причиной возникновения взаимоблокировок, бесконечных отсрочек, «гонки» данных и других проблем. Таким образом, ошибки распараллеливания не только сложно выявить из-за недетерминированности поведения параллельных приложений, но даже если ошибка обнаружена, ее часто сложно воспроизвести повторно. Кроме того, после модификации кода, не так просто убедиться, что ошибка действительно устранена, а не замаскирована

1.3.2 Методы проверки правильности параллельных и распределенных приложений

В тестирование параллельных и распределенных приложений входят проверки правильности, стабильности, производительности и масштабируемости. Правильность работы определяется при помощи таких методов, как статический анализ, динамический анализ и проверка моделей.

Статический анализ

Статический анализ — это анализ кода без запуска приложения. Обычно при статическом анализе проверяются метаданные откомпилированного приложения или откомментированного исходного кода. Зачастую в него также включается некий этап формальной проверки, который должен гарантировать, что предположения, на которые опирался программист во время разработки, не приводят к неправильному поведению приложения. Статический анализ имеет как преимущества, так и недостатки. В качестве преимуществ можно назвать возможность убедиться в правильности проекта приложения, точность отчетов об ошибках, упрощающих поиск и устранение дефектов в программе. Статический анализ помогает устранять дефекты только при наличии достаточного количества комментариев к коду. Комментарии должны быть верными. Кроме того, средства статического анализа дают большое количество ложных положительных результатов и требуют немалых усилий для борьбы с ними.

В качестве примера средства статического анализа для параллельного выполнения можно привести **RacerX**. Это средство статического анализа, зависящее от организации операций, которое используется для выявления состояний гонки и взаимоблокировок. Оно позволяет избежать необходимости скрупулезно комментировать весь исходный код. Единственная информация, необходимая ему — это таблица, в которой указываются интерфейсы API, используемые для получения и освобождения блокировок, атрибуты примитивов блокировки.

На первом этапе RacerX несколько раз проверяет каждый файл исходного кода и составляет график логики управления — Control Flow Graph (CFG). График CFG содержит информацию о вызовах функций, общей памяти, использовании указателей и другие данные. Когда график CFG полностью построен, начинается фаза анализа, включающая проверку на предмет наличия состояний гонки и взаимоблокировок. Проверка графика CFG может занимать много времени, но для максимального сокращения этой операции применяются

технологии сокращения и кэширования. После проверки контекстных потоков включается алгоритм блокирования, выявляющий потенциальные состояния гонки. В ходе анализа взаимоблокировок для каждого захвата блокировки вычисляются циклы блокировки.

Динамический анализ

При динамическом анализе отслеживание дефектов ведется по результатам запуска приложения. Динамический анализ бывает двух типов: интерактивный и автономный. Средства интерактивного динамического анализа проверяют приложение непосредственно во время его работы; средства автономного динамического анализа фиксируют ход работы приложения, а затем проверяют записи, выявляя дефекты. Динамический анализ удобен тем, что он не требует практически никаких дополнительных усилий от разработчика на этапе создания кода, он дает меньше ложных положительных результатов и упрощает устранение наиболее очевидных проблем. Поскольку дефекты приложения определяются только в ходе работы приложения, первыми обнаруживаются те неполадки, которые связаны с наиболее часто выполняемыми операциями. Такая особенность динамического анализа снижает затраты на повышение стабильности приложения. Недостатком динамического анализа является то, что проверка происходит исключительно во время работы приложения, и процент покрытия кода при тестировании зависит от того, насколько хорошо написаны тестовые примеры. Некоторые средства способны зафиксировать состояние гонки, только если оно возникло в данном (проверяемом) сеансе работы программы. То есть, если средство анализа сообщает, что ошибок нет, это не гарантирует отсутствие ошибок при следующих запусках. Другой недостаток состоит в том, что большинство средств динамического анализа зависят от некоего инструментария, позволяющего менять поведение исполняющей среды. Из-за сложности таких средств, их производительность не слишком высока.

Примером средства динамического анализа является **CHESS**. Средство **CHESS**, разработанное специалистами Майкрософт, выявляет ошибки параллельной обработки операций путем систематического исследования диспетчеризаций потоков и чередования. Это средство способно обнаруживать неполадки, связанные с состояниями гонки, взаимоблокировками, зависанием, активными блокировками и повреждением данных. Оно также упрощает устранение неполадок, поскольку дает полностью воспроизводимые результаты. **CHESS**, будучи средством динамического анализа, последовательно выполняет обычный модульный тест циклически с помощью специализированного планировщика.

Еще одним инструментом динамического анализа является **Intel Thread Checker**. Позволяет выявлять взаимоблокировки (в том числе и потенциальные), зависания, состояния гонки за данные и случаи некорректного использования интерфейсов API Windows для синхронизации. Средство **Thread Checker** требует, чтобы в исходном коде или скомпилированном двоичном файле было инструментирование, позволяющее наблюдать каждую ссылку на область памяти и каждый стандартный примитив синхронизации Win32. Крупным недостатком данного средства является то, что оно не может учитывать синхронизацию через взаимозаблокированные операции, например, через операции, используемые в круговой блокировке.

1.3.3 Проверка моделей

Проверка моделей – метод проверки правильности работы конечного автомата, в котором применяется параллельная обработка. Этот метод допускает формальный дедуктивный анализ. Средство проверки моделей пытается смоделировать состояния гонки или взаимоблокировки. Проверка на основе моделей позволяет формально обосновать отсутствие этих дефектов. Такой метод помогает убедиться в правильности проекта и архитектуры приложения, полностью охватывает все его компоненты и требует

минимального количества внешних ресурсов. Проверка на основе моделей имеет свои недостатки. В большинстве случаев очень сложно автоматически вывести модель из кода. Вывод модели вручную — занятие трудоемкое. Кроме того, при взрывном расширении пространства состояний (условии, при котором имеется слишком много возможных состояний автомата) объем проверки увеличивается до недопустимого. Расширение пространства состояний можно в некоторой степени контролировать, применяя методы сокращения, использующие сложную эвристику, но это может привести к тому, что некоторые дефекты будут пропущены. Проверка на основе моделей требует наличия качественного плана проектирования и реализации. Она может подтвердить правильность проекта, но не гарантирует отсутствия ошибок в реализации.

Примерами инструментов проверки моделей являются **KISS** и **Zing**.

KISS — разработка специалистов Майкрософт для параллельных приложений, написанных на C. **KISS** преобразует параллельное приложение на языке C в последовательное приложение и моделирует чередование. После этого выполняется анализ с применением последовательного средства проверки на основе моделей. Это средство не дает ложных положительных результатов. Данное средство является исследовательским прототипом, оно применялось группой по разработке драйверов для Windows, в которых в основном используется язык C.

Средство **Zing** предназначено для тестирования проектов параллельных приложений. В **Zing** есть свой собственный язык, применяемый для описания сложных состояний и переходов. Это средство имеет все необходимое для моделирования параллельных конечных автоматов. Средство также позволяет контролировать расширение пространства состояний, применяя современные приемы редукции. Модель, используемая в средстве **Zing** для проверки правильности программы, должна создаваться либо трансляторами, либо

вручную. Полного и эффективного преобразователя для приложений в машинном коде и приложений CLR на данный момент нет. По этой причине **Zing** применяется в крупных проектах только для проверки правильности отдельных блоков.

1.3.4 Тестирование распределенных приложений на основе моделей

Описанные выше методы тестирования параллельных и распределенных программ не могут полностью решить проблему обеспечения качества программных систем, поэтому в настоящее время все больше внимания стали уделять формальным методам. Под формальными методами понимаются любые попытки использования математических подходов к разработке программного обеспечения с целью повышения его качества. Как правило, математический подход включает:

- модель системы, которая формально представляет состояние системы, переход из одного состояния в другое;
- метод спецификации для представления желаемых свойств на некотором логическом языке;
- метод доказательства того, что модель удовлетворяет этим свойствам.

Многочисленные исследования в области формальных методов нашли свое отражение и в развитии новых технологий тестирования. Одно из наиболее активно развивающихся направлений – тестирование на основе моделей.

Тестирование на основе моделей (**model-based testing**) – это тестирование программного обеспечения, в котором варианты тестирования частично или целиком получаются из модели, описывающей некоторые аспекты тестируемой системы. Модель — некоторое отражение структуры и поведения системы. Модель может описываться в терминах состояния системы, входных воздействий на нее, конечных состояний, потоков данных и потоков

управления, возвращаемых системой результатов и т.д.

Существует два подхода к тестированию на основе моделей. Первый подход предполагает автоматизацию процесса генерации тестов по формальным спецификациям (модели) системы. Упор делается в первую очередь на генерацию тестов, ориентированных на поток управления. Тестирование сложного поведения систем, зависящего от потока данных, является трудоемкой задачей. Второй подход предполагает отслеживание поведения системы, включающее динамический сбор сведений о состояниях, переходах, возвращаемых результатах, и сверке реального поведения системы с поведением формальной модели.

Применение метода тестирования на основе моделей требует от пользователя не только умения построить формальную модель приложения, но и специализированных знаний как о приложении, так и о технике тестирования на основе моделей.

1.4 Обзор средств тестирования на основе моделей

В настоящее время существует ряд программных средств, реализующих методику тестирования на основе моделей. Наиболее известны среди них такие программные продукты, как UniTesK, Microsoft Spec Explorer, IBM Rational Test RealTime, Conformiq Test Generator и AGEDIS.

1.4.1 UniTesK

Технология промышленного тестирования UniTesK разработана группой спецификации, верификации и тестирования Института системного программирования РАН и основана на формальных спецификациях. В составе UniTesK имеются средства тестирования приложений C++, C#, Java.

В качестве основной модели в технологии UniTesK выступает модель требований. Эта модель создается первой и именно она является основой для

разработки остальных моделей. UniTesK для описания модели предлагает использовать так называемые неявные спецификации или спецификации ограничения. Они задаются в виде пред- и постусловий процедур и инвариантных ограничений на типы данных. Предусловие операции определяет, какие значения входных параметров являются допустимыми для передачи целевой системе. Постусловие операции определяет, какие выходные значения параметров являются корректными для данных значений входных параметров. Этот механизм не позволяет описывать в модели алгоритмы вычисления ожидаемых значений функций, а только их свойства.

Тесты в UniTesK генерируются автоматически. Все, что требуется от разработчика теста — это задание способа вычисления состояния модели на основании состояния целевой системы и способа перебора применяемых в текущем состоянии тестовых воздействий. Эти вычисления записываются в тестовых сценариях. Очередное тестовое воздействие выбирается на основании спецификации сценария в зависимости от результатов предыдущих воздействий. По мере выполнения теста строится конечный автомат пройденных состояний.

Спецификации на основе программных контрактов в рамках технологии UniTesK могут использоваться не только как вставки в исходный код целевой системы. Они могут быть отделены от целевого кода и использоваться в неизменном виде для тестирования различных реализаций одной и той же функциональности, таким образом, представляя собой формализацию функциональных требований к ПО.

UniTesK поддерживает интеграцию составляющих его инструментов в популярные средства разработки программ (например, Microsoft Visual Studio).

1.4.2 Microsoft Spec Explorer

Microsoft Spec Explorer появился в результате серьезной проработки

методов тестирования на основе моделей в компании Microsoft. Модель системы в Spec Explorer описывается на специальном языке выполнимых спецификаций Spec# или на языке ASML. В ASML реализованы все основные идеи подхода ASM (Abstract State Machines): минимизация объема спецификации, выполнимость, разделение модели на несколько уровней детализации, ясная и однозначная семантика.

Модель описывает переменные состояний и переходы. Переходы между состояниями — выполнение методов моделируемой программы, которые удовлетворяют данным предусловиям состояния. В результате Spec Explorer строит конечный граф с состояниями и переходами.

Spec Explorer исследует возможные трассы выполнения алгоритмов и автоматически создает на их основе тестовые наборы поведения системы, которые могут быть запущены вместе с тестируемой системой для проверки описанного и полученного поведения. Результат работы реализованной системы на множестве тестовых наборов сравнивается с поведением абстрактной модели системы.

Данный программный продукт интегрируется с Microsoft Visual Studio.

1.4.3 IBM Rational Test RealTime

Rational Test RealTime — продукт разработан для автоматизации тестирования и анализа систем реального времени, встроенных и сетевых систем и предлагает следующие возможности:

- создание тестовых сценариев, драйверов, сопровождения, заглушек и проб для анализа времени выполнения прямо из исходного кода;
- выполнение тестов на целевом компьютере осуществляется из пользовательской среды разработки;
- загрузка результатов прогонов тестов в сводные отчеты с детальной

информацией о покрытии кода (9 уровней);

- обнаружение утечек памяти, профилирование узких мест и трассировка функций времени выполнения;
- поддержка связи между кодом, тестами и UML-моделями;
- поддержка регрессивное тестирование;
- интеграция с ведущими продуктами сторонних производителей, такими как Mathworks Simulink, Microsoft Visual Studio и TI Code Composer Studio;
- расширение для Eclipse обеспечивает интеграцию с Eclipse C/C++ Development Tools (CDT).

В качестве целевых компьютеров могут служить компьютеры всех уровней сложности и всех общеизвестных платформ – от 8-bit микрочипов до 64-bit RTOS. Rational Test RealTime был одним из первых средств тестирования на основе моделей.

1.4.4 Conformiq Test Generator

Conformiq Test Generator™ – коммерческий инструмент для автоматизированного тестирования на основе моделей распределенных систем, таких как системы контроля реального времени, телекоммуникационных систем и web-приложений. Модели представлены UML диаграммами состояний с поддержкой свойств реального времени. Для описания действий и структур данных вводится язык Action Language. Редактор UML моделей позволяет каждой модели содержать несколько конечных автоматов, которые в свою очередь состоят из нескольких уровней иерархии. Экземпляры конечных автоматов могут создаваться динамически, наподобие создания новых объектов в объектно-ориентированных языках. Работает в пакетном и динамическом режимах тестирования. Тестовые наборы генерируются автоматически в языки

1.4.5 AGEDIS

Методология тестирования AGEDIS включает:

- создание модели поведения тестируемой системы;
- аннотацию модели информацией о тестировании – описание критериев покрытия, специальных тестовых назначений и ограничений тестирования;
- автоматическую генерацию тестового набора;
- проведение анализа модели, тестовой информации, тестового набора совместно с разработчиками и заказчиками;
- автоматическое выполнение тестового набора и журнализации результатов.

AGEDIS обеспечивает следующие инструменты для реализации своей методологии тестирования:

- Model Compiler – на вход принимает модель, описанную на AGEDIS Modeling Language (AML), и формирует результат в стандартном промежуточном формате, который может воспринимать генератор тестового набора. Язык моделирования включает как модель поведения, так и директивы генерации тестов;
- Test Suite Generator – читает промежуточный формат и генерирует тестовый набор в формате Abstract Test Suite (ATS);
- Test Execution Engine – читает абстрактный тестовый набор и директивы выполнения тестов, выполняет тестовый набор на тестируемом приложении и журнализирует результаты прогона;

- AML Profile – дает возможность инструменту моделирования Objecteering UML формировать модели в AML;
- браузер абстрактного тестового набора.

Разрабатываются инструменты – интегрированная платформа для всех инструментов AGEDIS, браузер отчетов, автоматизированные компоненты для анализа и повтора шагов методологии.

1.4.6 Продукты фирмы Compuware

QADirector – интегрированная среда для автоматизированного тестирования распределенных крупномасштабных приложений на протяжении их полного жизненного цикла. При использовании вместе с Reconcile (система управления требованиями, позволяющая создавать, изменять, отслеживать требования к проекту и составлять о них отчеты) дает возможность генерировать тест-планы в соответствии с изменяющимися требованиями.

QARun – инструмент для автоматизированного функционального тестирования распределенных систем. Для автоматизации генерации тестовых сценариев используется объектно-ориентированный подход. Служит для тестирования веб-приложений, Java-приложений, а также клиент-серверных и основанных на эмуляторах приложений. Поддерживается репозиторий для хранения сценариев, проверок, событий и определений объектов и ссылок между ними.

TestPartner – инструмент для автоматизированного функционального тестирования, который был специально спроектирован для тестирования сложных приложений, основанных на Microsoft, Java и web-технологиях. Позволяет создавать многократно используемые тесты с помощью визуализации сценариев и автоматических мастеров.

QACenter Performance Edition – интегрированная среда тестирования

производительности и мониторинга сервера для веб-приложений и распределенных клиент-серверных приложений. Позволяет записывать и проигрывать виртуальные пользовательские транзакции для воспроизведения точных промышленных симуляций и измерения производительности и масштабируемости приложений.

1.4.7 AsmL Test Tool

AsmL Test Tool – интегрированная среда для тестирования на основе моделей AsmL, которая находится в стадии прототипирования. AsmL (Abstract State Machine Language) – язык исполнимых спецификаций, основанный на теории Abstract State Machines. Текущая версия AsmL2 встроена в Microsoft Word и Microsoft Visual Studio.NET и использует XML и Word для написания спецификаций. Инструмент для генерации тестов, который работает с этим спецификационным языком, поддерживает как генерацию тестовых последовательностей, так и генерацию параметров. Модель ASM сводится к конечному автомату, что и позволяет генерировать тестовые последовательности. Инструмент позволяет выполнять AsmL модель параллельно с конкретной реализацией для проверки на соответствие данной реализацией со спецификацией. Алгоритм генерации тестов выполняет полное покрытие переходов полученного конечного автомата, при этом тестовые наборы вырабатываются на основе модели. Эта интегрированная среда реализует полуавтоматический подход, требуя от пользователя в графическом редакторе аннотировать модели информацией для генерации параметров и последовательностей вызовов, а также формировать связывания между моделью и реализацией для тестирования на соответствие.

1.5 Модели распределенных многопоточных взаимодействующих систем

Кратко рассмотрим модели (как математические, так и графические), которые разработаны для описания взаимодействующих систем:

- сети Петри;
- язык SDL;
- язык UML.

Определим их пригодность для целей тестирования.

1.5.1 Сети Петри

В 60-х годах XX века математиком К. Петри был предложен способ описания асинхронных взаимодействующих систем на основе сетей специального вида. Под сетью здесь понимается двудольный ориентированный граф, содержащий вершины двух типов — вершины-места и вершины-переходы. Формально, сеть Петри задаётся как:

$$Net = (P, T, F, W, M_0, M). \quad (1.2)$$

Здесь (P, T, F) - конечная сеть из вершин-мест из множества P , переходов из множества T , F - функция инцидентности, W - функция кратности дуг M_0 - начальная разметка, M - множество разметок сети. Функция $W: F \rightarrow N / \{0\}$ определяет для каждой дуги число $n > 0$ - кратность дуги (число на дуге, показывающее, что две смежные вершины сети соединены пучком из n дуг); $M_0: P \rightarrow N$ сопоставляет первоначально каждому месту $p \in P$ некоторое число $M_0(p) \in N$ - разметку места (в графической интерпретации - рисуется помещением в вершину p числа фишек(точек), равного $M_0(p)$). На основе отношения инцидентности F и кратности дуг W вводится понятие функции инцидентности $F(x, y): P \times T \vee T \times P \rightarrow N$, которая принимает значение 0 в случаях $\neg(xFy)$ или n , когда $(xFy) \wedge W(x, y) = n$.

Работа сети описывается формально как множество последовательностей срабатываний переходов. Разметка сети — это функция $M: P \rightarrow N$. Переход $t \in T$ может сработать при некоторой разметке M сети, если

$$\forall p: pFt \Rightarrow M(p) \geq F(p, t). \quad (1.3)$$

Срабатывание перехода t при разметке M порождает следующую разметку M' по следующему правилу:

$$\forall p \in P: M'(p) = M(p) - F(p, t) + F(t, p). \quad (1.4)$$

На рисунке 1.2 показано схематичное изображение сети Петри.

В зависимости от структуры сети могут одновременно срабатывать несколько переходов (не имея общих входных мест), их срабатывания являются независимыми действиями и выполняются параллельно. Так можно описывать параллельные процессы. Если же переходы имеют общие места и в данный момент времени готовы сработать (выполняется условие 1.3), то сработать может любой переход, который, сработав, может заблокировать другие, готовые сработать, переходы. Здесь мы имеем недетерминизм.

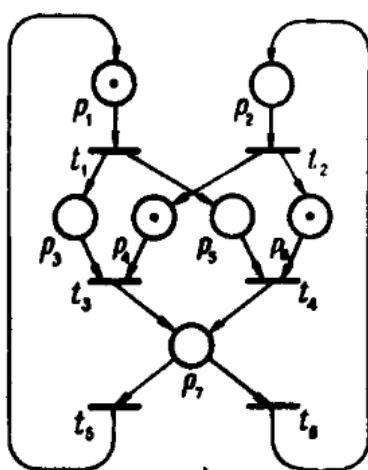


Рисунок 1.2 — Сеть Петри

В настоящее время теория сетей Петри хорошо проработана, имеются готовые способы анализа параметров различных сетей. Однако, на наш взгляд, теория таких сетей является скорее математико-алгоритмической абстракцией, позволяющей нестандартно моделировать параллельные процессы. Такие сущности как переход, процесс, сообщение — все должно быть описано в виде изменений разметок сети. Построить сеть Петри для реальной многопоточной

или взаимодействующей программы сложно, описывать модель в таком виде неприемлемо ни для тестировщиков, ни для программистов, поскольку прямо не связано ни с кодом, ни с требованиями к системе.

1.5.2 Язык SDL

С 1972 года ITU-T и телекоммуникационными компаниями, такими как Ericsson и Motorola, разрабатывается язык SDL (Specification и Description Language – язык описания и спецификаций). Текущая версия стандарта языка — SDL-2000 включает в себя SDL/GR (графическое представление) и SDL/PR (представление при помощи фраз). Для каждого элемента имеется его представление в языковой форме и наоборот.

Язык позволяет описывать графически и в текстовом виде системы, процессы, процедуры, сигналы (получение, отправка), каналы, исключения, структуры данных, классы и интерфейсы. На рисунке 1.3 приведено описание поведения системы при помощи SDL.

Язык применяется для описания взаимодействующих систем и систем реального времени в телекоммуникационных компаниях, для анализа сетевых протоколов. Изначально язык не поддерживал описание объектно-ориентированных систем, однако в новом стандарте языка было добавлено такое описание, притом оно было почти полностью позаимствовано с рассматриваемого далее языка UML (диаграмма классов). По текстовому описанию модели на SDL возможна генерация кода скелета работающей системы.

В принципе, данный язык пригоден для описания многопоточной взаимодействующей системы в целях её дальнейшего тестирования. Однако стандарты языка постоянно менялись, чтобы соответствовать меняющейся ситуации в сфере программной инженерии.

Key SDL-2000 feature: Behaviour

Extended finite state machine:EFSM

- **Start** (symbol) — followed by initialisation going to
- **State** (symbol) — where the machine waits until an
- **Input stimulus** (a **Signal**) of the state as defined by the attached **Input Symbols** is available in the input queue.
- the **Transition**, to the next State consumes the first such signal and interprets its actions such as each **Task** (symbol) or **Decision** (symbol) or **Output** (symbol) sending a signal leading to the **NextState** or a **Stop** (symbol) terminating the process

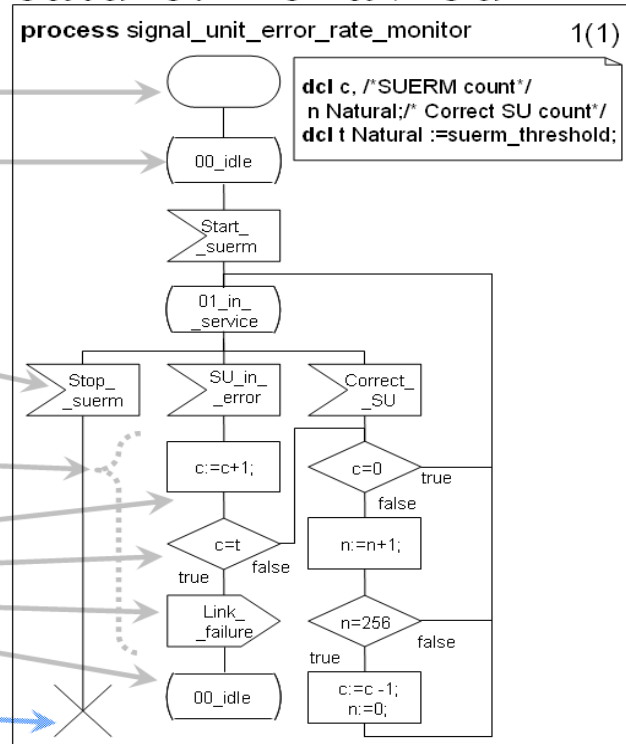


Рисунок 1.3 — Моделирование поведения системы при помощи SDL

Последний стандарт языка был выпущен 9 лет назад, и, судя по последним нововведениям, намечается объединение языка с UML.

1.5.3 Язык UML

Унифицированный язык моделирования (Unified Modeling Language) - стандартизированный язык описания программных (и вообще любых) систем, разработанный в Rational Software, и на сегодня является основным средством и индустриальным стандартом моделирования в области разработки программ; благодаря этому языку, собственно, и стало развиваться направление применения моделирования в области программной инженерии.

UML позволяет описывать поведение системы с разных точек зрения *графически* — статическое поведение (или структурная модель) и динамическое поведение (поведенческая модель), с использованием различных видов диаграмм. Для распределенных систем можно применить диаграмму развертывания, диаграмму компонентов, диаграмму последовательностей, для моделирования работы каждого компонента внутри можно применить диаграмму состоя-

ний. На рисунке 1.4 показаны модели системы в виде диаграмм последовательностей и развертывания с диаграммами компонентов внутри.

UML используется в первую очередь для первоначального проектирования системы её архитекторами, аналитиками совместно с заказчиками. Что касается применения UML в тестировании, то следует отметить работы по UML testing profile - UTP (профиль тестирования в UML). Профиль определяет унифицированные Object Management Group требования к тому, как тестировать системы на основе построения на базе них тестирующих моделей на UML и проектировать тестирующие системы.

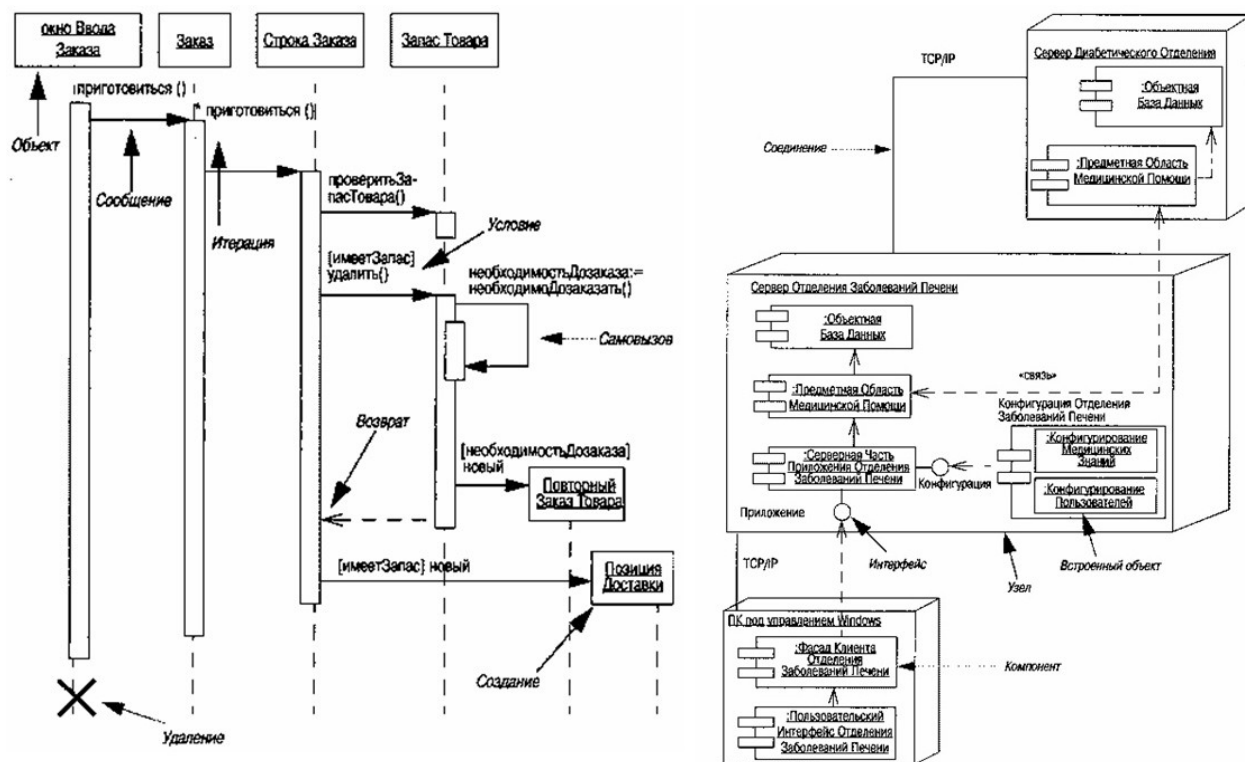


Рисунок 1.4 — Диаграммы UML, моделирующие распределённую систему

Для последующего проведения тестирования спроектированные тесты различного уровня предлагается реализовать с использованием xUnit (для тестирования на уровне компонентов) и, преобразовав диаграммы в описание на языке TTCN-3, провести тестирование с использованием сторонних тестирующих систем на других уровнях (функциональное, интеграционное,

приёмочное тестирование).

В применительно к UTP говорится, что автоматическая генерация тестовых наборов по модели возможна только на основе теории конечных автоматов, на основе диаграмм State Machine и Activity, при этом перспективным направлением, по которому проводятся исследования, сегодня считается «исполняемый UML» .

Однако, на наш взгляд, язык UML пытается быть «одним средством для всего», что перегружает язык различными видами диаграмм, и лучше создать предметно-ориентированную модель, учитывающую все особенности моделируемой системы с небольшим количеством диаграмм.

Задание к главе 1

1. Проанализировать представленные в главе 1 модели и программные средства
2. Обсудить их достоинства и недостатки
3. Предложить свою модель, обосновав ее применение для целей тестирования
4. Определить, какие виды тестирования можно применять на основе предложенной модели
5. Предложить способы тестирования на ее основе.

2 Пример построения математической модели современных распределенных недетерминированных систем на основе автоматов

Данное пособие предполагает изучение свойств компьютерных программ на базе их моделей. Поэтому, в соответствии с понятием модели, необходимо выделить те важные черты рассматриваемого класса компьютерных программ, которых будет необходимо и достаточно для эффективной формализации и исследования программных систем в рамках поставленной задачи – проверки их правильности (тестирования). От выделенных свойств зависит осуществимость поставленных задач и эффективность предлагаемого решения.

2.1 Модели в исследовании свойств алгоритма. Проблема тестирования

Свойства компьютерных программ как эффективных алгоритмов были формализованы при помощи представления их в виде различных эквивалентных математических моделей, например:

- машин Тьюринга;
- частично-рекурсивных функций;
- нормальных алгоритмов Маркова.

В 30ые годы XX века был выдвинут тезис Тьюринга-Черча, который гласит, что “любой алгоритм может быть вычислен при помощи соответствующей машины Тьюринга”. Тезис дан в качестве аксиомы, но само построение существующих компьютеров, когда программе в каждый момент времени доступно ограниченное количество “обозримых ячеек” памяти (память процесса), а доступ к остальной, практически безграничной (динамически выделяемой и дисковой) памяти осуществляется при помощи арифметики указателей (движение головок машины), основано на теории таких машин в соответствии с данным основополагающим тезисом теории алгоритмов.

Используем данную формализацию для определения и доказательства достаточно очевидного факта - неразрешимости проблемы тестирования в общем случае.

Определение 2.1. *Проблема тестирования* заключается в том, возможно

ли для любой программы определить, выполняет ли она заданный алгоритм или имеет ошибки и не выполняет заданный алгоритм?

Допустим, что программа работает в простой однозадачной среде, имеет набор контрольных состояний, и после каждого шага по переходу в следующее состояние осуществляется вывод значений переменных. Если после завершения программы вывод окажется в точности равен ожидаемому, то программа признается верной. Проблема тестирования состоит в том, возможно ли такую проверку сделать для любой программы по ее описанию.

Теорема 2.1. Проблема тестирования неразрешима в рамках существующей теории алгоритмов.

Доказательство. Представим тестируемую программу P в виде машины Тьюринга (она существует в соответствии с тезисом Тьюринга-Черча) и зададим ее геделевским номером $N_{Gedel}(P)$. Тестирование заключается в анализе значений выводимых переменных после перехода в заключительное состояние. Предположим, что проблема тестирования разрешима. Тогда должна существовать машина Тьюринга по переводу цепочки на ленте с выведенными значениями переменных, описанием тестируемой программы в виде ее геделевского номера и ожидаемым результатом - в цепочку с булевым результатом:

$$T_n : X * N_{Gedel}(P) * Y \Rightarrow \begin{cases} 1, P \text{ работает правильно} \\ 0, P \text{ работает неправильно} \end{cases}, \quad (2.1)$$

где X - вывод значений переменных после перехода в заключительное состояние, Y - ожидаемый вывод. Заметим, что X является функцией от исходных данных $X = F(X_{исх})$, поскольку от них зависит результат работы программы на каждом шаге. Тогда данная проблема неразрешима, поскольку является более общим случаем проблемы переводимости, которая является неразрешимой проблемой. ■

Доказанная теорема говорит о том, что в общем случае невозможно доказать правильность любой программы, имея только её описание, притом даже в однозадачной среде для программ с последовательным потоком управления.

Возникает вопрос — как же тогда осуществляется проверка правильности программ сегодня? В настоящее время обычно доказываются правильность специальных видов программ (алгоритмов) с точки зрения какого-то из методов, при этом программа может оказаться правильной с точки зрения данного метода, но неправильной с точки зрения другого. При этом скорее доказываются не правильность программы, а неправильность ее, т.е. ищется какая-либо ошибка в программе. Если таких ошибок не найдено, программа считается правильной до выявления ошибки пользователями в процессе ее эксплуатации. Дейкстра сказал: «Тестирование программ может использоваться для демонстрации наличия ошибок, но оно никогда не покажет их отсутствие».

Использование более сложных механизмов, формальных спецификаций позволяет математически доказать правильность программ, созданных со специальными требованиями к ним, например, в Австрийскому исследовательскому центру NICTA удалось разработать ядро ОС на языке С и доказать его правильность, при этом в работе использовались прототипы на функциональном языке Haskell.

Наличие, кроме самой программы, её модели даёт возможность использовать для проверки правильности дополнительный инструмент, позволяющий сравнительно легко находить ошибки и узкие места в программах, анализирование которых без использования моделей было бы очень сложной задачей. Использование моделей, а не формальных спецификаций, позволит расширить круг возможных тестируемых систем, при этом описание разработанной модели сравнительно несложно и не требует особого образования, однако мы будем говорить о проверке соответствия системы модели, а не доказательство ее безошибочности.

2.2 Распределенные недетерминированные взаимодействующие системы. Постановка задачи на тестирование.

Современные системы часто работают не на одном компьютере, а на их множестве, на встроенных устройствах, при этом обычно состоят из отдельных логических компонентов, и взаимодействие между компонентами системы

осуществляется через локальную сеть организации или глобальную сеть Интернет.

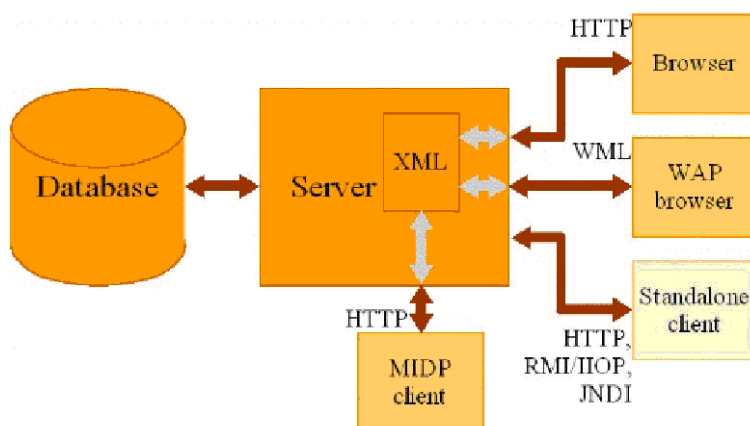


Рисунок 2.1 — Современное распределенное приложение (Sun Microsystems)

На рисунке 2.1 представлена типовая современная программная система, которая имеет базу данных, серверную часть, настольный клиент, «тонкие клиенты» для браузера компьютера и для мобильного-браузера телефона, загружаемое в мобильное устройство приложение.

Дадим основные определения.

Определение 2.2. *Распределенной системой* назовем систему, компоненты которой расположены на нескольких узлах сети или на одном узле с эмуляцией работы по сети. Распределенные системы могут быть реализованы как на основе стандартных средств (сокеты, веб-сервисы, MPI, ...), так и при помощи их комбинаций и собственных протоколов.

Определение 2.3. *Компонент* распределенной системы— это отдельный программный «проект» в терминах интегрированной среды разработки (IDE), программа, которая имеет свою логику работы и может взаимодействовать с другим компонентами. Компонент может находиться на отдельном узле в распределенной системе и указываться при проектировании на UML диаграмме развертывания.

Определение 2.4. *Многопоточной системой* назовем систему, в которой

работает несколько потоков или процессов, выполняющих параллельные действия одновременно. Многопоточность в приложении может создаваться за счет явного создания потоков/процессов операционной системы, неявного создания потоков при помощи библиотеки OpenMP, работой с потоками на разных узлах с помощью средств MPI или же за счет создания легковесных потоков, встроенных в новое поколение функциональных языков.

Определение 2.5. *Недетерминированной* назовем программу, экземпляры которой, запущенные одновременно на одних и тех же данных, в некоторый момент времени демонстрируют разное поведение. Будем считать, что действия такой программы в каждый момент времени носят случайный характер. Конечно, программы проектируются исходя из некоторого предположения о будущей их работе, однако в силу сложности взаимодействующей системы для упрощения моделирования логики работы можно считать ее недетерминированной. Недетерминированность может возникать из-за параллельных действий клиентов в клиент-серверном приложении, возникновения событий в ответ на внешние действия, специальной недетерминированной логикой работы (например, с использованием датчика случайных чисел). При моделировании таких систем можно учитывать не все возможные параметры моделируемой системы, а лишь те, которые интересны с точки зрения поставленных задач.

Современные системы, как правило, являются распределенными, многопоточными и недетерминированными. Обычно, где есть распределенность, там есть и многопоточность и недетерминированность, и такие системы — объект нашего исследования. Следует заметить, что однопоточные системы в однозадачной среде являются подмножеством данного вида систем, и поэтому в соответствии с методом сведения, теорема 2.1 является справедливой для более широкого класса распределенных систем.

Следствие из теоремы 2.1. Проблема тестирования распределенных недетерминированных систем неразрешима в общем виде в рамках существующей теории алгоритмов.

В работе рассмотрено тестирование, прежде всего, применительно к *взаи-*

моделированию между компонентами распределенной системы. При этом мы будем рассматривать тестирование на основе модели методом «белого ящика», т.е. изначально строить модель на базе связности с исходным кодом системы.

2.3 Модель распределенных недетерминированных систем

В настоящем пособии будем строить модель распределенных недетерминированных систем инкрементным методом - последовательно, постепенно усложняя ее шаг за шагом в зависимости от требований к таким системам и приводя доводы в пользу выбранных решений.

2.3.1 Конечный автомат

Конечный автомат — это математическая абстракция, которая представляет собой дискретное изменение состояний объекта в зависимости от его текущего состояния и входных событий, при этом множество состояний конечно. Состояние объекта – это ситуация в его жизни, на протяжении которой он удовлетворяет некоторому условию, осуществляет определенную деятельность или ожидает какого-то события.

Автомат, находясь в некотором состоянии, получает внешнее событие и переходит в какое-то состояние в соответствии с функцией переходов, может быть представлен в виде ориентированного графа (рисунок 2.2).

Рассмотрим применение конечных автоматов для создания модели тестируемой программы.

Сначала опишем модель компонента системы. Определим конечный автомат для симуляции работы компонента системы¹ как пятерку вида:

$$A=(S, s_0, \delta, S_{fin}, D). \quad (2.2)$$

¹ в данном параграфе описываются системы пока без внешнего взаимодействия(как программа в однопоточной среде), далее будет дано определение программы как набора взаимодействующих компонентов, каждый из которых описывается расширенным конечным автоматом.

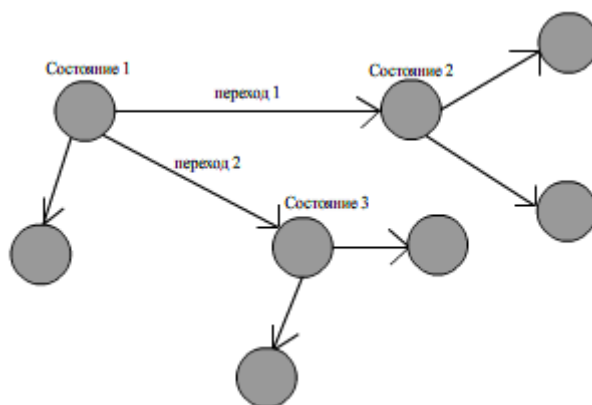


Рисунок 2.2 - Конечный автомат

Здесь S – конечное множество состояний компонента тестируемой программы;

s_0 – начальное состояние системы, точка входа в компонент системы, $s_0 \in S$. Также будем обозначать его $first(A)$.

δ – функция переходов;

S_{fin} – множество заключительных состояний, $S_{fin} \subseteq S$, также можно обозначить $last(A)$;

D – множество действий, алфавит автомата.

В соответствии с методологией объектно-ориентированного программирования, любой объект, в том числе и *программа*, характеризуется состоянием и поведением. Существует много подходов, как именно описать состояние программы. Например, в работе состояние системы — это состояние ключевых переменных, и, когда хотя бы одна из переменных системы изменяется, считается, что система перешла в следующее состояние.

Наша работа посвящена проверке распределенных взаимодействующих систем, в которых прежде всего важно взаимодействие, а при существующих подходах с изменением состояния переменных учет взаимодействия может быть затруднен, поэтому мы будем действовать иначе для определения понятия «состояние».

Состояние можно определить физически и логически. Физическое опре-

деление состояния включает привязку состояния к исходному коду программы, а логическое — параметры состояния, которые его характеризуют в нашей модели. В данном случае мы рассматриваем пока физическое понятие состояния, логическое же в полной мере будет рассмотрено в виде его атрибутов при описании состояния как объекта в приложении Б.

Определение 2.6. *Состояние тестируемой системы* — это участок исходного кода программы (либо компонента программы), который разработчик или архитектор системы определяют как единый логический результат деятельности фрагмента программы.

Реальная распределенная система может состоять из нескольких компонентов, каждый компонент представляет собой проект $System = \bigcup_i Project_i$ (здесь используется теоретико-множественная операция объединения множества проектов в смысле их неупорядоченной совокупности). В интегрированной среде разработки проект - это единая сущность, связывающая относящиеся к нему файлы с исходными кодами и средствами для настройки компилятора для создания объектного кода из набора исходных файлов. Каждый проект, в свою очередь, состоит из нескольких файлов $SrcFile_f$ с исходным кодом ($Project = \bigcup_f SrcFile_f$) на каком-то языке программирования, а каждый файл с исходным кодом — из набора строк Src_i с кодом программы некоторой длины: $SrcFile = \bigcup_{i=1...|SrcFile|} Src_i$ (отметим, что множество строк с исходным кодом упорядочено в соответствии с их положением в файле). Обобщив сказанное для сущностей «проект» и «исходный код» и опустив пока понятие «система» (объединение проектов или компонентов системы в единое целое будет рассмотрено в 2.3.5), можно сказать, что

$$Project = \bigcup_{\substack{f \in SrcFile \\ i=1...|f|}} Src_{fi}. \quad (2.3)$$

(Следует учесть, что в общем случае, на одной линии кода могут находиться несколько операторов программы. Правила хорошего стиля программирования [2] рекомендуют размещать на одной строчке кода один значимый оператор, и

здесь предполагается, что именно так и поступают разработчики).

Тогда состояние $s \in S$ относительно к коду компонента предлагается рассматривать как определяемую разработчиком последовательность $\{p, \dots, r\}$ линий кода Src_{f_i} в исходном файле $f \in SrcFile$, которая, по его мнению, отражает неделимое состояние системы в каждый момент времени:

$$s = \left\{ \bigcup_{i=p \dots r} Src_{f_i} \mid f \in SrcFile, \text{ в файле } f \text{ с линии } p \text{ до линии } r \text{ код — логически значимый} \right\}. \quad (2.4)$$

Для автомата A множество его состояний будем обозначать также как $S(A)$.

Функция переходов δ автомата A задает логику перехода из текущего состояния в следующее («поведение программы»). Поскольку мы изучаем недетерминированные системы, то будем использовать в общем случае недетерминированную функцию перехода.

Итак, первым вариантом функции перехода (в дальнейших параграфах он будет расширен) будет следующее отображение:

$$\delta : S \times D \rightarrow 2^S, \quad (2.5)$$

где 2^S - означает множество всех подмножеств множества состояний S и показывает недетерминированность перехода.

Множество D действий автомата A определяет переходы из состояния в состояние. Предлагается называть «действием» некоторое словесное условие для перехода между состояниями. В графической интерпретации автомата действие — это надпись на дуге между двумя состояниями (рисунок 2.3), $d \in D$.

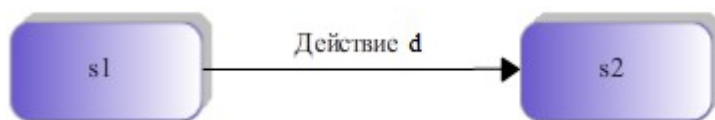


Рисунок 2.3 — Действие из множества D

Рассмотрим теперь, что же является переходом с физической точки зрения для конечного автомата в применении к тестируемой программе.

Определение 2.7. *Переход* из одного состояния в модели компонента

определяется двумя позициями в ее исходном коде: во-первых, это последовательность линий кода, в котором осуществляется какое-либо взаимодействие, с точки зрения разработчика, приводящее к изменению состояния системы (начало перехода), а во-вторых, это линия кода - место начала состояния, в которое осуществляется переход (конец перехода).

Формально, переход $s_x \rightarrow s_y$ в программе из состояния s_x в состояние s_y определяется двумя строками u, v в исходном коде компонента:

$$s_x \rightarrow s_y : \{(u, v) | s_x = (\bigcup_{i=p_x \dots r_x} Src_{f_{1i}}), s_y = (\bigcup_{i=p_y \dots r_y} Src_{f_{2i}}), p_x \leq u \leq r_x, \\ f_1, f_2 \in SrcFile, Src_u \text{ определяет взаимодействие}, v = p_y\}. \quad (2.6)$$

Здесь состояние s_x определяется как последовательность строк кода от p_x до r_x , а состояние s_y — соответственно от p_y до r_y .

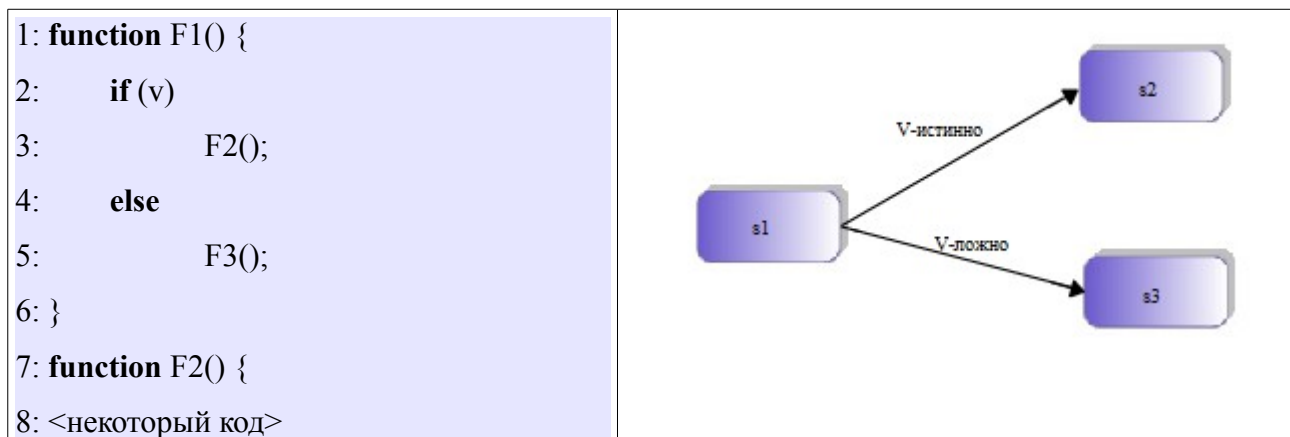
Иначе, переход $s_x \rightarrow s_y$ можно определить строкой в исходном коде, идентифицирующей начало перехода и состоянием, в которое производится переход:

$$s_x \rightarrow s_y : \{(u, s_y) | s_x = (\bigcup_{i=p_x \dots r_x} Src_{f_i}), \\ f \in SrcFile, p_x \leq u \leq r_x, Src_{f_u} \text{ определяет взаимодействие}\}. \quad (2.7)$$

То, что именно определяет взаимодействие, как и при описании состояния, решается разработчиком или архитектором системы.

Пример: Рассмотрим пример привязки состояний и переходов модели компонента системы в виде конечного автомата к исходному коду. Пусть в коде имеется три функции F_1, F_2, F_3 (приведена нумерация строк, рисунок 2.4).

При построении модели разработчик решил, что каждой функции будет соответствовать свое состояние — s_1, s_2, s_3 соответственно.



<pre> 9: } 10: function F3(){ 11: <некоторый код> 12: } </pre>	
--	--

Рисунок 2.4 — Код и автомат

Тогда:

$$D = \{v - \text{истинно}, v - \text{ложно}\}; S = \{s_1, s_2, s_3\};$$

s_1 находится с 1 по 6 строках кода; s_2 находится с 7 по 9 строках кода;

s_3 находится с 10 по 12 строке кода.

Переходы определим согласно формуле 2.6: $s_1 \rightarrow s_2 : (3, 7); s_1 \rightarrow s_3 : (5, 10)$.

После описания перехода возникает вопрос о перекрытии состояний. С точки зрения логики, состояния перекрываться не должны. Однако структура обычной программы такова, что описать состояния и переходы между ними без перекрытия достаточно сложно. Например, если программа состоит из условного оператора и больших блоков кода:

```

if (<условие>) {
    <действия>
    ...
} else {
    <действия>
    ...
}

```

Если объявить все это за состояние «проверка условия» и осуществлять переход в состояния «действия, в зависимости от того, что условие выполнено» и «действия, в зависимости от того, что условие не выполнено» и, если конечно код между { и } не представляет собой вызов функции (как в предыдущем примере), то в данном случае мы можем видеть перекрытие состояний, т.к. состояние «проверка условия» еще не завершилось, а «условие выполнено» уже должно начаться. Поэтому предлагается разрешить описывать состояния внутри других состояний. Описания состояния внутри другого - элемент физической

привязки к исходному коду, и в самой модели и функциях переходов автомата данный момент нигде не используется.

Рассмотрим количественный критерий качества описания состояний (переходы определяются из состояний, поэтому важно именно описание состояний). Можно предложить критерий эффективности их описания- метрику [5] покрытия кода состояниями, её можно определять как для отдельного файла (формула 2.8), так и для проекта (формула 2.9), а также среднее значение метрик файлов проекта (формула 2.10):

$$Metric_{cover}(f) = \frac{\sum_{S_i \in f} |S_i|}{|f|}, f \in SrcFile \quad (2.8)$$

$$Metric_{cover}(project) = \frac{\sum_{f_i \in project} \sum_{S_j \in f_i} |S_j|}{\sum_{f_i \in project} |f_i|} \quad (2.9)$$

$$\overline{Metric_{cover}}(project) = \frac{\sum_{f_i \in Project} Metric_{cover}(f_i)}{|project|}. \quad (2.10)$$

Здесь $|S_i|$ - длина кода состояния, $|f|$ - длина файла с исходным кодом, $|project|$ - мощность множества проекта, т.е. число файлов с исходным кодом в проекте. Для подсчета метрик *вложенные состояния не учитываются*, иначе одни и те же линии исходного кода будут посчитаны несколько раз.

Рассмотрим далее способ повышения степени абстракции за счет логической группировки состояний, так чтобы в модели можно было эту группу считать за одно состояние, внутреннее поведение которого является сложным и может быть скрыто.

Определение 2.8. Группой состояний с разрешенными зависимостями расширенного автомата A назовем автомат A' , определенный на подмножестве состояний $A' = A|_{S=S'}, S' \subseteq S$, в который входят такие состояния S' , что для любого состояния из S' в результате действия функции δ^1 из этого состояния (описанной в данном параграфе и с изменениями в последующих)

¹ А также функции γ , которая будет введена далее

переходы возможны только в состояния из S' , кроме переходов из специально выделенных заключительных состояний подавтомата. Иными словами, попав в состояние из группы состояний с разрешенными зависимостями, система не перейдет ни в какое другое состояние и не затронет в результате взаимодействия никакого состояния, кроме как состояния этого подавтомата, и все исходящие связи с другими состояниями автомата возможны только через заключительные состояния подавтомата. Далее такую группу состояний с выделенными конечными и начальным состоянием будем называть *подавтоматом* и обозначить как $A' \subseteq A$.

Таким образом, подавтомат A' автомата A ($A' \subseteq A$) кроме свойств автомата, имеет выделенное начальное состояния $s'_0 \in S$ и набор своих конечных состояний $S'_{fin} \subseteq S$, т.е. $A' = (A | S = S', s'_0, S'_{fin})$.

Заметим, что подавтомат с разрешенными зависимостями во всех формулах настоящей работы можно заменять на состояние на более высоком уровне абстракции (рисунок 2.5), назовем его «сверхсостояние». При этом все входящие в начальное состояние подавтомата переходы — это переходы в такое состояние, а все исходящие переходы из специально выделенных заключительных состояний подавтомата — это переходы из сверхсостояния.

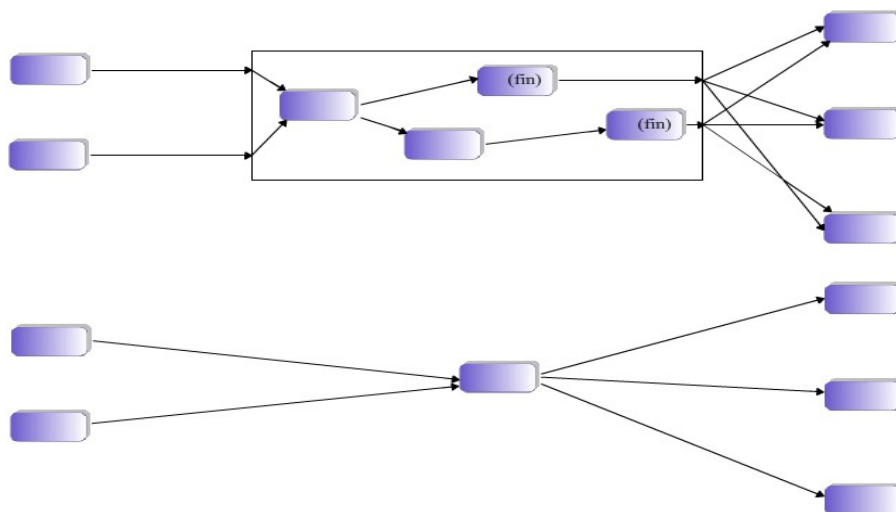


Рисунок 2.5 — Подавтомат как состояние

2.3.2 Конечный автомат с вероятностными переходами

Для моделирования недетерминированных программ рассмотренный в 2.3.1 конечный автомат хоть и приспособлен, но мало. Да, он имеет недетерминированную функцию переходов, однако переход осуществляется только по словесному описанию, по «действию». Для анализа правильности сложных систем целесообразно расширить модель программы в виде автомата путем добавления в функцию переходов элемента случайности — вероятности перехода из заданного состояния в другое заданное. После этого наш автомат будет считаться вероятностным конечным автоматом.

Такие автоматы применяются, например, для моделирования защищенных автоматизированных систем, где элемент случайности — это атака злоумышленника на систему. Мы будем применять автомат с вероятностными переходами для моделирования сложной взаимодействующей системы, где элемент случайности — это переход в следующее состояние под воздействием неизвестной природы. Использование вероятностных переходов делает наш автомат истинно недетерминированным.

Например, для моделирования оператора $if(A)B; else C$, переход на B осуществляется с вероятностью $P(A)$ и на C - с вероятностью $P(\neg A) = 1 - P(A)$.

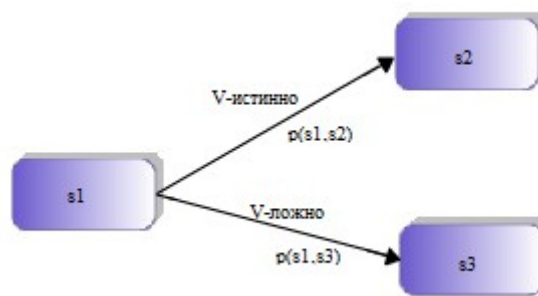


Рисунок 2.6 — Вероятностный переход

Таким образом, дополним формулу 2.5:

$$\delta : S \times D \times P \rightarrow 2^S, \quad P \subseteq \mathbb{R}. \quad (2.11)$$

Это означает, что в модели программы происходят недетерминированные переходы из состояния $s_1 \in S$ с вероятностью $p \in P$ с указанием действия

$d \in D$ в состояние $s_2 \in S$. Вероятность перехода определяется для каждой пары состояний (состояние, из которого осуществляется переход; состояние, в которое осуществляется переход на следующем шаге). Здесь можно говорить о матрице переходных вероятностей, $P_{ij} = \|p_{ij}\|$, где $1 \leq i, j \leq |S|$ и каждый элемент p_{ij} матрицы — вероятность попасть из состояния i в состояние j .

Естественно предполагать также, что :

$$\begin{aligned} 0 \leq p_{ij} \leq 1, \\ \forall i, 1 \leq i \leq |S|: \sum_{j=1 \dots |S|} p_{ij} = 1. \end{aligned} \quad (2.12)$$

Матрица P_{ij} является стохастической матрицей.

Вероятности перехода из состояния в состояние могут быть заданы разработчиками системы и откорректированы на основе данных о переходах между состояниями после запуска системы.

Определение 2.9. *Априорными вероятностями* тестируемой системы назовем вероятности переходов между состояниями вероятностного конечного автомата, которые заданы разработчиками/архитекторами системы исходя из предположений о работе системы. Априорные вероятности определяют ожидаемое поведение системы.

Определение 2.10. *Апостериорными вероятностями* тестируемой системы назовем вероятности переходов модели, вычисленные после работы программы на основании статистики переходов. Апостериорные вероятности определяют реальное поведение системы после её запуска.

Апостериорные вероятности вычисляются в процессе работы системы, чем обеспечивается связь между моделью и системой (модель \rightarrow система и система \rightarrow модель).

Можно отметить, что апостериорные (вычисленные) вероятности могут быть использованы как априорные при следующем запуске моделирования.

Заметим, что по построению функции переходов автомата мы осуществляем переход из текущего состояния в следующее без учета предыдущих посещённых состояний (в работе мы рассматриваем переходы вида $S \times D \times P \rightarrow 2^S$,

но не вида $\underbrace{S \times S \times \dots \times S}_n \times D \times P \rightarrow 2^S$, где n - максимальное число запоминаемых автоматом посещенных ранее состояний).

Проведем небольшое исследование, всегда ли можно осуществить разметку состояний, базируясь на факте, что переход определяется только текущим состоянием и не зависит от предыдущих? Вспомним, что состояние по формуле 2.4 определяется подмножеством строк в исходном коде программы. Сравним две возможности:

- а) если до перехода в текущее состояние автомат уже побывал в некоторых состояниях;
- б) если до перехода в текущее состояние автомат не побывал ни в каких состояниях.

С точки зрения нашей функции переходов (2.11), данные две возможности равны при выборе следующего состояния. Но после привязки состояний и выполнения программы в первом случае пребывание в некотором количестве состояний может привести к изменению переменных системы («побочный эффект»). Однако мы при моделировании не учитываем изменение переменных и полагаемся полностью на вероятностную модель — из любого состояния переход сопровождается вероятностью этого перехода - заданной или вычисляемой, поэтому мы можем говорить о вероятностном переходе в зависимости только от текущего состояния. Нахождение автомата в предшествующих состояниях влияет на модель через апостериорные вероятности при динамическом тестировании по модели (параграф 3.6). Можно вспомнить также, что мы рассматриваем в первую очередь взаимодействующие системы, а они, как правило, построены на схеме «запрос-ответ» без «побочных эффектов». Кроме того, следует отметить, что новое поколение функциональных языков предполагает отсутствие «побочного эффекта» у любой вычислительной конструкции, поскольку это создает проблемы в распараллеливании программ на нескольких процессорах и компьютерных узлах.

Таким образом, модель зависит от текущего состояния и не зависит от

предыстории, и мы можем описать поведение нашей модели как марковский случайный процесс с дискретными состояниями и дискретным временем (однородная цепь Маркова).

Пусть $\zeta(t)$ - состояние автомата в момент времени t , $S_i \Rightarrow S_j$ означает переход из состояния S_i в состояние S_j , $P(S_i \Rightarrow S_j | \zeta(t) = S_m)$ - вероятность перехода из состояния S_i в S_j , при условии, что в момент времени t автомат находится в состоянии S_m . Марковское свойство модели равносильно следующему: для любого фиксированного текущего состояния $S_i \in S$ в момент времени t любые предшествующее состояние S_j и следующее состояние S_k условно независимы, т. е.

$$P(S_j \Rightarrow S_k | \zeta(t) = S_i) = P(S_j \Rightarrow S_i | \zeta(t) = S_i) * P(S_i \Rightarrow S_k | \zeta(t) = S_i). \quad (2.13).$$

2.3.3 Конечный автомат с вероятностными переходами и обработкой событий и исключений

Современные программы построены не на линейном вычислительном процессе, который представляет собой просто переходы из состояния в состояние, а являются событийно-ориентированными. Движение курсора мыши, нажатие на кнопку, изменение состояний объектов в программе - могут генерировать события, и их обработка представляет собой реакцию программы на такие воздействия, по «голливудскому принципу» (Don't call us, we will call you – не звоните нам, мы сами вам позвоним). Иначе говоря, нет необходимости опрашивать объект об изменении его параметров — он сам «сообщит» нам об этом, инициировав событие.

Событийно-ориентированными прежде всего являются системы с графическим пользовательским интерфейсом, системы с сетевым взаимодействием, web-приложения, т.е. современные программные системы, модель которых предлагается в данном исследовании. Расширение POSIX Realtime extension для программ под Unix, работающих в режиме реального времени, также использует сигналы (как события) и их обработчики.

Все современные языки программирования имеют механизм обработки

исключительных ситуаций, обычно возникающих в программе в результате ошибок времени выполнения и позволяющих без потерь для работы программы корректно обработать ошибку и продолжить выполнение, при условии, что разработчик намеренно предусмотрел данную ситуацию. Исключение и событие похожи, поскольку исключение — это событие в ответ на ошибку. Обработчик события или исключения — это та функция, которая вызывается при их возникновении.

Для моделирования работы событийной системы множество состояний разделим на

- состояния, которые достижимы в компоненте в результате его последовательной работы, - активные состояния (S_s),
- состояния, которые достижимы как модели обработчиков событий или исключений, - пассивные состояния (S_E):

$$S = S_s \cup S_E. \quad (2.14)$$

Внесем изменение в функцию переходов (формулу 2.11):

$$\delta: S \times D \times P \rightarrow 2^{S_s}. \quad (2.15)$$

Рассмотрим новое множество E - множество, описывающее события или исключения (E от англ. Event/Exception – Событие/Исключение):

$$E = S_E \times P \times W_{flag}. \quad (2.16)$$

Событие или исключение определяется:

- 1) состоянием из множества S_E - тем состоянием, куда будет осуществлен переход при возникновении события. Данное состояние является $first(A_E)$, где A_E - подавтомат, моделирующий функцию — обработчик события/исключения;
- 2) вероятностью того, что сработает именно данный обработчик среди всех обработчиков (обозначим такую вероятность с индексом «E»: $p_E \in P$);
- 3) а также флагом типа boolean W_{flag} ($W_{flag} = \{true, false\}$), показывающим, что переход к обработке данного события определяет или нет пере-

ход в так называемое ошибочное состояние.

Определение 2.11. *Ошибочным состоянием* системы назовем такое состояние, попадание в которое говорит о возникновении какой-либо контролируемой ошибки в программе с точки зрения разработчика. Частое нахождение системы в таких состояниях говорит о ненадежности окружающей среды программы. С другой стороны, поскольку ошибочное состояние может объявляться у обработчика событий или исключений (скорее, у исключений), то описание такого состояния уже само по себе является указанием со стороны разработчика на возможность ошибочной ситуации.

Для формализации событий в определение расширенного автомата добавим еще один элемент γ — функцию переходов «по ребрам». Данная функция предполагает зависимость возможности возникновения события от ребра в представлении расширенного автомата в виде графа, т.е. перехода из состояния в состояние, и определяет следующее отображение:

$$\gamma : S \times S \times \Delta t \rightarrow E^* . \quad (2.17)$$

Формула 2.17 означает, что при переходе из состояния $s_1 \in S$ в состояние $s_2 \in S$ в результате некоторого возможного таймаута Δt (превышения допустимого времени между началом и концом перехода) могут возникнуть некоторые события или исключительные ситуации, каждое из которых предполагает переход в состояние-обработчик с заданной вероятностью и, возможно, установку флага ошибки.

Часто требуется продолжить работу программы после обработки события с прерванного в результате работы обработчика места. В данном случае обработчик события должен осуществлять переход в то состояние, в которое вел переход согласно функции γ перед срабатыванием события. Для моделирования такого действия вводится флаг продолжения работы с прерванного места и состояние, в которое вел прерванный переход. Поэтому в формулу 2.16 внесем следующие изменения:

$$E = S_E \times P_E \times W_{flag} \times I_{flag} \times S_{return} . \quad (2.18)$$

Здесь I_{flag} – логический флаг, определяющий необходимость перехода в прерванное обработкой события состояние; $S_{return} \in S$ – состояние, куда нужно вернуться после обработки события. При этом переход в данное состояние осуществляется при установленном флаге I_{flag} неявно, т. е. не описывается в модели, и предполагается, что при переходе подавтомата обработчика события в заключительное состояние выполняется переход в состояние S_{return} с вероятностью, равной 1. Добавим множество исключений и событий, а также функцию переходов «по ребрам» в определение автомата:

$$A = (S, s_0, \delta, \gamma, S_{fin}, D, E). \quad (2.19)$$

Рассмотрим пример обработки исключительных ситуаций в коде на языке Java (приведены номера строк):


```

1      try {
2          <Код>
3      } catch (ArithmeticException ex1) {
4          <Код обработки исключения, выполнится, если в <код>
5          произойдет ошибка в арифметике>
6      } catch (Exception ex2) {
7          <Код обработки исключения, выполнится, если в <код>
8          произойдет ошибка любого другого типа>
9      } finally {
10         <выполнится в любом случае>
11     }
```

Выделим состояния:

s_1 – код на строке 2 ; s_2 – код обработки исключения `ArithmeticException` на строках 4-5 ; s_3 – код обработки исключения `Exception` на строках 7-8 ;

s_4 – код, выполняющийся в любом случае на строке 10.

Схема соответствующего автомата с обработкой исключений представлена на рисунке 2.7. Здесь и далее знаком «» будем обозначать наступление события/исключения.

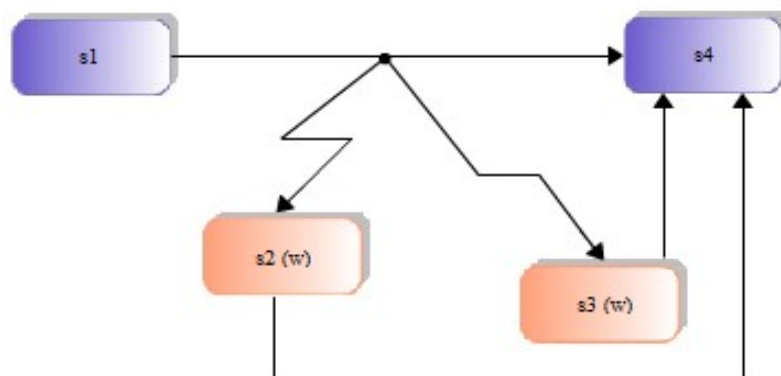


Рисунок 2.7 – Автомат с обработкой исключений

Введенные ограничения на возникновения события только во время переходов из состояния в состояние не являются существенными. Если событие возникает внутри состояния, то можно либо разбить код на несколько последовательных состояний и определить события при переходе из этих состояний в последующие, либо сделать переход из состояния в это же состояние (петлю) и определить возникновение событий во время этого перехода.

Рассмотрим теперь процесс моделирования по системе диалоговой программы. Для ОС Windows, на каком бы языке программирования приложение ни было бы реализовано, все взаимодействие на уровне пользовательского интерфейса транслируется в код, который получает сообщения из очереди сообщений операционной системы, и отправляет нужные сообщения в очередь приложения. Под сообщениями понимается извещение о некотором имевшем место событии, посылаемом системой Windows в адрес приложения. Сообщений из очереди транслируется в события более высокого уровня, обрабатываемые пользователем.

Рассмотрим главный класс приложения Windows.Forms на языке C#:

```

static class Program
{
    [STAThread]
    static void Main()
    {

```

```

Application.EnableVisualStyles();
Application.SetCompatibleTextRenderingDefault(false);
Application.Run(new FormWin());
}
}

```

Приложение запускается методом Run, который создает оконную форму и начинает обрабатывать события. Код, реализующий оконную форму класса FormWin с кнопкой button на ней, имеет вид:

```

public partial class FormWin : Form
{
    public FormWin()
    {
        InitializeComponent();
    }
    private void button_Click(object sender, EventArgs e)
    {
        <пользовательский код обработки нажатия на кнопку>
    }
}

```

Выделим состояния:

s_1 – Код инициализации формы (выполняется при вызове new FormWin());

s_2 – Application.Run();

s_3 – Код обработки события нажатия на кнопку - button_Click().

Автомат, соответствующий данному коду с обработкой возникновения события, представлен на рисунке 2.8.

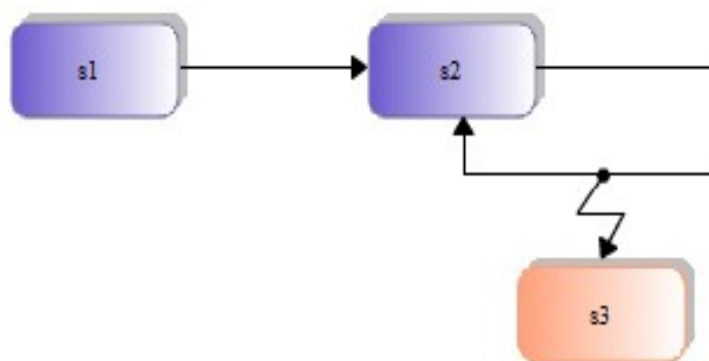


Рисунок 2.8 — Автомат с обработкой события

Рассмотрим вероятности переходов при обработке событий/исключений. Естественно предположить, что при учете возникновения событий или исключений вероятность перехода в следующее состояние уменьшится, поскольку во время перехода могут сработать эти события и/или исключения.

Предположим, что происходит переход из состояния s_i в s_j , вероятность перехода при этом p_{ij} . Мы можем говорить о вероятности корректного «выхода» из состояния s_i , которая не равна вероятности «выхода из состояния s_i и захода в состояние s_j », кроме того, возможно возникновение $k \geq 0$ событий или исключительных ситуаций $E^k = E_1 \times \dots \times E_k$ с вероятностями их возникновения $p_{E_m}, m = 1..k$.

По формуле произведения вероятностей вероятность перехода компонента системы из состояния s_i в s_j с учетом событий и исключений будет равна:

$$p_{ij}^E = 1 - \sum_{m=1..k} p_{ij} \cdot p_{E_m}. \quad (2.20)$$

Рисунок 2.9 поясняет формулу 2.20.

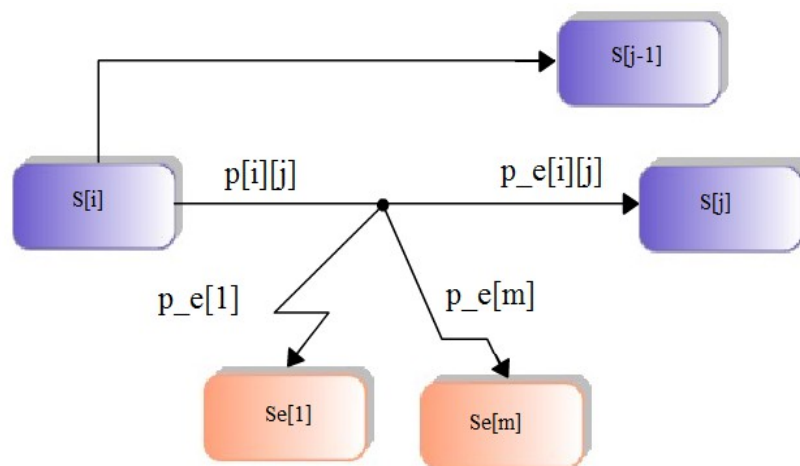


Рисунок 2.9 — Скорректированная вероятность попадания в состояние

События можно рассматривать не только для отдельно взятых состояний, но и в целом для подавтомата. Определим *событие для подавтомата* как событие, которое может произойти при всех переходах между состояниями подавто-

мата (рисунок 2.10).

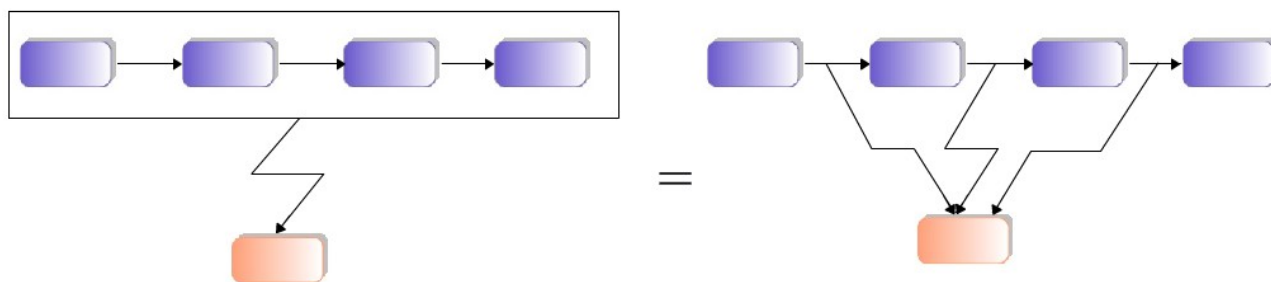


Рисунок 2.10 — События для подавтомата

Событие для подавтомата — это «копирование» события ко всем переходам в подавтомате. Состояние возврата (формула 2.18) определяет переход в случае описания в модели возникновение события только у одного из переходов подавтомата.

2.3.4 Расширение автомата для поддержки моделирования многопоточных приложений

Предложенный в 2.3.3 расширенный вероятностный конечный автомат не позволяет осуществлять более одного перехода за один шаг и моделировать работу нескольких агентов (поток) одновременно, т.е. не обеспечивает моделирование современных многопоточных систем. Дополним описание модели средствами взаимодействия в рамках одного компонента.

2.3.4.1 Параллелизм

Известно, что в современных взаимодействующих системах параллелизм достигается за счет таких сущностей, как параллельные процессы или потоки. Строго говоря, процессы или потоки имеют отличия в общности данных (потоки имеют дело с общими данными программы, а процессы - нет, они взаимодействуют только с разделяемыми данными со стороны ОС). Поскольку мы моделируем взаимодействие, а не анализ данных, то не имеет принципиального значения — процесс или поток, главное в том, что в системе выполняются параллельные действия.

Определим, что такое параллельно выполняющиеся действия примени-

тельно к моделям компонентов системы в виде расширенного конечного автомата.

Определение 2.12. Множество состояний $Next(S_{start})$, достижимых из заданного состояния S_{start} , определяется рекурсивно:

$$\begin{aligned} Next^0(s_{start}) &= \{s_{start}\} & Next^1(s_{start}) &= \{s \in S \mid P(s_{start}, s) \neq 0\} \\ &\dots \\ Next^{n+1}(s_{start}) &= Next^n(s_{start}) \cup \{s \in S \mid P(s', s) \neq 0, s' \in Next^n(s_{start})\}. \end{aligned} \quad (2.21)$$

Здесь $P(s_{start}, s)$ - вероятность перехода из состояния s_{start} в состояние s , согласно функции переходов. Вероятность вычисляется с учетом прямых переходов и переходов после возникновения событий (по формуле 2.20).

Определение 2.13. Отношением порядка \leq_s над множеством состояний расширенного автомата S назовем двухместное отношение над $S \times S$:

$$s_1 \leq_s s_2 \Leftrightarrow \exists k > 0 : s_2 \in Next^k(s_1). \quad (2.22)$$

Определение 2.14. Параллельными действиями в системе назовем действия по переходу между состояниями, которые выполняются неупорядоченно и одновременно.

Неупорядоченность. Действия подавтоматов $A' = A|_{S=S'}, S' \subseteq S$ и $A'' = A|_{S=S''}, S'' \subseteq S$ автомата A не упорядочены, если

$$\forall s_1 \in S', s_2 \in S'' : \neg (s_1 \leq_s s_2 \vee s_2 \leq_s s_1). \quad (2.23)$$

Действия n подавтоматов автомата A не упорядочены, если все они попарно не упорядочены.

Одновременность (глобальное состояние). Действия n неупорядоченных автоматов выполняются одновременно, если в каждый момент времени каждый из подавтоматов, выполняющих действия, находится в своем состоянии из множества локальных состояний $S_{L_i}, i=1..n$, а множество общих глобальных состояний автомата можно представить как $S = S_{L_1} \times S_{L_2} \times \dots \times S_{L_n}$.

На рисунке 2.11 показана модель параллельных процессов и их локальных состояний. Система находится в объединении состояний $S_{L_12} \in S_{L_1}, S_{L_21} \in S_{L_2}$

$$S_{L_1} \subseteq S \ \& \ S_{L_2} \subseteq S \ \& \ S_{L_3} \subseteq S.$$

Об общем, глобальном состоянии расширенного автомата при рассмотрении параллельных систем говорить обычно не приходится, поскольку его определить достаточно сложно. Можно предположить последовательную нумерацию локальных состояний и основанную на переборе номеров возможных состояний нумерацию соответствующего глобального состояния в виде S_{l_1, l_2, \dots, l_n} , где $l_1 \in 1..|S_{L_1}|, \dots, l_n \in 1..|S_{L_n}|$.

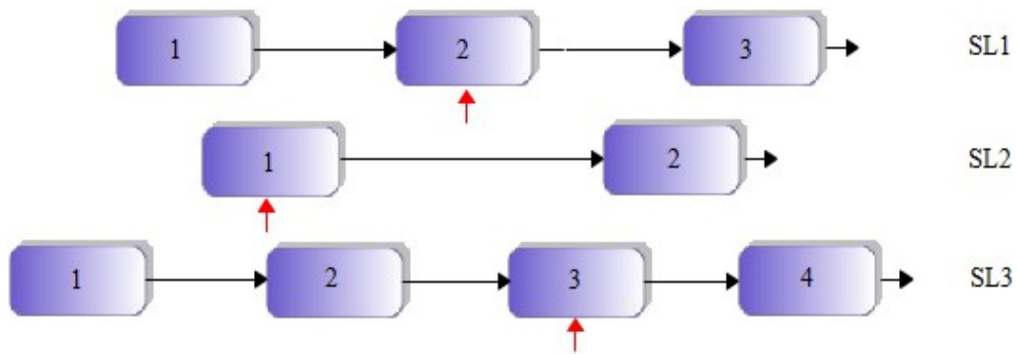


Рисунок 2.11 — Локальные состояния

Однако мощность множества глобальных состояний автомата, выполняющего параллельные действия, с учетом локальных состояний подавтоматов, велика (для перебора вариантов, $|S_{l_1, l_2, \dots, l_n}| = \prod_{i=1}^n |S_{L_i}|$). Поэтому обычно глобальные состояния в взаимодействующих системах не рассматриваются. Мы будем рассматривать моделирование переходов в *локальных* состояниях внутри компонента, способы создания подавтоматов, выполняющих параллельные действия, и средства их взаимодействия и синхронизации.

Определение 2.15. *Потоком* в модели компонента, заданного расширенным автоматом A , назовем параллельно выполняющиеся действия, заданные:

- 1 *Потоком-родителем* T_{parent} , создавшим данный поток. Родитель есть у всех потоков, создаваемых в системе, кроме главного потока приложения ($Main$).
- 2 Подавтоматом $A' \subseteq A$, который отвечает за логику работы потока. Подав-

томат определен на группе состояний $S' \in S$ ($A' = A|_{S=S'}$), и эти состояния и взаимодействие между ними определяют работу потока.

3 Именем потока *Name* (строковый тип).

Таким образом, для дочернего и родительского потока имеем:

$$T_{Name} = (T_{parent}, A' = A|_{S=S'}, Name), \quad (2.24)$$

$$T_{Main} = ((\emptyset, \emptyset, \emptyset), A, 'Main'). \quad (2.25)$$

Поток начинает работу в состоянии $first(A')$ и заканчивает в одном из заключительных состояний подаutomата A' или заключительных состояниях автомата A . Рисунок 2.12 демонстрирует потоки, подаutomаты и родительские потоки.

2.3.4.2 Операция создания потока и понятие кратности

Для создания потока необходимо выполнить особое действие по переходу из состояния в несколько состояний, создав при этом несколько потоков. Мы назовем такое действие «вилкой» или операцией «fork» (англ. вилка), по аналогии с популярной системной функцией созданием копии процесса в UNIX системах (строго говоря мы поступаем совсем не так, как одноименная функция - не создаем копию процесса, а создаем новый поток, находясь в каком-либо состоянии).

Определение 2.16. Здесь и далее в тексте под словом «операция» мы имеем ввиду логически обособленное действие по переходу из состояния расширенного автомата в его другое состояние с заданной вероятностью, выполняющее помимо перехода некоторые действия со множествами расширенного автомата. Операция является подмножеством функции перехода.

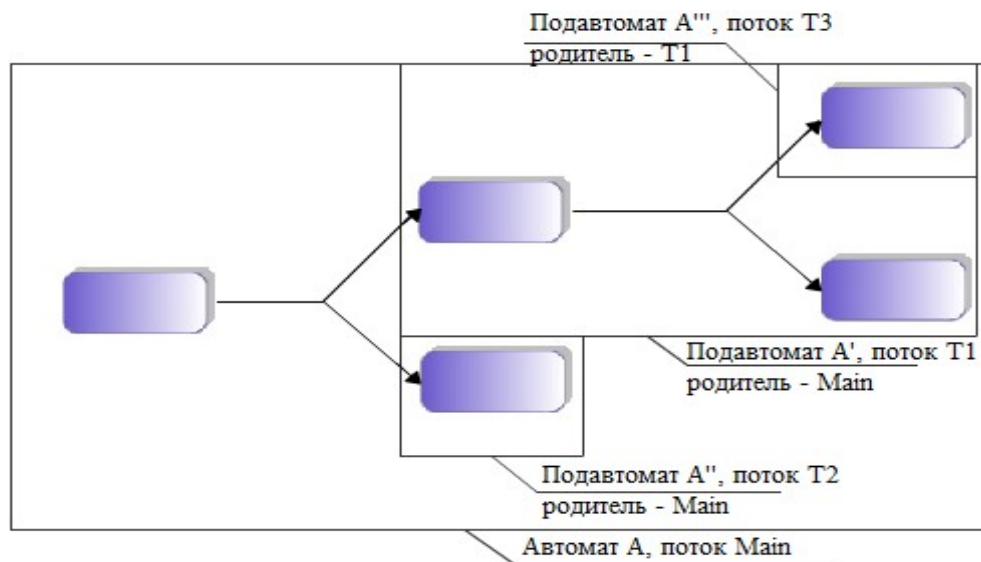


Рисунок 2.12 – Поток, подавтоматы и родители

Операция создания потоков включается в функцию переходов δ :

$$\delta \ni fork : P \times S \rightarrow (S \times T)^+ . \quad (2.26)$$

Будем обозначать эту операцию «».

Поясним различие однопоточного и многопоточного переходов (рисунок 2.13).

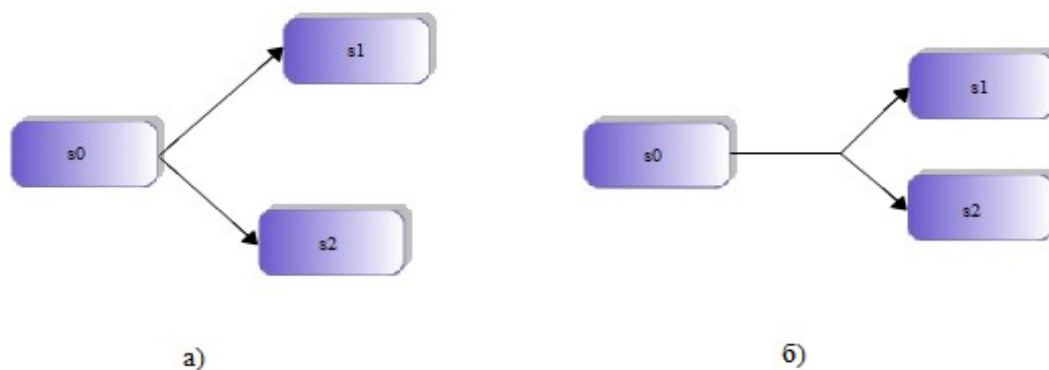


Рисунок 2.13 — однопоточный и многопоточный переходы

В случае обычного перехода - а) производится переход из состояния s_0 в состояние $s_1 \vee s_2$, то есть либо в s_1 , либо в s_2 ; в случае же перехода с созданием потока - б) из состояния s_0 осуществляется переход в $s_1 \& s_2$, то есть и в s_1 и в s_2 . При выполнении операции «создать поток» главный поток продолжает работу как один из потоков.

Следует отметить, что можно создавать как потоки, выполняющие равнозначные действия (например, в программе создается несколько потоков как экземпляров одного и того же класса), так и совершенно разные. Разные действия моделируются разными поавтоматами, одинаковые же — одним подавтоматом (при этом код может быть написан неэффективно и дублирован, занимать разные строки в исходном файле и выполнять одинаковые действия, в этом случае при моделировании возможно как дублирование состояний, так и использование их повторно).

Рассмотрим создание потока в цикле (рисунок 2.14).

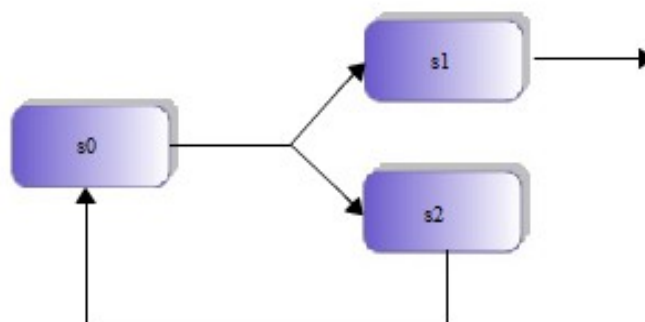


Рисунок 2.14 – Создание потока в цикле

В данном случае в состоянии s_0 осуществляется операция «fork», при этом создается поток с $first(S') = s_1$, который начинает выполнять какие-либо действия; кроме этого, основной поток (или поток-родитель) продолжает работу в s_2 и через конечное число состояний вновь попадает в точку разделения потоков (состояние s_0), после чего еще один новый поток с $first(S') = s_1$ может быть создан вновь. Таким образом, может быть создано некоторое количество потоков с $first(S') = s_1$, параллельно выполняющих одинаковые действия. Это приводит нас к определению *кратности* состояния и кратности потока.

Определение 2.17. *Кратностью состояния* называется целое число $n \in \mathbb{N}$, которое показывает, что в данном состоянии может находиться одновременно не более n потоков. По умолчанию кратность состояния принимается равной 1.

Определение 2.18. *Кратностью $N(T)$ потока T назовем кратность его стартового состояния:*

$$N(T) = N(\text{First}(S')),$$

если $T = (T_{\text{parent}}, A' = A|_{S=S'}, \text{Name})$.

Переход в состояние с другой кратностью осуществляется только при создании потока(ов), причем первоначально справедливо соотношение:

$$\forall s' \in S': N(s') \geq N(\text{first}(A')). \quad (2.27)$$

Данное соотношение означает, что кратность любого состояния подавтомата, реализующего логику работы потока, может быть либо кратностью его первого состояния, либо может быть большее ее, поскольку в некотором состоянии подавтомата может быть проведена операция создания другого потока, что только увеличит кратность.

Тут возникнет вопрос — если какой-либо поток кратности $N > 1$ создаст еще один поток с кратностью 1, то будет ли внутри вновь созданного потока кратность равна 1? Таким образом, мы приходим к понятию *кратности состояния (потока) с точки зрения какого-либо потока*.

Кратность состояния s_{start} с точки зрения какого-либо потока T (обозначим $N(T, s_{\text{start}})$) вычисляется по формуле:

$$N(T, s_{\text{start}}) = \prod_{T_{\text{inner}} \text{ создается по пути } s' \rightarrow s_{\text{start}}} N(T_{\text{inner}}). \quad (2.28)$$

Будем действовать по следующему алгоритму: стартовым объявляется состояние s_{start} и на каждом шаге осуществляется переход назад (в соответствии с обратной функцией переходов δ^{-1}) до тех пор, пока не будет достигнуто состояние $s' = \text{first}(S')$, при $T = (A', T_{\text{parent}}, \text{name})$, $A' = A|_{S=S'}$, при этом при обратном проходе через операцию создания потока происходит перемножения кратностей состояний, которые начинают подавтоматы потоков.

Аналогично определяется и кратность потока T с точки зрения другого потока T_s ($N(T, T_s)$), поскольку кратностью потока является кратность его первого состояния. Естественно, в цепочке событий поток T должен быть со-

здан после T_s .

Вместе с понятием кратностей состояния и потока предлагается ввести и понятие перехода по кратности. Можно заметить, что на рисунке 2.14 состояние s_1 является кратным, однако вечно создавать потоки невозможно, поскольку их число в операционной системе ограничено, и надо промоделировать выход из операции создания потоков. Другой вариант перехода по достигнутой кратности, который можно привести в качестве примера, - это модель чата, когда сервер ждет подключения минимум двух клиентов (в модели — создания потока кратности 2, инкапсулирующего поведение клиента), чтобы после этого был разрешен обмен сообщениями между ними.

Определение 2.19. *Переход по кратности n осуществляется, либо когда кратность текущего потока равна n : $N(T_{current})=n$, либо когда кратность заданного потока T с точки зрения текущего потока равна n : $N(T, T_{current})=n$ (это не равнозначные понятия, а варианты перехода по кратности).*

На рисунке 2.15 показан измененный пример рисунка 2.14, в который добавлен выход из процесса создания кратных потоков по достижимости их числа, равного n .

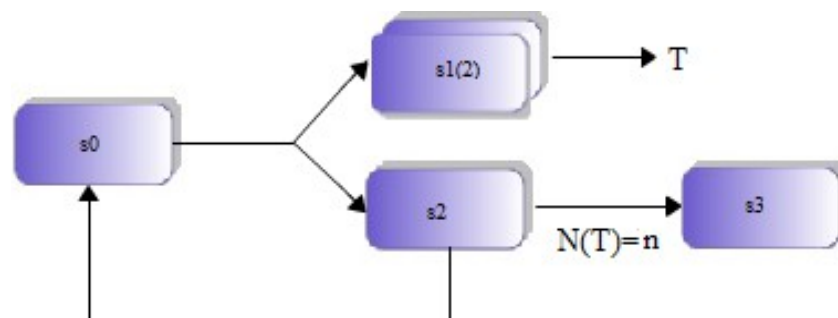
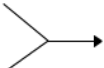


Рисунок 2.15 — Переход по кратности

2.3.4.3 Операция ожидания потоков

Противоположной созданию потока операцией является *редукция* — операция, при которой поток-родитель дожидается завершения потоков-потомков (может быть один), после чего продолжает работу. Согласно устоявшейся терминологии, назовем такую операцию «join» - «соединение в точку». Графиче-

ской интерпретацией данной операции является обратный символ операции «fork» по созданию потоков: «» и, формально, операция определяется как

$$join: P \times (S \times T)^+ \rightarrow S \times T. \quad (2.29)$$

Находясь в некотором подмножестве T' множества потоков T , $T' \subseteq T$, так что $\forall t \in T': Parent(t') = T_{parent}$, каждый поток находится в соответствующем заданном состоянии своего подавтомата из множества состояний S , с некоторой вероятностью расширенный автомат может осуществить *редукцию*, после которой продолжает выполняться только один поток T_{parent} в заданном состоянии из S . Рисунок 2.16 иллюстрирует операцию создания двух потоков главным потоком и дальнейшую их редукцию, после которой выполняется один главный поток. Следует отметить, что редукция представляет собой модель примитива синхронизации «барьер».

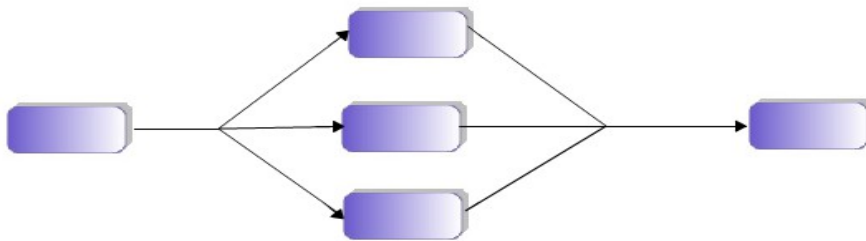


Рисунок 2.16 — Создание потоков и последующая редукция

Тело потока представляет собой подавтомат, и для правильного его описания нужно учесть правило, что другие порожденные им потоки должны быть завершены и редуцированы до перехода в конечное состояния подавтомата потока.

2.3.4.4 Обмен сообщениями

Наличие многопоточной составляющей в модели системы приводит к необходимости использования **средств взаимодействия**. Обычно клиент-серверные приложения взаимодействуют через отсылку и получение *сообщений*, при этом некоторые языки программирования построены полностью на обмене ими. Добавим сообщения в предлагаемую модель.

Определение 2.20. Множество сообщений Msg в модели состоит из

элементов, характеризующихся следующим кортежем из атрибутов:

1. Состояние, из которого сообщение отсылается ($s_{\text{from}} \in S$);
2. Состояние, в которое сообщение посылается ($s_{\text{to}} \in S$) или подавтомат;
3. Идентификатор сообщения ($msgid \in MsgID$).
4. Тип доставки сообщения, определяет, как сообщение будет доставляться в кратные состояния ($type = \{ ' \forall ', ' N '=n, ' \forall /me ' \}$):
 - \forall - сообщение «всем» («broadcasting») - отсылается всем экземплярам потока в кратном состоянии;
 - $N=n$ - сообщение конкретному экземпляру потока с его номером;
 - \forall /me - сообщение «всем, кроме меня» - отсылается всем экземплярам потока из потока этого же экземпляра, кроме самого себя.
5. Строка-текст сообщения, txt (строковый тип)

Таким образом,

$$Msg = \{ (s_{\text{from}}, s_{\text{to}}, msgid, type, txt) \}. \quad (2.30)$$



Следует заметить, что сообщения — это не обязательно клиент-серверный обмен с использованием, например, сетевых сокетов, но и оповещение одного объекта системы об изменении своего состояния для других её объектов (шаблон проектирования «Наблюдатель» - «observer»).

Для поддержки сообщений в определении расширенного автомата добавляется множество Msg :

$$A = (S, s_0, \delta, \gamma, S_{\text{fin}}, E, D, Msg). \quad (2.31)$$

Отсылка и получение сообщений тесно связаны с возникновением событий и исключений. Ведь получение сообщения инициирует событие, но сообщение может быть потеряно при пересылке и не дойти до адресата, и, чтобы получатель не ждал не дошедшего сообщения, генерируется исключение при отправке и/или исключение по тайм-ауту при получении. Во взаимодействующей системе сообщения являются основными «генераторами» событий и исключений.

Проиллюстрируем отправку и получение сообщений. Отправка — это

либо синхронная операция, либо асинхронная, однако всегда можно выделить состояние отсылки сообщения и состояние его приема). Получение сообщения осуществляется в состоянии, которое ожидает его получение; ожидание может быть прервано по тайм-ауту. Прием и передача сообщений показаны на рисунке 2.17. Здесь и далее сообщения будем обозначать «» или «».

В принципе, существует и другой подход, нежели ожидание и проверка доставки — вероятностное получение сообщений. Сообщение может быть отправлено и соответственно получено с некоторой вероятностью и обработано, как событие (рисунок 2.18,а) или как обычный вероятностный переход (рисунок 2.18,б)). В нашей работе мы будем рассматривать только первый способ (рисунок 2.17).

Заметим, что операции отправления и приема сообщения являются парными, т.е. для каждого отправленного сообщения должно быть место, в котором оно примется. Это можно выявить автоматически на этапе проектирования системы.

После добавления операции обмена сообщениями можно рассмотреть моделирование операции завершения потока родителем без ожидания корректного завершения работы потока (ранее мы рассмотрели операцию редукции, осуществляющую ожидание потоков, в данном же случае поток-родитель сам инициирует завершение потока).

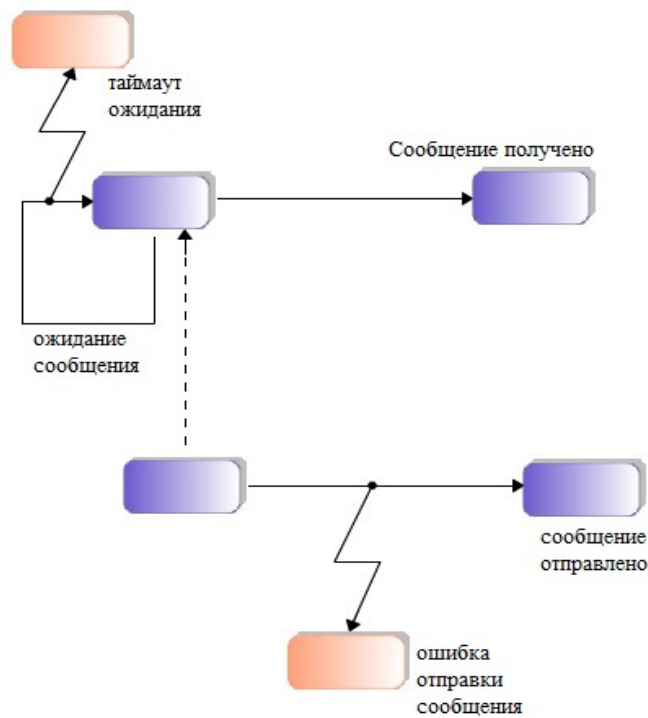


Рисунок 2.17 — Отправка и получение сообщения

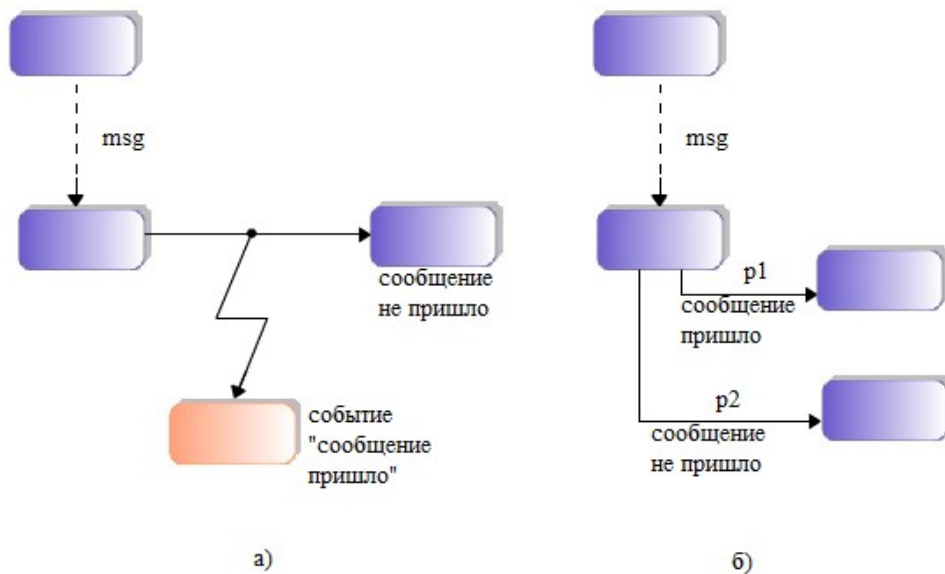


Рисунок 2.18 — Вероятностное получение сообщений

Операция принудительного завершения потока, например, в языке Java, является опасной и поэтому устаревшей (deprecated) и оставлена только для совместимости, поскольку может нарушить логику работы программы из-за возможных установок потоком блокировок без их снятия в результате

аварийного завершения. Завершая дочерний поток, родитель не знает, в каком состоянии тот находится (если только дочерний поток не посылает родителю сообщений о своём состоянии). Поэтому завершение дочернего процесса также можно реализовать при помощи сообщений, так впрочем и делается в UNIX системах при помощи сигналов.

В данном случае, однако, сообщение отсылается не состоянию, а всему подавтомату потока, что эквивалентно посылке сообщения любому из его состояний (разве что кроме заключительного, т.к. из него нет переходов, и, следовательно, реакция на сообщение не определена) и вероятностной обработки событий по получению сообщений. Определение посылки сообщения потоку:

$$\begin{aligned} & msg(s_{\text{from}}, T, msgid, type, txt) \Leftrightarrow \\ & \Leftrightarrow msg(s_{\text{from}}, s, msgid, type, txt) \forall s \in S', S' - \text{состояния } T. \end{aligned} \quad (2.32)$$

На рисунке 2.19 представлена ситуация отсылки сообщения о завершении потока родителем.

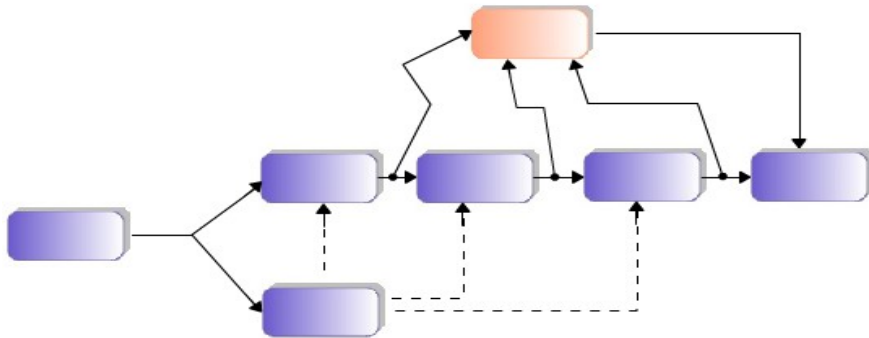


Рисунок 2.19 — Сообщение Kill

Здесь показана ситуация, когда поток отсылает одинаковое сообщение во все состояния потока (подавтомата), при этом возникают одинаковые события и осуществляется переход по этому событию в заключительное состояние.

Добавим в функцию переходов δ отсылку и прием сообщений:

$$\delta \supseteq P \times S \times Msg \rightarrow 2^{S \times P \times Msg}. \quad (2.33)$$

Это означает, что автомат, находясь в некотором состоянии, с некоторой вероятностью, после получения сообщения, осуществляет недетерминированный переход и с некоторой вероятностью отсылает сообщение.

2.3.4.5 Блокировка ресурсов

Перейдем к моделированию блокирующих ресурсов. Проблемы разделения доступа к ресурсам решаются с использованием примитивов синхронизации, такие как семафор или мьютекс, которые блокируются или разблокируются при помощи атомарных системных вызовов в ОС. Тестирующая среда является внешней по отношению к модели так же, как и операционная система является внешней по отношению к программе, поэтому внешняя тестирующая среда также может предоставлять модели общие блокируемые ресурсы и контролировать их использование.

Дополним описание расширенного автомата множеством общих блокируемых ресурсов Res :

$$A = (S, s_0, \delta, \gamma, S_{fin}, E, Msg, Res). \quad (2.34)$$

Рассмотрим операции $block(R), R \in Res$ и $unblock(R), R \in Res$, блокирующие и разблокирующие ресурсы из множества Res .

Операция $block(R)$ проверяет, кто на текущий момент является хозяином ресурса R (обозначим его $Host(R)$):

- Если $Host(R) = T_{current}$ ($T_{current}$ – текущий поток) - не делается ничего;
- Если $Host(R) = \emptyset$, то устанавливается $Host(R) = T_{current}$ и блокировка считается успешной, производится переход по успешной блокировке в соответствии с расширенной функции переходов;
- Если же $Host(R) \neq \emptyset, Host(R) = T, T \neq T_{current}$, то $T_{current}$ блокируется (т.е. осуществляется переход $s \rightarrow s', s \in S', T_{current} : S'$ и последующая проверка $Host(R)$). Блокировка может быть прервана по тайм-ауту, причем переход в состояния между блокировками запрещен.

Операция $unblock(R)$ делает следующее:

- Если $Host(R) = T_{current}$, то $Host(R) = \emptyset$ и блокировка снимается;
- Если $Host(R) \neq \emptyset$ и $Host(R) \neq T_{current}$, то ничего не делается.

Следует заметить, что после введения блокировок, динамическая кратность внутри потока в кратных состояниях между успешной операцией $block$

и *unblock* становится равной 1, поскольку в данный момент в состоянии может находиться только один поток. Поэтому соотношение 2.27 заменяется на соотношение

$$\forall s' \in S', \neg(s_b: block(R) \leq_{s'} s' \leq_{s'} s_u: unblock(R)) : N(s') \geq N(first(A')), \quad (2.35)$$

здесь $s_b: block(R)$ и $s_u: unblock(R)$ – соответственно состояния, где происходит успешное применение блокировки $R \in Res$ и его разблокировки. Рисунок 2.20 показывает пример автомата с блокировкой ресурсов.

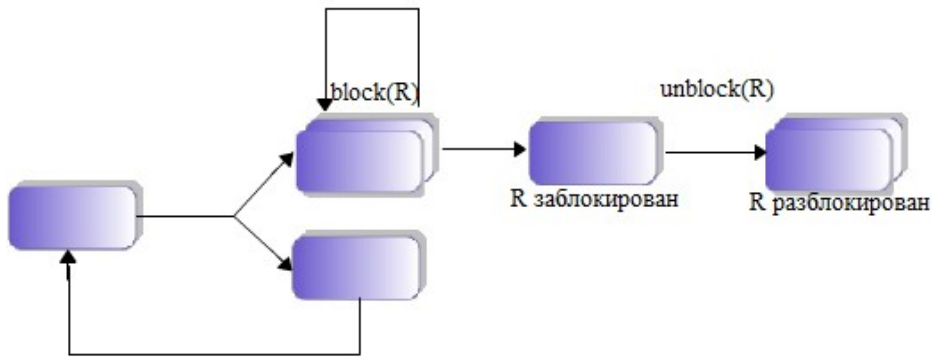


Рисунок 2.20 — Блокировка, разблокировка и кратность

Добавим в функцию переходов δ блокировку и разблокировку:

$$\delta \supseteq P \times S \times Res \rightarrow 2^{S \times P \times Res}. \quad (2.36)$$

Это означает, что расширенный автомат, находясь в некотором состоянии, после возможной успешной блокировки ресурса осуществляет недетерминированный переход в состояние, и после этого перехода возможно снятие блокировки с некоторого общего ресурса.

Добавив в расширенный конечный автомат как многопоточность, так и средства отсылки сообщений и блокировку общих ресурсов, заметим, что мы можем теперь считать наш автомат *агентом* [4], поскольку, в соответствии с агентно-ориентированным подходом, он обладает своим поведением (переходы между состояниями), взаимодействует с внешней средой (через блокировку ресурсов и отсылку сообщений) и может быть модифицирован другими агентами (он меняет свое состояние после получения сообщения).

2.3.5. Модель многокомпонентной системы высокого уровня

Расширенный вероятностный многопоточный автомат представляет собой модель одного компонента системы. Реальные распределенные системы состоят из взаимосвязанных компонентов, возможно, реализованных на различных языках программирования. Рассмотрим моделирование связей в многокомпонентной системе. Модель распределенной многокомпонентной системы можно представить как:

$$System = (Node, A^*, A_{structured}, BoundNodes, Msg, Res, CP(A^*)). \quad (2.37)$$

Здесь:

$Node$ – множество узлов, на которых работает моделируемое распределенное приложение. $Node = \{(NodeName, NodeAddress, NodeType)\}$ – кортеж, содержит следующую информацию об узле: $NodeName$ – имя узла, $NodeAddress$ – адрес протокола IP или DNS имя, по которому узел доступен в сети; $NodeType = \{WindowsPC, LinuxPC, Mobile, SmartPhone\}$ – множество различных типов платформ, на которых могут работать узлы (ПК под управлением Windows, Linux, мобильный телефон, смартфон – возможны и другие типы);

A^* – набор расширенных вероятностных многопоточных конечных автоматов, каждый из которых моделирует работу одного из компонентов системы;

$BoundNodes : Node \rightarrow AN, n \in N$, где $AN = \{(A, n)\}$ – отображение, определяющее связь узлов с компонентами, для узла $Node$ задает, какие компоненты, представленные в виде расширенного конечного автомата A ($A \in A^*$), выполняются на данном узле и с какой кратностью;

Msg – глобальное для системы множество посылаемых и получаемых сообщений. В данной работе предполагается, что компоненты системы как внутри, так и с друг с другом взаимодействуют посредством отправки-получения сообщений.

Res – глобальное множество разделяемых ресурсов, к которым имеют доступ все компоненты системы.

$CP(A^*)$ - Множество точек сопряжения компонентов системы (рисунок 2.22) показывает логические связи между состояниями в различных компонентах системы. Это множество определяет попарные связи между моделями компонентов:

$$CP(A^*) \ll \{CP(A_1, A_2), A_1 ! A_2, A_1 ! A^*, A_1 ! A^*\},$$

где $CP(A_1, A_2) \ll \{(s_1, s_2) \mid s_1 ! S_1, S_2 ! S_2, A_2 \ll A(S_1), A_2 \ll A(S_2), s_1 \text{handshake } s_2\}$.

Для того, чтобы представить точку сопряжения, можно привести пример рукопожатия двух людей (handshake). При этом два человека находятся каждый в своем состоянии, а также в совместном (рисунок 2.23).

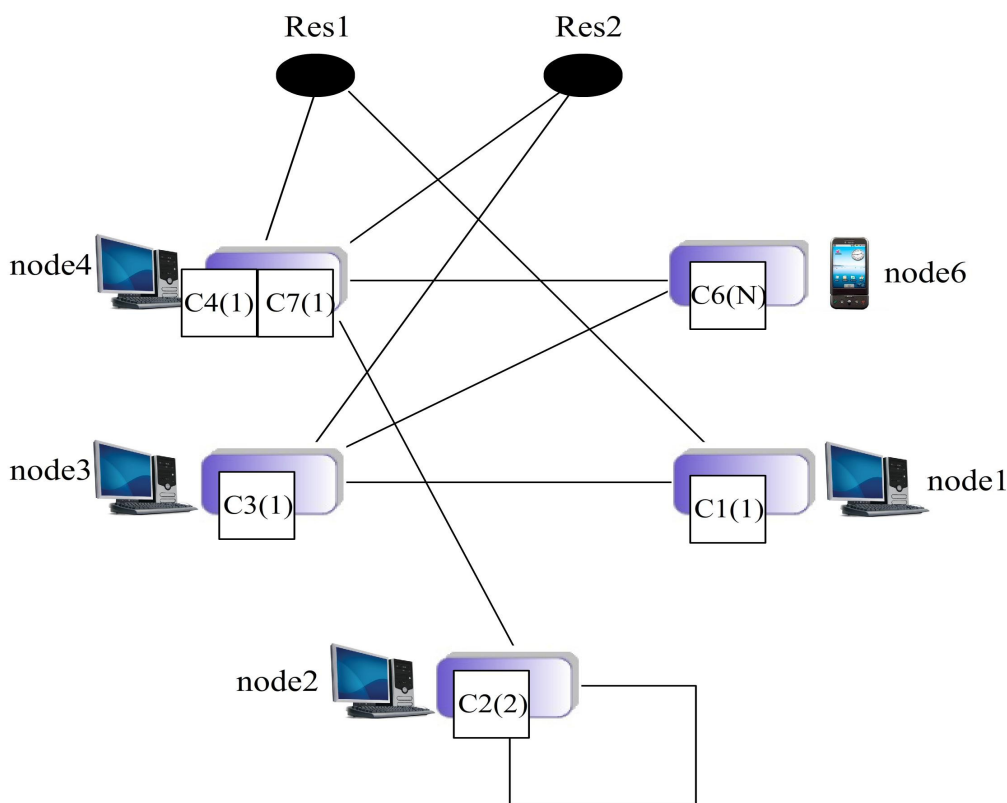


Рисунок 2.21 — Распределенная система

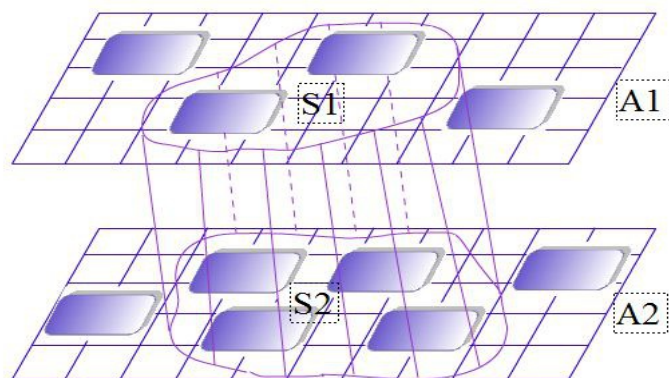


Рисунок 2.22 – Точки сопряжения

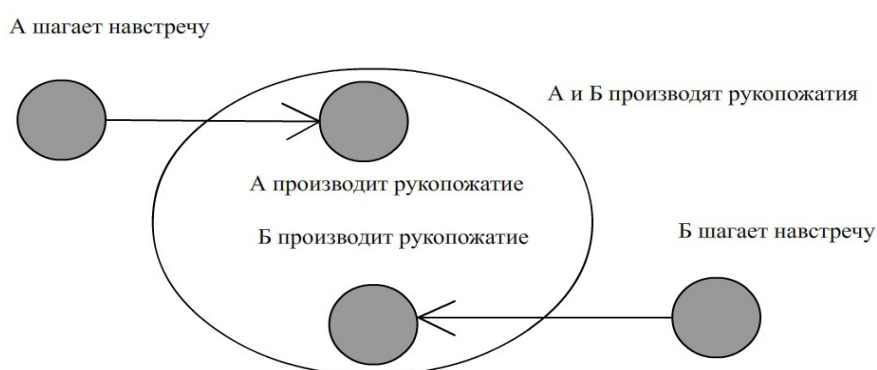


Рисунок 2.23 – Рукопожатие А и Б

Если одному состоянию при взаимодействии соответствует ровно одно состояние, то можно говорить о мощности связи 1:1, если несколько (подавтомат), то 1:N, а если подавтомату соответствует другой подавтомат — то M:N. Поскольку точки сопряжения задают соответствия подавтоматов, то соответствие можно задавать не только между различными компонентами, но и внутри отдельно взятого компонента между его подавтоматами потоков.

Точки сопряжения разделяются на неявные и явные:

- Неявные точки сопряжения возникают в результате использования примитивов синхронизации, не описываются явно и вычисляются автоматически:
 - При моделировании синхронного обмена сообщениями из *Msg* состояние отсылки сообщения соответствует состоянию получения со-

общения.

- При моделировании блокировки ресурса из Res для организации работы критической секции состояние блокировки соответствует состоянию разблокировки.
- Явные точки сопряжения описывают ожидаемые связи в системе. Таких связей нужно избегать, однако нужно моделировать и такие связи (возможно, взаимодействие корректно, но организовано очень нестандартно).

Точки сопряжения также могут быть использованы для определения многокомпонентной связанности при моделировании аспектно-ориентированных систем, что и рассматривается в 2.3.6.

Взаимодействие компонентов определяет структурный автомат $A_{structured}$, являющийся развитием структурной схемы, предложенной в [3].

Состояния структурного автомата — это компоненты системы с заданной кратностью, а переходы — взаимодействия между ними как результат отсылки и ожидания сообщения. Автомат взаимодействия определяется как четверка вида:

$$A_{structured} = (Cn, \delta_{structured}, InputPin, \Sigma). \quad (2.38)$$

Здесь:

Cn — множество состояний автомата $A_{structured}$, одно состояние — это взаимодействующий компонент системы с учетом кратности, т.е. $Cn = \{(A, n_+)\}$, где n_+ — суммарная по всем узлам сети кратность компонента системы, заданного расширенным автоматом A .

Σ — алфавит автомата, описывает действие, идентифицирующее переход, строковая надпись, описывающая отсылаемое сообщение.

$InputPin$ — отображение, определяет для каждого компонента один из входов, через которые принимаются сообщения, причем вход может принимать сообщения как от одного, так и от нескольких компонентов по типу «И»-ждать

сообщения от всех заданных компонентов (не важно, в каком порядке) и по типу «или» - ждать сообщение от одного из указанных компонентов:

$$InputPin : Cn \rightarrow Cn^* \times InType, InType = \{ \text{и, или} \} .$$

На рисунке 2.24 показаны примеры входов различных типов с точки зрения компонента С (слева рисунок содержит диаграмму состояний структурного автомата, справа — соответствующую диаграмму расширенного конечного автомата для компонента С) :

а) два простых входа принимают сообщения от различных компонентов А и В;

б) вход типа И принимает сообщения от компонентов А и В, при этом первое сообщение может быть послано компонентом А, тогда второе- компонентом В и, наоборот, первое-компонентом В, второе- компонентом А;

в) вход типа ИЛИ принимает сообщения либо от компонента А, либо от компонента В.

Автомат $A_{structured}$ действует согласно недетерминированной функции переходов $\delta_{structured} : Cn \times \Sigma \times InputPin \times SeqNum \rightarrow 2^{Cn}$, при этом $c_1 \rightarrow c_2$ означает, что c_1 инициирует отсылку сообщения, а c_2 - ждет сообщения (или сообщений по типу И либо ИЛИ). Компонент может посылать сообщения самому себе.

В определении функции переходов $\delta_{structured}$ множество $SeqNum$ - частично-упорядоченное множество номеров взаимодействия. Каждое взаимодействие (посылка сообщения) в системе должно иметь свой номер. Однако для недетерминированных и многопоточных систем определить глобальную нумерацию сложно. Мы используем подход, схожий с подходом при нумерации сообщений в диаграмме последовательностей языка UML.

2.3.6 Моделирование межкомпонентной связности

Технология аспектно-ориентированного программирования (АОП) – новая методология, которая нашла применение при разработке систем с нетри-

виальными связями и основана на определении функциональности, которая срабатывает до или после целевого действия. Технология АОП достаточно нова, в различных языках поддерживается по-разному, однако средств, позволяющих проводить моделирование и тем более проверку правильности программ, на сегодня не существует, тогда как даже создатели признают, что применение АОП может усложнять понимание программ, рассмотрим АОП в системе тестирования программ с созданием модели. При моделировании АОП будем предполагать, что задается аспект с некоторой функциональностью, которая может выполняться до заданного действия, либо после и вместо него. Рассмотрим моделирование аспектов на основе точек сопряжения.

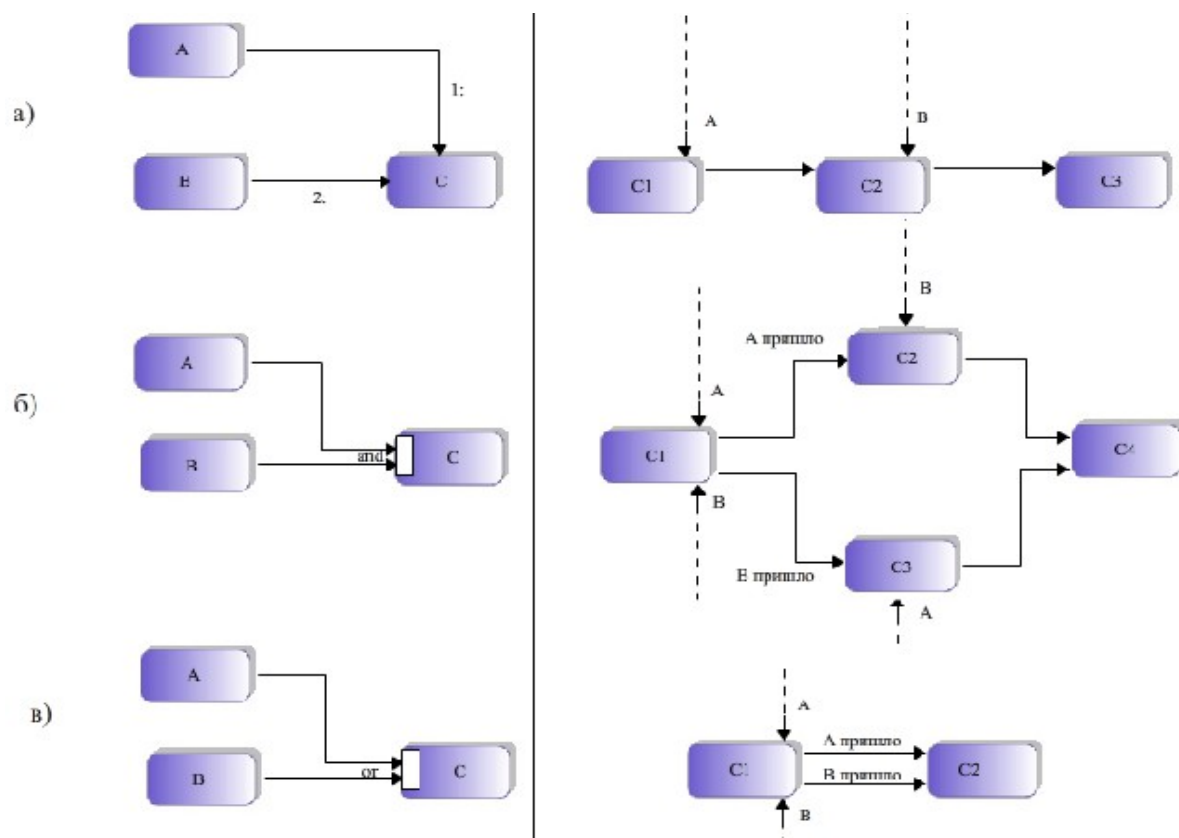


Рисунок 2.24 – Типы входов структурного автомата

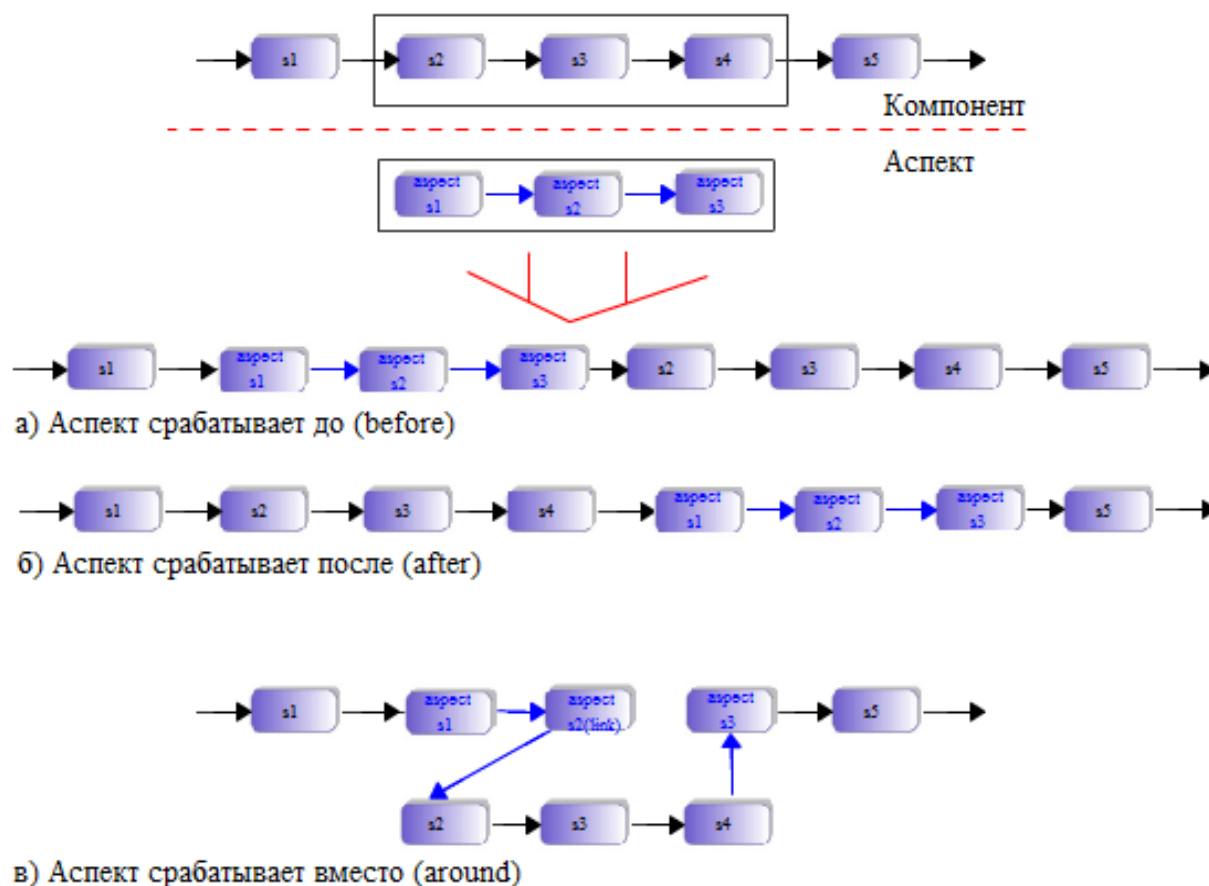


Рисунок 2.25 – Моделирование работы связанных компонентов с использованием технологии АОП

На рисунке 2.25 представлен компонент системы, содержащий пять состояний, три из них связаны через точки сопряжения с аспектом, имеющим три состояния, и показано, как изменяются автоматы связанного компонента системы в результате применения аспектов.

Для поддержки работы с аспектами следует ввести, помимо точек сопряжения, множество аспектов Asp , причем компонент, отвечающий за обработку аспектов, не предназначен для запуска, а содержит только описания аспектов. Каждая модель аспектного проекта состоит из нескольких моделей аспектов:

$$Project_{aspect} = \cup Asp. \quad (2.39)$$

При этом модель системы состоит как из проектов, которые моделируют работу системы, так и из описывающих аспекты:

$$System = (\cup Project) \cup (\cup Project_{aspect}). \quad (2.40)$$

Каждый аспект, в свою очередь, задается автоматом, реализующим его функциональность, типом срабатывания аспекта и точкой сопряжения:

$$Asp = (A, type, cp). \quad (2.41)$$

Тип срабатывания $type \in \{before, after, around\}$ - срабатывание аспекта соответственно до, после и вместо (см. рисунок 2.25). Точка сопряжения из глобального множества точек сопряжения $cp \in CP$ связывает аспект с другими компонентами системы, функциональность которых нужно изменить.

Теорема 2.2. Для любого автомата с аспектной группой существует эквивалентное преобразование в автомат без аспектной группы.

Доказательство. Обозначим $S_{old} \subseteq S$ – подавтомат, реализующий «прежнюю» функциональность (до применения аспектов). S_{new} - состояния, описывающие аспект, $x \rightarrow y$ переход из состояния x в y за один шаг (дуга из x в y)

В соответствии с предложенной терминологией АОП, обозначим за $weave(A): A \times Asp \times CP \rightarrow A'$ операцию применения аспектов к компонентам системы. Теорема утверждает, что $weave(A): A \rightarrow A'$ всегда осуществимо.

Докажем теорему методом конструирования этого преобразования. В соответствии с типом аспекта $type$ возможно три варианта изменения функциональности компонента:

- а) *Срабатывание до.* В данном случае все переходы, ведущие в состояние, являющееся начальным состоянием группы связанных состояний компонента, заменяются на переходы в начальное состояние аспектной группы, а из всех заключительных состояний аспектной группы осуществляются в начальное состояние группы с такой-же вероятностью, которая была при заходе в начальное состояние группы связанных состояний компонента:

$$\begin{aligned} \forall x, x \rightarrow first(S_{old}) &\Rightarrow x \rightarrow first(S_{asp}); \\ \forall z \in last(S_{asp}) &\Rightarrow z \rightarrow first(S_{old}) \\ P(z \rightarrow first(S_{old})) &= P(x \rightarrow first(S_{old})). \end{aligned} \quad (2.42)$$

- б) *Срабатывание после.* В данном случае все переходы, ведущие из заклю-

чительных состояний группы связанных состояний компонента, заменяются на переходы в начальное состояние аспектной группы, и из заключительных состояний аспектной группы осуществляется переход в состояния, в которые вели переходы из заключительных состояний группы связанных состояний с той же вероятностью:

$$\begin{aligned} \forall x \in last(S_{old}) \forall y: x \rightarrow y \Rightarrow x \rightarrow first(S_{aspect}); \\ \forall z \in last(S_{aspect}) \Rightarrow z \rightarrow y \\ P(z \rightarrow y) = P(x \rightarrow y). \end{aligned} \quad (2.43)$$

- в) *Срабатывание вместо*. В этом случае все переходы, ведущие в первое состояние группы связанных состояний, осуществляются в первое состояние аспектной группы, а из последнего состояния аспектной группы - в состояния, в которые осуществлялся переход из заключительного состояния группы связанных состояний с той-же вероятностью. При этом возможен вызов «прежней» функциональности из состояния, принадлежащего аспектной группе и специально помеченного как состояние связи (link). Из такого состояния перед переходом в его следующее состояние осуществляется переход в первое состояние группы связанных состояний, а из последнего состояния группы — в состояния, в которые осуществляется переход из состояния связи

$$\begin{aligned} \forall x, x \rightarrow first(S_{old}) \Rightarrow x \rightarrow first(S_{aspect}); \\ \forall y \in last(S_{aspect}) \forall z: last(S_{old}) \rightarrow z \Rightarrow \\ \Rightarrow y \rightarrow z, \forall s \in S_{link} \subseteq S_{aspect} \Rightarrow s \rightarrow first(S_{old}), \\ \forall w \in Next^1(s) \forall v \in Last(S_{old}) \Rightarrow v \rightarrow w; \\ P(v \rightarrow w) = P(s \rightarrow first(S_{old})) = 1, P(y \rightarrow z) = P(x \rightarrow first(S_{old})). \end{aligned} \quad (2.44)$$

Таким образом, мы получаем на выходе измененные расширенные конечные автоматы в зависимости от определенных аспектов по правилам 2.42, 2.43, 2.44 и точек их сопряжения с компонентами системы. ■

2.3.7 Итоговая модель системы в виде взаимодействующих компонентов

В предлагаемой математической модели программной системы можно вы-

делить следующие уровни абстракции:

- на уровне системы — структурный автомат ($A_{structured}$);
- на уровне компонентов - комбинация расширенных вероятностных многопоточных конечных автоматов с обработкой событий и исключений, с возможностью моделирования межкомпонентных и внутрикомпонентных связей при помощи посылки сообщений, блокировки ресурсов (автоматы A^*) и изменения функциональности посредством применения аспектов (множество Asp);
- на нижнем уровне - состояния, привязанные к реальному коду программного компонента (см. формулу 2.4), состояния могут группироваться в подавтоматы (например, логика работы потока задается подавтоматом) и рассматриваться как сверхсостояния.

При проектировании модель системы может быть задана двумя видами диаграмм — диаграммой, описывающей модель верхнего уровня (структурный автомат) и связанными с ним диаграммами для каждой из моделей компонентов в виде расширенного конечного автомата, а также дополнительной диаграммой, показывающей для распределенной системы расположение компонентов по узлам сети (рисунок 2.21).

Итоговая модель современной распределенной недетерминированной тестируемой системы имеет вид:

$$System = (Node, A^*, A_{structured}, BoundNodes, Msg, Res, CP(A^*), Asp). \quad (2.45)$$

Здесь:

$Node$ — множество узлов, на которых выполняются компоненты распределенной системы (см. формулу 2.37);

A^* — множество расширенных конечных автоматов, каждый из которых моделирует один компонент системы, для которой строится модель;

$A_{structured}$ — структурный автомат, модель высокого уровня для взаимодействующей системы (описан в 2.3.5);

Asp – множество определенных аспектов, реализующих сквозную функциональность (см. 2.3.6);

$CP(A^*)$ – множество точек сопряжения между моделями компонентов и для связи с аспектами (см. 2.3.5 и 2.3.6);

Res – множество глобальных общих блокируемых ресурсов;

Msg – множество глобальных посылаемых сообщений.

Заметим, что в формуле 2.30 мы рассматривали отсылку сообщений из множества Msg в рамках текущего автомата. На самом деле, для моделирования особой разницы нет — моделировать сообщения в рамках одного автомата или нескольких, главное — знать состояние, в котором данное сообщение будет принято.

Структурный автомат задается как четверка вида:

$$A_{structured} = (Cn, \delta_{structured}, InputPin, \Sigma),$$

где

Cn – состояния структурного автомата — компоненты моделируемой системы с учетом кратностей. В структурном автомате переход между состояниями-компонентами системы означает отсылку сообщения между ними.

$\delta_{structured}$ – функция переходов структурного автомата, определяет переход-отсылку сообщения в зависимости от компонента, номера сообщения согласно определенной системы нумерации (множество $SeqNum$), входа из множества $InputPin$, куда будет послано сообщение и идентификатора сообщения из Σ .

Каждый элемент множества A^* представляет собой расширенный конечный автомат, построенный инкрементно на протяжении данной главы, вида:

$$A = (S, s_0, \delta, \gamma, S_{fin}, E, Msg' \subseteq Msg, Res' \subseteq Res), \quad (2.46)$$

где Msg' и Res' – соответственно локальные для компонента распределённой системы части глобальных множеств Msg и Res . Определяя множества Res и Msg в M как глобальные, мы тем самым допускаем, что компоненты системы могут посылать сообщения как внутрикомпонентно, так и межкомпо-

нентно, а также блокировать общие ресурсы.

Остальные множества подробно были описаны на протяжении предыдущих параграфов при инкрементном построении автомата.

Напомним, что логика работы автомата A в общем случае определяется двумя функциями переходов:

- Недетерминированной функцией переходов δ , она задает отображение:

$$\begin{aligned} O: S! D! P! N! T! Msg'! Res'! !! \parallel \\ 2^{\wedge}(((T! S)^*! S)! Msg'^*! Res') \end{aligned} \quad (2.47)$$

Находясь в состоянии из S кратности N по действию из D с вероятностью из P в потоке из T и по полученному сообщению из Msg' , после установки блокировки ресурса на Res' модель системы недетерминировано переходит или в несколько состояний, создав несколько новых потоков из T , или просто в следующее состояние в текущем потоке; при этом возможны отсылка сообщения из Msg' и разблокировка ресурса из Res' .

- Функцией переходов «по ребрам» γ , определяющей возможность возникновения некоторого числа событий или исключительных ситуаций при переходе из состояния в состояние $\gamma: S \times S \rightarrow E^*$.
- Операцией редукции, осуществляющей сворачивание потоков в один:

$$join: (S \times T)^+ \rightarrow S \times T.$$

Компоненты системы, каждый из которых задан расширенным автоматом A , могут взаимодействовать друг с другом через:

- посылку и получение сообщений из множества Msg ;
- блокировку, ожидание разблокировки и разблокировку общих ресурсов из Res ;
- точки сопряжения (взаимодействие типа «рукопожатие») из множества CP , задающие ожидаемую синхронизацию компонентов системы.

Для моделирования межкомпонентной связности, в модели присутствует также аспекты из множества Asp (см. 2.3.6), каждый из которых задается

своим автоматом, типом работы и точкой сопряжения ($Asp = (A, type, cp)$), аспекты могут изменить функциональность связанных через точки сопряжения с ними компонентов системы — добавить функциональность, которая срабатывает до, после или вместо связанной функциональности. Операция $weave: A^* \times Asp \times CP \rightarrow A^{*'}$ изменяет расширенные автоматы системы путем применения сквозной функциональности (формулы 2.42, 2.43, 2.44), моделируемой аспектами, как правило, до начала моделирования (работы) системы.

Таким образом, завершено описание модели взаимодействующей распределенной системы, далее по рассмотренному теоретико-множественному представлению модели будет построена объектная мета-модель и описана в виде формального языка, созданы средства для разработки и тестирования на основе построенной модели.

2.4 Предварительная оценка модели

Предлагаемая модель позволяет описывать системы в зависимости от потребностей тестирования. В таблице 2.1 приведены типы систем и предложения по описанию модели.

Таблица 2.1 - Виды тестируемых систем и способы описания модели

Вид тестируемой системы	Способ описания модели	Соответствующий пункт
1. Недетерминированные, вероятностные системы	Описание вероятностных переходов в модели	2.3.1 2.3.2
2. Многопоточные системы	Моделирование операций создания потоков, редукции, синхронизации при помощи сообщений, блокировка общих ресурсов	2.3.4
3. Событийно-	Моделирование событий и исключений,	2.3.3,

ориентированные системы	вероятностные переходы	2.3.2
4. Распределённые системы	Моделирования обмена сообщениями между компонентами системы, общие блокировки, синхронизация через точки сопряжения	2.3.5 2.3.7
5. Аспектно-ориентированные системы	Моделирование аспектов как специального вида компонентов, связанных через точки сопряжения	2.3.6
6. Гибридные системы	Описание модели с использованием 1-5, применяя нужный вид системы там, где это необходимо.	2.3.1 - 2.3.7

Таким образом, рассмотренная модель является моделью, предназначенной собственно для моделирования взаимодействующих систем в целях их тестирования, привязана к исходному коду компонентов системы и этим и отличается от других проанализированных моделей. Адекватность модели будет доказана в разделе 3.3.

Задание к разделу 2

1. Проанализировать предложенную в данной главе модель
2. Обсудить ее достоинства и недостатки
3. Сравнить данную модель с моделью, предложенную вами в задании к главе 2
4. Обсудить возможности расширения как представленной, так и своей модели

3 Практическая реализация процессов разработки и тестирования на основе моделей

3.1 Способ представления автомата в виде состояний, переходов и операций

Для проведения дальнейшего объектно-ориентированного анализа, а также поскольку описывать поведение сложными функциями перехода затруднительно, можно предложить альтернативный способ задания расширенного автомата как тройки из набора состояний, переходов и операций, которые могут произойти в данном состоянии (возможно, с генерацией событий):

$$A = (S, Trans, Op), \quad (3.1)$$

где $Trans = S \rightarrow S \times E^*$ – отображение, определяет переход с возможной генерацией событий, $Op = \{fork, join, send, receive, block, unblock\} \times E^*$ – множество рассматриваемых ниже операций (сложных переходов) в расширенном автомате также с возможными связанными событиями, подмножество функций переходов $\delta \cup \gamma$.

Операции (подробно описаны в 2.3.4):

- Создание потока $fork: P \times S \rightarrow (S \times T)^+$. Находясь в некотором состоянии из S , с некоторой вероятностью P , компонент переходит в несколько состояний (хотя бы в одно), создав несколько потоков (элементы множества T , хотя бы один поток, при этом текущий поток является родителем для вновь созданных потоков и продолжает выполняться вместе с ними).
- Ожидание завершения потоков

$$join: P \times (S \times T_{parent}) \times (S_{fin} \times T_{slave})^+ \rightarrow S \times T_{parent}.$$

Здесь $T_{parent} \in T$ – родительский поток, который осуществляет ожидание подчиненных потоков $T_{slave} \subseteq T$. Более кратко это можно записать как $join: P \times (S \times T)^+ \rightarrow S \times T$. Находясь в некотором состоянии, в некотором потоке, с вероятностью P компонент начинает ожидать завершения некоторого количества потоков из множества T (перехода их подавтоматов в заключительное состояние), и после этого переходит в

новое состояние из S в своем потоке.

- Посылка сообщения $send \subseteq (\delta \vee \gamma): S \times T \times P \rightarrow (S \times Msg) \vee E$. Компонент, находясь в состоянии из S в потоке из множества T с некоторой вероятностью P , инициирует отсылку сообщения из множества Msg , при этом переходит в новое состояние из S или, при неудачной попытке отсылки, инициирует событие из множества E .
- Получение сообщения $receive \subseteq (\delta \vee \gamma): S \times T \times P \times Msg \rightarrow S \vee E$. Компонент, находясь в состоянии из S в потоке из множества T с некоторой вероятностью P , начинает ожидать сообщение Msg и, получив его, переходит в новое состояние из S . При ошибке получения сообщения (например, по таймауту), генерируется событие из множества E .
- Блокировка разделяемого ресурса

$$block \subseteq (\delta \vee \gamma): S \times T \times N \times P \times Res \rightarrow (S \times 1 \times Res) \vee E.$$

Находясь в состоянии из S в потоке из T с некоторой кратностью N с вероятностью P , поток текущего компонента начинает блокировку ресурса из Res и при удачной блокировке переходит в новое состояние с кратностью 1. При неудачной блокировке поток зависает и ждет освобождения ресурса, при неуспешном ожидании может быть сгенерировано событие из E .

- Разблокировка разделяемого ресурса

$$unblock \subseteq (\delta \vee \gamma): S \times 1 \times Res \times P \rightarrow (S \times N \times Res) \vee E.$$

Находясь в состоянии из S , владея заблокированным ресурсом Res и имея кратность состояния 1 (никакой другой поток не находится в данном состоянии), компонент с некоторой вероятностью P осуществляет попытку разблокировки ресурса Res , при успешной разблокировке переходит в новое состояние с любой допустимой кратностью, при неудачной – может генерировать событие из E .

3.2 Способы описания модели и ее соотношение с исходным кодом системы.

Пусть имеется распределенная многокомпонентная система, которую необходимо протестировать на основе модели. Существует два варианта: программный код системы уже имеется, и модель нужно построить по существующему коду; или код отсутствует, и система существует в форме спецификаций и диаграмм, описывающих ее поведение. Модель должна строиться и в том, и в другом случае.

Нами предполагается строить соотношение кода системы и модели по предлагаемому автором принципу - «код и модель — одно целое», как при отсутствии, так и при наличии кода системы к моменту описания модели.

3.2.1 Пример описания модели при отсутствии кода

Для программных систем, которые находятся в стадии проектирования, предлагается использовать разработанную объектно-ориентированную модель в виде системы классов, реализующих предложенную модель в качестве взаимодействия распределенных компонентов, каждый из которых задается набором состояний, переходов и операций (как было рассмотрено в 3.1). В соответствии с техникой проектирования на основе моделей, по математической модели была создана мета-модель, которая устанавливает, какие элементы и связи между ними могут быть использованы. Мета-модель современной программной системы разработана с использованием EMF (Eclipse Modeling Framework) и представляет собой систему классов с атрибутами, построенных по модели, рассмотренной в 3.1. По мета-модели были разработаны классы и написаны методы, реализующие на языке Java описанные в 2.3 и 3.1 принципы работы математической модели (пример использования таких классов приведен в приложении В).

Проектирование системы в этом случае строится путем манипулирования разработанными классами (инстансирования, задания их атрибутов, наследования, установки пользовательского кода, который выполняется в состояниях). При этом первоначально модель проектируется в виде диаграмм — структурного автомата, диаграммы распределения компонентов по узлам сети (п. 2.3.5), и расширенных многопоточных конечный автоматов для каждого из компонентов системы, далее при помощи программных редакторов модели (рисунок 3.1) автоматически генерируются конструкторы объектов-экземпляров классов мета-модели с параметрами, которые и определяют заданное пользователем поведение модели. После генерации пользователь может написать свой программный код (см. пример в приложении В), который определит поведение системы по выполнению нужных пользователю специфических действий, все взаимодействие же уже реализовано в предлагаемых классах. Процесс разработки при этом полностью становится управляемым моделями(MDD).

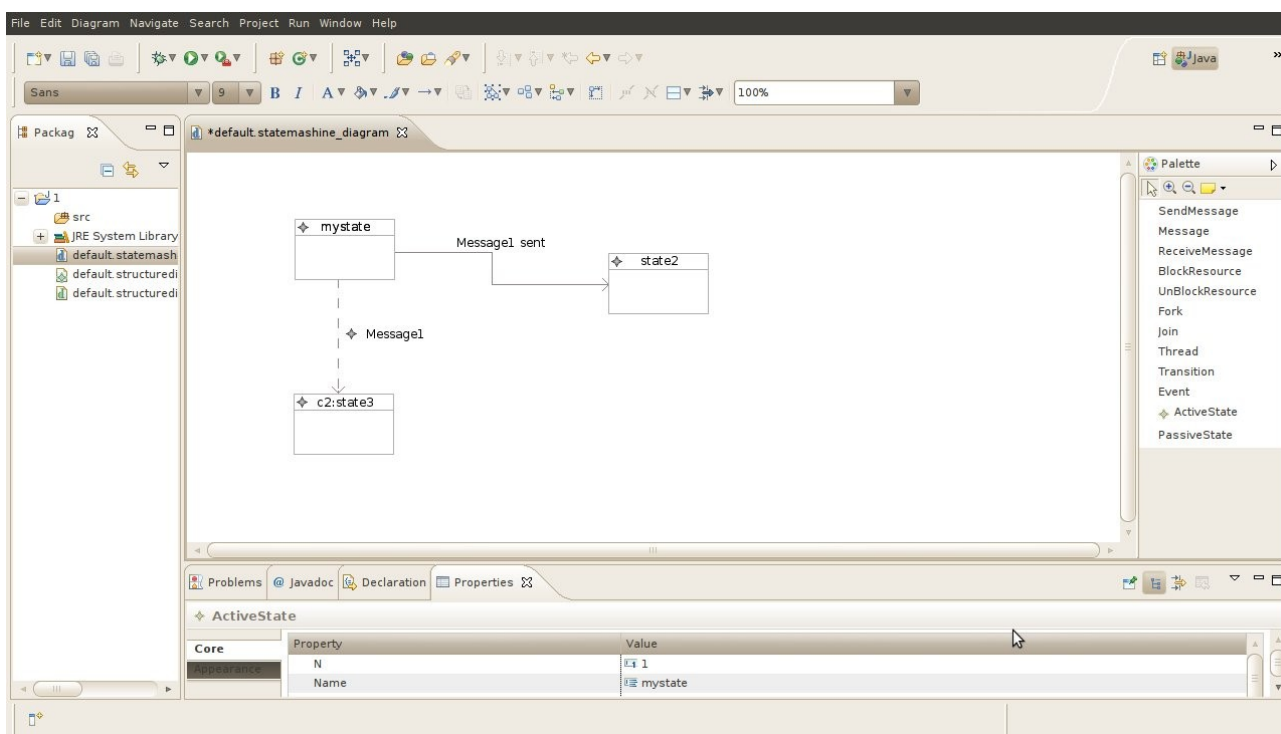


Рисунок 3.1 – Прототип для проектирования расширенного вероятностного конечного автомата, созданного при помощи Eclipse GMF

Тестировать такую систему проще, так как мы имеем разработанную мо-

дель плюс систему классов, которые помимо основной работы по организации логики переходов прозрачно для пользователя взаимодействуют с тестовым окружением.

Следует учесть, что, используя нашу мета-модель в качестве каркаса будущей системы, приходится разрабатывать программу в терминах состояний и систем переходов, т.е. по сути методом «State driven agent design» («проектирование агентов, управляемое состоянием»), когда приходится явно выделять состояния. Сегодня в программной инженерии существует так называемый «автоматный подход» к программированию и проектированию, когда система изначально реализуется как конечный автомат с использованием либо простого оператора switch, либо с применением объектно-ориентированных надстроек (класс «автомат», «состояние» и прочие), в MS Visual Studio официально входит Microsoft Workflow Foundation, позволяющая создавать системы в виде автоматных диаграмм с генерацией классов для этого. Построение логики программы в терминах состояний и переходов становится все более актуальным, особенно во встраиваемых и мобильных системах.

3.2.2 Пример описание модели при наличии кода

Для программных систем, код которых уже написан, необходимо встроить модель в исходный код. Парадигма «код и модель-одно целое» предполагает описание модели на специальном языке описания совместно с исходным кодом. Переходы и состояния привязаны к файлам и строкам исходного кода, поэтому модель описывается на месте, где определяются состояния, переходы, сообщения, потоки и т. д.

Описание модели в коде не вызывает ошибок компиляции, если модель внедряется в псевдокомментариях к исходному коду, которые будут пропускаться при компиляции кода приложения, но будут обрабатываться анализирующей и тестирующей системой. Таким образом, код модели является *метаданными* к исходному коду.

Можно было, конечно, поступить по-другому, используя стандартные

средства для описания метаданных — аннотации в Java и атрибуты в языках .NET. Однако аннотации и атрибуты можно писать только к классу, к интерфейсу и к методу, в то время как состояние может описывать любую часть метода. Требовать от разработчиков, чтобы они переписали код методов (функций), точно определяющих состояния системы, означает поменять полностью культуру разработки, и мы не будем этого делать.

Еще одна возможность — использовать технологию Linq, которая позволяет использовать лямбда-выражения внутри операторов (например, код состояния можно было бы описывать внутри объявления состояния, например так: `state(<параметры состояния>, body=> {<тело состояния>})`). Однако ориентированность только под язык .NET снижает область применения данного подхода.

Нашей целью было создать независимый от кода системы язык описания предлагаемой модели системы, конструкции которого можно помещать в теле программы в виде комментариев специального вида, которые обрабатываются предлагаемым далее инструментарием для тестирования и не влияют на компиляцию исходного кода компиляторами языка, на котором этот код написан.

Для синтаксической структуры языка авторами была выбрана XML структура, которая хорошо описывает вложенные конструкции, применяется повсеместно, позволяет использовать существующий синтаксический анализатор, и, что самое важное, рассмотренная выше мета-модель напрямую отображается в XML сущности с атрибутами. Грамматика языка приведена в приложении Г. Для описания модели при помощи диалоговых средств (рисунок 3.2) с генерацией представления на XML языке был разработан прототип, который позволяет в среде разработки выделять код, определять состояния, переходы и операции и создавать описание модели в псевдокомментариях.

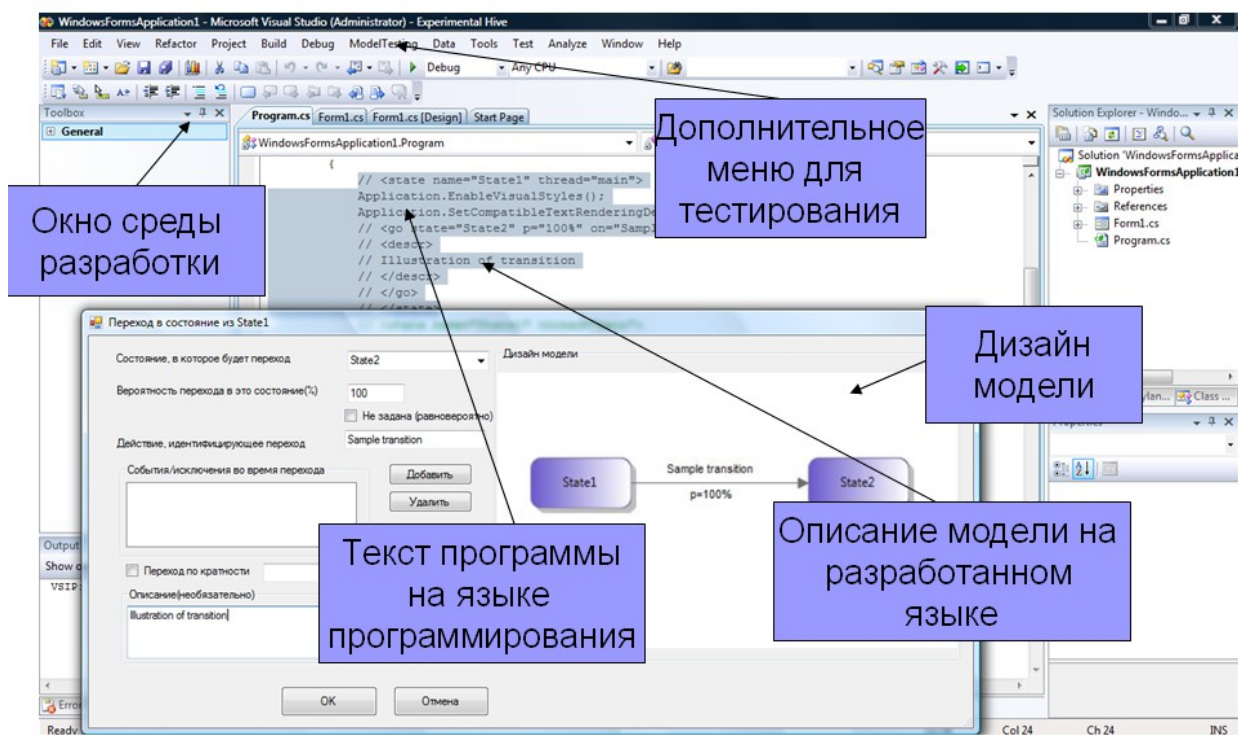


Рисунок 3.2 – Описание модели с генерацией кода на XML языке

Конечно, авторы отдают себе отчет в том, что описание модели как комментария к исходному коду не очень хорошо с точки зрения структуры кода, поскольку мы используем комментарии-средство языка, не предназначенное для этого. Описание модели в комментариях имеет также обратную сторону, связанную с сопровождением кода в процессе цикла жизни программного проекта. Код может быть изменен в процессе рефакторинга или оптимизации, и мета-комментарии с описанием модели уже далее могут не соответствовать написанному коду. Однако, отслеживая код в интегрированной среде разработки при его изменении только из среды, можно пресечь возможные ошибки в изменении местоположения элементов модели и кода, а также попытки вставки описания модели в логически неверные места кода (например, в обычные комментарии) и допустить вставку элементов модели только между операторами. Идея добавления информации в «псевдокомментарии» не нова, так поступают, например, среда разработки MS Visual Studio, генерируя в комментариях специального вида теги для описания документации системы, среда Netbeans, сохраняя в спе-

циального вида комментариев информацию о расположении элементов графического интерфейса, при этом, не показывая эту информацию пользователям.

3.3 О практической значимости и адекватности модели

В данном параграфе сформулируем и докажем несколько утверждений, важных с точки зрения адекватности модели. Предварительная оценка, сделанная в 2.4, не дает ответа на вопрос, практически реализуемы ли принципы создания модели для конкретной системы.

В результате проведенного исследования можно утверждать, что модель, рассмотренная в 2.3, **практически реализуема**. Автором сначала по математической модели построена объектная мета-модель (приложение Б), далее к разработанным классам добавлены методы, реализующие многокомпонентную распределенную систему на основе состояний, переходов и операций на языке Java (см. приложение В). Перед использованием различных вариантов модели в процессе разработки необходимо показать их эквивалентность.

Утверждение 3.1. Модель системы в виде классов, модель системы, описанная на XML языке, и математическая модель в виде множеств эквивалентны с точностью до описания взаимодействия.

Доказательство. Рассмотрим, как получить модель в виде множеств из XML модели. Будем действовать следующим образом: пройдемся по всем (в том числе, вложенным) тегам и сформируем множества, каждое из которых задается отдельным тегом. Те множества, которые представляют собой кортеж из нескольких множеств, собираются из атрибутов, заданных у XML тегов. В итоге, созданные множества будут описывать многокомпонентную систему, каждый компонент которой будет иметь модель в виде состояний, переходов и операций (параграф 3.1). Она отличается от искомой тем, что отображение δ разобрано по операциям и переходам, т. е. для получения отображения δ нужно объединить все операции и переходы без учета множества исключений E : $\delta = \cup (Op \cup Trans) \setminus E^*$, или сконструировать его из множеств по определению

(формула 2.47).

Обратно, чтобы получить из модели в виде множеств XML модель, нужно для всех множеств в модели создать соответствующие XML теги, возможно, с атрибутами, если множество представляет собой кортеж.

Поскольку XML модель привязана к исходному коду (согласно принципу «код и модель- одно целое», она объявляется внутри исходного кода в комментариях), то модель в виде множеств может не соответствовать реальным состояниям в файлах с исходным кодом (т.е. она не привязана к коду), однако, мы говорим о соответствии моделей с точки зрения моделирования взаимодействия.

Для получения XML модели по модели из системы классов нужно проанализировать синтаксическим анализатором файлы с главными классами компонентов, где происходит создание всех экземпляров классов модели (см. листинг в приложении В) и узнать, какие объекты соответствующих классов создаются и с какими атрибутами, и для каждого экземпляра класса сформировать свое XML описание (также возможно для каждого компонента после его запуска использовать точку останова и с использованием Java Debugging API [1] динамически получить данные о всех созданных экземплярах классов модели с их атрибутами). Поскольку модель в виде классов разрабатывалась не только для описания взаимодействия, но и для возможности реализации реальных систем, исходя из состояний, переходов и операций, то в ней имеются возможности, которые XML модель не включает (например, пользовательский код в состояниях, общие переменные), однако мы можем говорить об эквивалентности с точки зрения взаимодействия.

Обратно, для получения модели в виде классов, нужно просканировать XML модель и создать соответствующие классы с атрибутами в виде XML атрибутов. Если атрибут в XML указывает строковое имя, например, следующее состояние для перехода, то сделать ссылку на соответствующий экземпляр класса, имеющий данное строковое имя как атрибут.

Эквивалентность модели в виде множеств и модели в виде классов доказывается через эквивалентность этих моделей XML модели с точностью до описания взаимодействия. ■

Утверждение 3.2. С помощью предложенной модели в виде классов можно реализовать любой параллельный алгоритм без использования внешних примитивов синхронизации и без внешнего управления состоянием потока программы, с использованием только описанных переходов и операций.

Доказательство. Согласно теории машин абстрактных состояний (Abstract state machines, ASM), было доказано, что для любого параллельного алгоритма существует ASM машина, которая может по шагам симулировать данный алгоритм. Такая машина может быть описана при помощи формального языка. Если мы сможем практически реализовать интерпретатор формального языка для описания машины абстрактных состояний с использованием практической реализации нашей модели на основе состояний, переходов и операций, то данным интерпретатором мы сможем проинтерпретировать формальное описание любого параллельного алгоритма, следовательно, при помощи нашей реализации модели мы сможем реализовать любой параллельный алгоритм. Понятно, что, поскольку наша реализация модели написана на Java, с помощью средств данного языка можно реализовать любой интерпретатор (можно задать одно состояние и написать в нем код, который действует не исходя из состояний, а используя обычную ООП модель исполнения с использованием средств создания и синхронизации потоков в JavaVM). Задав в формулировке требование использовать модель переходов и операций, предполагается, что мы полагаемся на адекватность созданной модели.

Согласно [5] ASM машина задается алфавитом Σ (содержит имена функций), начальным состоянием (начальным значением функций), переходными правилами и начальным правилом.

Переходные правила представляют собой :

1. $f(s_1 \dots s_n) := t$ - в следующем состоянии значение функции f на наборе аргументов $s_1 \dots s_n$ устанавливается равным t .
2. $P \text{ seq } Q$ - правило P выполняется после правила Q .
3. $P \text{ par } Q$ - правила P и Q выполняются параллельно.
4. $\text{if } \phi \text{ then } P \text{ else } Q$ - если $\phi = \text{true}$, выполнить P , иначе выполнить Q .
5. $\text{let } x = t \text{ in } P$ - присвоить значение t локальной переменной x и выполнить P (область действия x после выполнения P заканчивается).
6. $\text{forall } x \text{ with } \phi \text{ do } P$ - выполнить параллельно P для каждого x , удовлетворяющего ϕ (область действия x после выполнения P заканчивается).
7. $\text{choose } x \text{ with } \phi \text{ do } P$ - недетерминированно выбрать x , удовлетворяющий ϕ и выполнить P .
8. skip - пустое правило.
9. $r(t_1 \dots t_n)$ - вызов правила r с параметрами $t_1 \dots t_n$.

Асинхронная ASM задается, кроме того, множеством агентов *Agents*, каждый из которых может выполнять параллельно другим агентам заданное начальное правило. Внутри правил текущий агент идентифицируется как *self*. Следует учесть, что синхронизация в ASM неявная, т.е. зависит от реализации. Считается, что обновление значения общих переменных атомарно. Кроме того, обычно считается, что при не заданной ветви *else* в условии $\text{if } \phi \text{ then } P$ и при ложном условии $\phi = \text{false}$ происходит блокировка агента до тех пор, пока условие не станет истинным.

Покажем, как можно реализовать интерпретатор для машины ASM.

Указанная машина в самой простой интерпретации (без типов данных) может быть описана такой КС-грамматикой:

```
asml : rulesDefinitions initialDefinitions agentDefinitions mainRule ;
```



```

rulesDefinitions: ('rule' name '(' args ')' '{' ruleBody '}' ';' )* ;
initialDefinitions: 'initial: ' ( name '(' args ')' '=' value ';' )*;
agentDefinitions: ('agent' name '=' name ';' )*;
mainRule: 'rule main()' '{' ruleBody '}' ;
args: name (',' name)* ;
values: value (',' value)*;
ruleBody:
name '(' args ')' ':=' value ';' |
'seq' '{' (ruleBody*) '}' |
'par' '{' (ruleBody*) '}' |
'if' value 'then' ruleBody ('else' ruleBody ';' )|(';' ) |
'let' '(' args ')' '=' (values) 'in' ruleBody ';' |
'forall' name 'with' value 'do' ruleBody ';' |
'choose' name 'with' value 'do' ruleBody ';' |
'skip' ';' |
'rule' name '(' args ')' ';' ; |
'start_agent' name '(' args ')' ';' ;
name : Letter (Letter | Digit)*;

```

Здесь `value` — это выражение, которое может содержать константы, переменные, арифметические и логические операции, операции над множествами, получение значений функций для заданного набора аргументов.

Для выполнения заданных условий утверждения разбор по грамматике нужно проводить, исходя из состояний, нерекурсивно, поэтому для нашего случая подходит реализация интерпретатора на базе алгоритма LL(1) анализа (рисунк 3.3). Как видно, этот алгоритм последователен, использует стек и может быть реализован конечным автоматом с состояниями в виде шагов алгоритма. Таблица анализатора для принятия решения по текущему нетерминалу в стеке и прочитанному символу строится один раз с использованием функций *first* и *follow*.

Для выполнения собственно действий по интерпретации в грамматику добавляются специального вида нетерминальные символы — дельта функции, которые определяют семантические правила по анализу и интерпретации.

Согласно нашей реализации модели, имеется набор общих переменных, которые доступны из пользовательского кода в состояниях, они хранятся в хэш-таблице (имя=значение, общего типа `Object`). Это позволяет хранить любые

структуры данных, в том числе стек для разбора грамматики и таблицы интерпретатора. В такой таблице для каждого правила `rule` хранится его название, позиция тела правила в исходном файле, значения параметров и локальных переменных; для каждой функции по ее имени хранится еще одна хэш-таблица, содержащая хэш для набора аргументов, которому соответствует значение этой функции.

Поскольку семантический код выполняется не последовательно, а по мере извлечения из стека, то для передачи значений между дельта-функциями используется еще один стек.

Сам LL-анализ проходит в двух режимах – в режиме анализа и в режиме интерпретации. В режиме анализа проверяется синтаксическая правильность, создаются записи в таблице интерпретатора, содержащие позиции правил в исходном коде. Режим интерпретации включается при обработке правила `initialDefinitions` или `mainRule`, если первое отсутствует. Первоначальные значения $f(s_1 \dots s_n) := t_0$ обрабатываются как вычисления правой части t_0 и запись в хэш таблицу по имени функции этого значения на хэше от данного набора аргументов.

С правила `main` начинается интерпретация правил с использованием семантических дельта-функций:

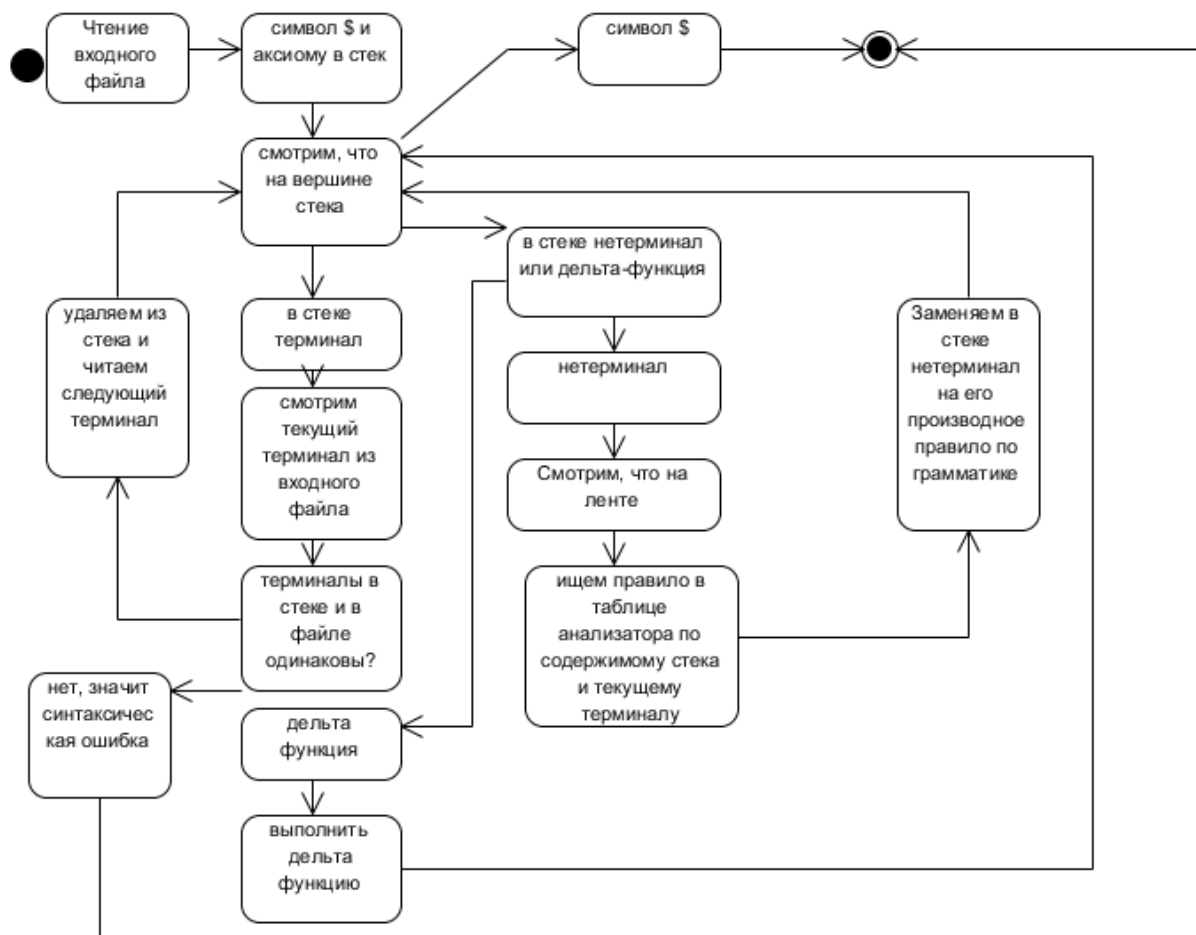


Рисунок 3.3 – Алгоритм LL(1) анализа

- `name '(' args ')' ':= ' value ';'` обрабатывается как поиск в таблице функций имени функции, заданной лексемой `name`, вычисления значений аргументов, вычисление значения выражения, заданного `value`, и поиск в таблице функций по ключу имени функции, далее по ключу хэша от значений аргументов запись значения выражения. Запись значения выражения происходит после операции блокировки ресурса по имени переменной и операции разблокировки после записи значения.
- `'seq' '{' (ruleBody*) '}'` - простое последовательное применение, без семантических дополнений.
- `'par' '{' (ruleBody*) '}'` - параллельное исполнение тела правила. Для каждого тела правила `ruleBody` создается подавтомат, который выполняет

тот же самый алгоритм LL-анализа с интерпретацией, но имеет свой стек для разбора грамматики, в который помещается аксиома `ruleBody`, и свой указатель на позицию в исходном файле, которая указывает на позицию тела правила; выполняется операция создания потока, который выполняет указанный подавтомат, и после каждой операции `fork` осуществляется переход в состояние ожидания, где применяется операция ожидания всех созданных потоков (`join`).

- `'if' value 'then' ruleBody ('else' ruleBody ';') | (';')` - вычислить значение выражения `value`. Если оно истинно, то выполнить тело правила в режиме интерпретации. Иначе выполнить тело в режиме синтаксического анализа, и, если ветвь `else` задана, то провести ее интерпретацию (при истинном условии, наоборот, просто анализ), если ветвь `else` не задана и значение выражения ложно, то вернуть стек в первоначальное положение, переместить указатель в файле на начало условия, и осуществлять анализ и возврат назад, пока условие не станет истинным.
- `'let' '(' args ')' '=' (values) 'in' ruleBody ';'` - вычислить значения переменных, заданных `args`, у текущего правила, записать их в таблицу имя=значение, проинтерпретировать тело правила и после этого убрать ранее добавленные значения переменных у текущего правила.
- `'forall' name 'with' value 'do' ruleBody ';'` - вычислить выражение, заданное `value`, и в цикле перебирать все его возможные значения (если тип выражения — множество, значениями будут его элементы; если тип целое — то весь диапазон значений целого типа; если массив - то элемент массива), и, для каждого *i*-го элемента применить операцию создания потока. Данный поток будет выполнять подавтомат, имеющий свой стек для работы с грамматикой с записанной на вершине аксиомой `ruleBody`, также свой указатель в файле, который перемещен на начало тела правила. Поток имеет копию по ссылке значений таблицы переменных для текущего правила, при этом в переменные добавлено «имя переменной `name` = зна-

чение i -го элемента». После применения операции создания потоков для всех элементов осуществляется переход в состояние, в котором применяется операция ожидания всех созданных потоков.

- `'choose' name 'with' value 'do' ruleBody ';' -` вычислить выражение, заданное `value`, и случайно выбрать один из элементов, принадлежащий выражению, записать в таблицу переменных для текущего правила «имя переменной=выбранное значение», проинтерпретировать `ruleBody` и удалить из множества его переменных значение для переменной, заданной лексемой `name`.
- `'start_agent' name '(' args ')' ';' -` вычислить аргументы, применить операцию создания потока с подавтоматом, который имеет свой стек для разбора грамматики, в котором записана аксиома `ruleBody`, а также свою таблицу для переменных для правила, заданного `name`, в которую записаны значения аргументов; также подавтомат имеет свой указатель в файле, указывающий на начало тела правила `name`.

Таким образом, показано, как при использовании переходов, операций создания и ожидания потоков, ожидания и освобождения ресурсов, а также набора переменных, общих в состояниях, реализовать интерпретатор для языка, описывающего машину абстрактных состояний (ASM), следовательно, можно говорить о возможности реализации любого параллельного алгоритма с использованием реализации нашей модели.

Следствие из 3.2. Для любого параллельного алгоритма можно построить предложенную в пункте 2.3 математическую модель. Для доказательства возьмем любой параллельный алгоритм. Согласно утверждению 3.2, для него можно построить практическую реализацию (модель в виде классов). Согласно 3.1, такая реализация эквивалентна математической модели с точностью до описания взаимодействия. ■

Утверждение 3.3. Любой распределенный алгоритм можно реализовать

при помощи предложенной многокомпонентной модели.

Доказательство. Собственно, распределенный алгоритм — это параллельный алгоритм, параллельные части которого работают не на одном, а на разных вычислительных узлах. Нужно обеспечить, чтобы параллельный алгоритм работал корректно в распределенной среде.

Рассмотрим интерпретатор для машины ASM (см. доказательство утверждения 3.2), поддерживающий работу агентов в распределенной системе. Каждый агент представляет собой распределенный компонент системы и выполняет удаленно некоторое правило. Агенты могут взаимодействовать друг с другом, изменяя и получая значения общих функций на некотором наборе аргументов.

Агент, интерпретирующий правило `main`, имеет особый статус - роль синхронизатора при доступе к значениям функций. Для синхронизации значений общих функций f_i создается отдельный поток, который ждет сообщения-запросы на получение и изменение значений функций, блокирует ресурс, связанный с именем функции, ищет значение функции для заданного набора аргументов в хэш таблице, изменяет или отдает значение с помощью сообщения, разблокирует ресурс. Основной поток интерпретирует исходный файл, начиная с начальных значений функций (правило `initialDefinitions`) как показано в доказательстве 3.2. Все запросы на получение или обновление значений функций проводятся также через обмен сообщениями со своим дочерним потоком.

При запуске других агентов (при выполнении правила `ruleBody` с термином `start_agent`), им осуществляется отправка сообщения о начале работы. Физически они располагаются по некоторому адресу (можно расширить грамматику, указав при запуске агента правило, которое он выполняет, и адрес в сети, где он находится). Удаленный агент также представляет собой интерпретатор, который интерпретирует исходный код программы, определяющей машину ASM, имея в стеке для разбора грамматики правило `ruleBody` и указатель на тело правила, которое должен выполнять данный агент.

Первоначально все интерпретаторы находятся в состоянии ожидания сообщений от интерпретатора, выполняющего `main`, о начале работы. Далее, они переходят в состояние интерпретации своего правила. Если оказалось, что требуется получить значение общей функции или установить его, то агенты отправляют сообщение агенту `main`, и либо блокируются до окончания операции, если в данный момент происходит чтение значения или его обновление другими удаленными агентами, либо получают или изменяют общие данные.

Таким образом, распределенный алгоритм реализуется при помощи параллельного с синхронизацией значений общих данных и может быть реализован при помощи многокомпонентной реализации нашей модели.

Следствие из 3.3. Для любого распределенного алгоритма можно построить предложенную в 2.3 математическую модель. Для доказательства возьмем любой распределенный алгоритм. Согласно утверждению 3.3, для него можно построить практическую реализацию (многокомпонентную модель в виде классов). Согласно утверждению 3.1, такая реализация эквивалентна математической модели с точностью до описания взаимодействия. ■

3.4 Процесс разработки и тестирования на основе моделей

Предлагается стандартные на сегодня промышленные процессы разработки ПО, такие, как Agile (гибкий), Rational Unified (рациональный унифицированный), расширить фазой тестирования на основе разработанных моделей для функциональности, реализованной на каждом этапе разработки (представлено на рисунке 3.4), а саму фазу разработки — дополнить описанием модели (рисунок 3.5).

Рассмотрим применение тестирования на основе моделей в процессе разработки и основные используемые в процессе артефакты (документы, являющиеся результатом деятельности в процессе разработки). Первоначально разработка или итерация разработки начинается со сбора требований к системе. Здесь могут быть использованы спецификации и UML диаграммы, исходя из которых можно говорить о примерной структуре нашей модели для системы. По-

сле выяснения требований происходит разработка системы (написание кода, модульное тестирование, и одновременное описание модели). После завершения разработки необходимо провести тестирование по модели реализованной функциональности.

Тестирование на основе модели предлагается проводить двумя способами:

- Статическое тестирование (верификация и симуляция) по модели, когда мы имеем модель системы и полностью заменяем систему на её модель. С реальным кодом системы, так и с выполняемыми компонентами, получившимися в результате компиляции исходного кода, работа не производится.
- Динамическое тестирование проводится на скомпилированной рабочей системе, с целью проверки её поведения относительно построенной модели.

При обнаружении некоторой ошибки в процессе тестирования возможны варианты:

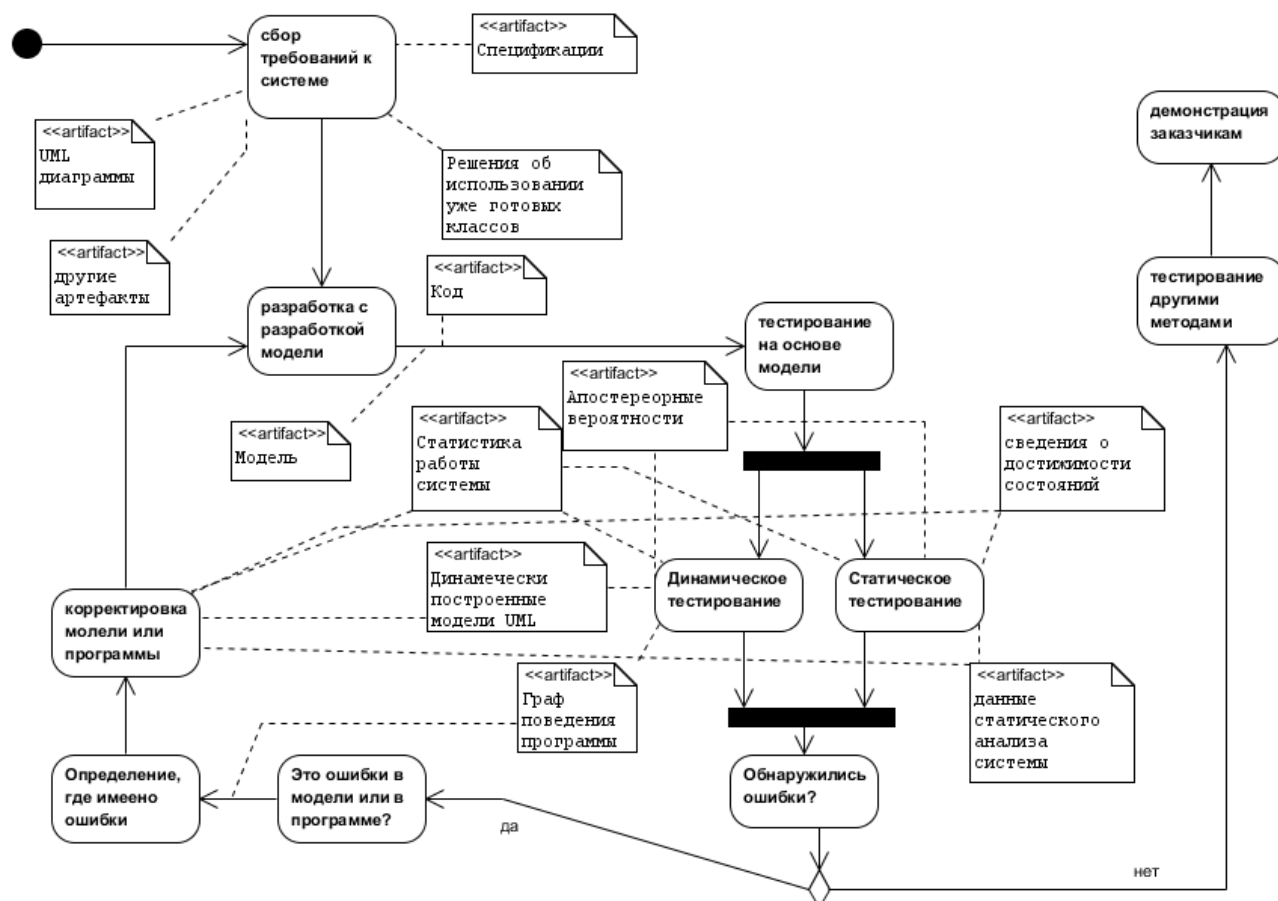


Рисунок 3.4 – Итерация процесса разработки

- Допущена ошибка в программе, т.е. программа не соответствует описанной модели.
- Несоответствующее поведение — это ошибка в описании модели, т.е. модель неправильно описывает систему.

Помочь понять правильно истинную причину ошибки могут артефакты, генерируемые автоматически (рисунок 3.4) в процессе тестирования. После определения места ошибки производится корректировка модели или кода системы и процесс тестирования повторяется.

Мы предлагаем интеграцию описания модели в процесс разработки с помощью сред разработки (IDE), в которых разработчики привыкли реализовывать свои программы. На сегодняшний день популярные среды разработки Eclipse и Microsoft Visual Studio имеют модульную структуру и средства для разработчиков, с помощью которых возможно написать расширения для данных сред, которые добавляют новые меню, средства управления проектом, средства компиляции, средства для управления кодом в редакторе и другое. Для повышения эффективности разработки с использованием модели реализован графический интерфейс для создания и редактирования параметров моделей с последующей генерацией представления модели на языке описания в виде сущностей XML, а также в виде экземпляров классов объектно-ориентированной реализации модели. Добавление в такие среды средства работы с моделью позволяет практически применить моделирование в рассматриваемый процесс разработки ПО. На рисунке 3.5 показана технология параллельной разработки N взаимодействующих компонентов системы с описанием модели. В конце фазы разработки мы должны получить работоспособный на уровне отдельных методов код (проверен модульным тестированием) и модель для кода, кото-

рая далее будет использована для проведения функционального тестирования на основе построенной модели статическим и динамическим методами.

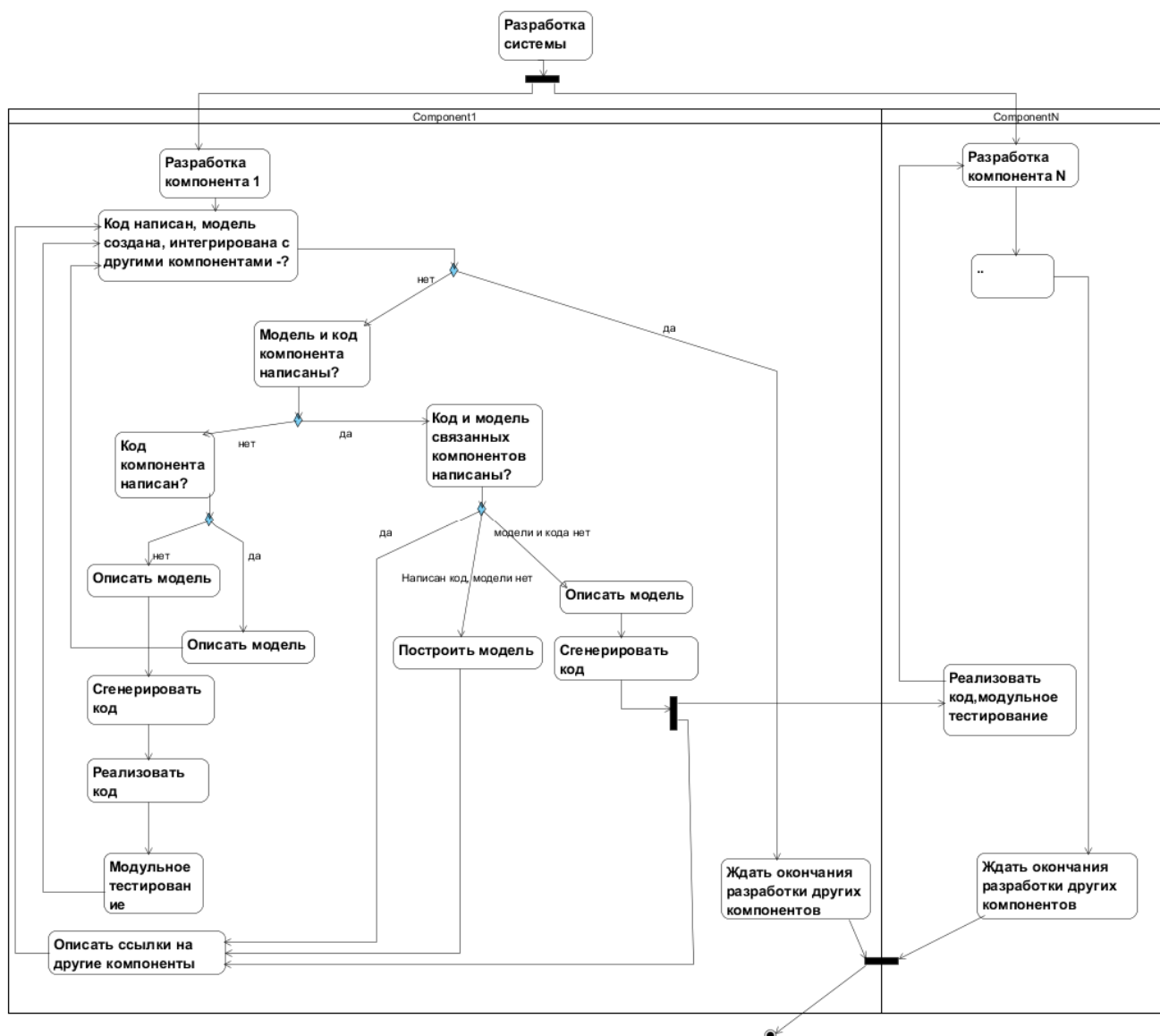


Рисунок 3.5 – Фаза разработки и описания модели

На рисунке 3.6 показаны роли разработчиков проекта при разработке на основе моделей. Если система была реализована ранее по обычной технологии и у разработчиков есть желание переделать ее в терминах состояний, операций и переходов с использованием системы классов объектно-ориентированной реализации, предложенной в приложении В, предлагается следующая технология рефакторинга:

1. Описать модель в условиях, когда код системы уже написан, путем внедрения XML тегов с описанием модели в код с использованием разработанных средств-расширений для интегрированной среды разработки.
2. По XML модели создать модель в виде экземпляров классов модели (см. доказательство утверждения 3.1).
3. Протестировать систему по модели статическими и динамическими методами.
4. Написать код, который работает в состояниях, переходах и при применении операций и реализует ожидаемое поведение системы с опорой на прежний программный код.
5. Для каждого пользовательского кода в состоянии, переходе и операции применить модульное тестирование.
6. Протестировать новую систему по модели статическими и динамическими методами, убедиться в одинаковых результатах с тестированием, проведенным в пункте 3.

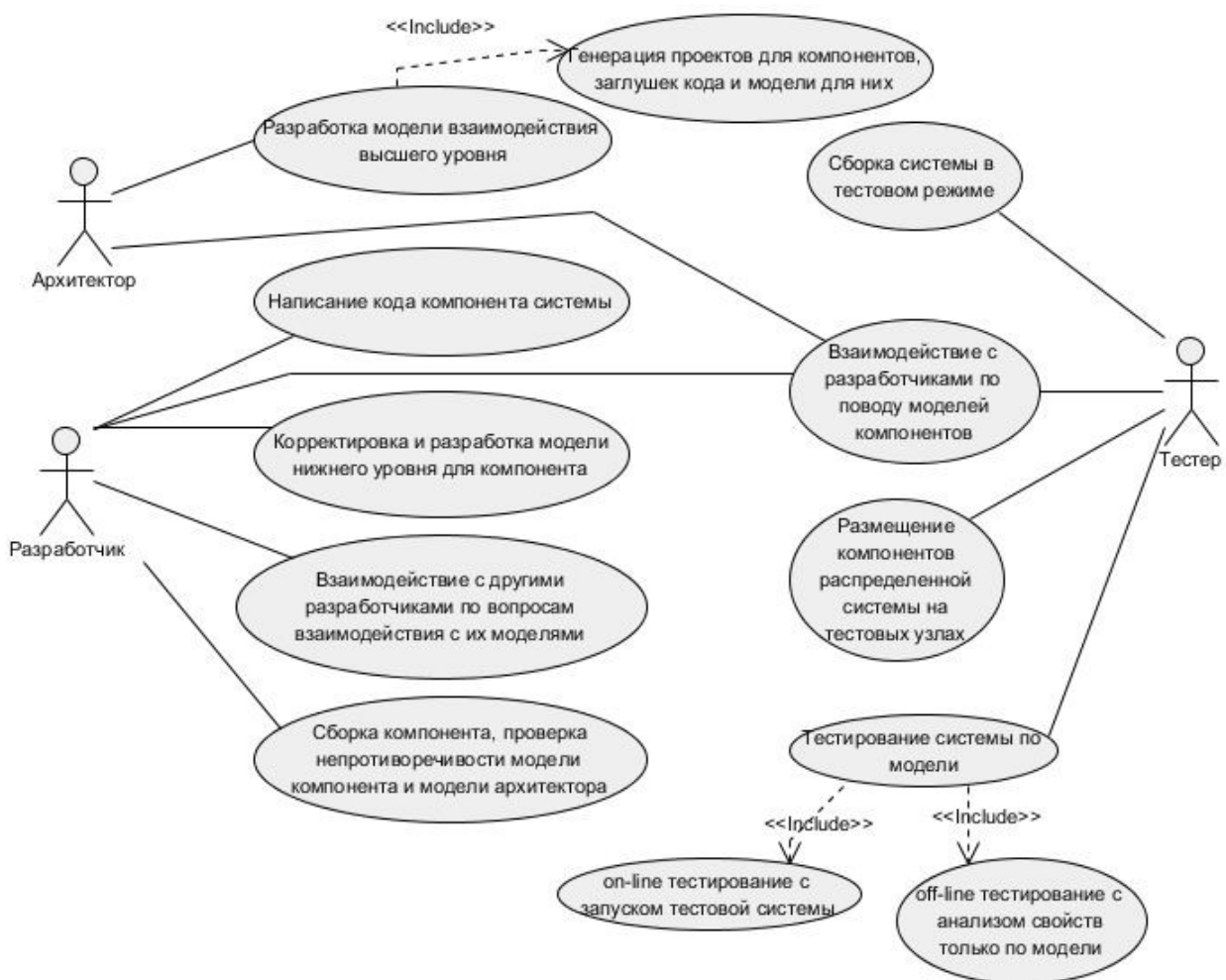


Рисунок 3.6 – Роли при разработке и тестировании

3.5 Статическая верификация модели

Статическое тестирование (верификация) позволяет оценить правильности взаимодействия в системе без ее запуска, дав ответ на вопросы по поводу выполнимости основных ожидаемых свойств модели. На рисунке 3.7 приведены основные свойства системы, которые можно установить по ее модели без ее запуска.

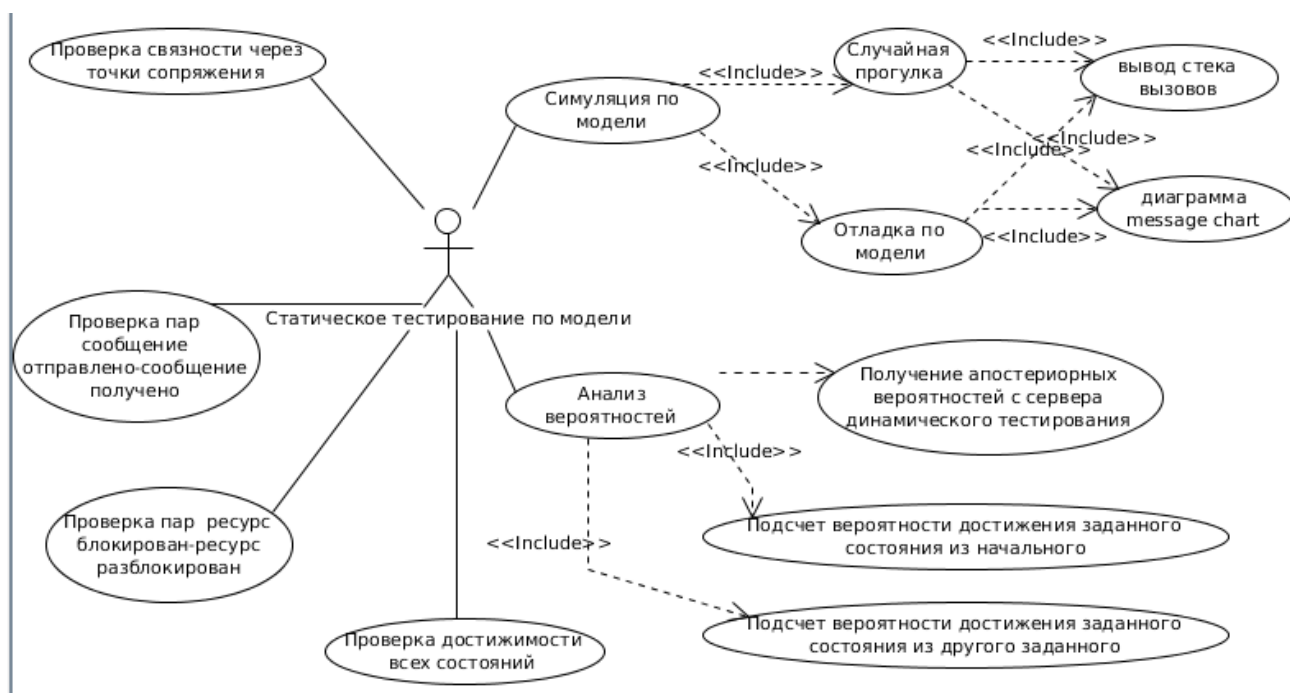


Рисунок 3.7 – Задачи статического тестирования

На сегодняшний момент в сфере проверки правильности программ сделано достаточно много, особенно в области Model checking. Популярным верификатором и симулятором многопоточных систем является продукт Spin, входным данным к которому являются модели на специальном языке Promela. Данный верификатор для загруженной модели позволяет провести симуляцию, как случайную, так и с пользовательским выбором вариантов, а также доказательство корректности или некорректности модели с поиском контр-приме-

ра, с заданием требований к модели в виде предикатов темпоральной логики.

Использование аппарата Model checking возможно при работе с предлагаемой моделью. Для этого ее достаточно преобразовать в модель, описываемую на языке Promela. С использованием нашего подхода пользователь-тестировщик, работая с нашей моделью с помощью разработанных средств, может даже не догадываться, какие модели используются внутри реализации для проведения тестирования.

Теорема. Для модели многокомпонентной системы, рассмотренной в 2.3, существует адекватная модель, описываемая на языке Promela.

Доказательство выполнено методом построения, способ преобразования рассмотренной модели в модель на Promela рассмотрен в приложении Д. ■

Решение задач статического тестирования выполняется на основе эквивалентной модели и правил темпоральной логики.

1. Симуляция по модели.

Сначала многокомпонентная модель на основе автоматов с возможностью создания потоков преобразуется в процессы Promela, которые действуют исходя из недетерминированного выбора следующего состояния (приложение Д), после чего можно проводить симуляцию по модели с использованием средств и режимов верификатора Spin:

- а) случайная прогулка проводится путем случайного выбора траектории поведения модели, осуществляется с учетом вероятностей переходов и операций (приложение Д.1);
- б) отладка по модели проводится запуском верификатора в интерактивном режиме, чтением его вывода, отображением с помощью графических средств взаимодействия с пользователем вариантов продолжения на шаге, требующим недетерминированного выбора перехода или операции, и записью варианта в поток ввода верификатора.

2. Анализ вероятностей.

Вероятности могут быть заданы как из модели, так и выгружены с сервера динамического тестирования после запуска реальной системы на выполнение и использованы для воспроизведения сценария реальной работы системы на модели.

а) Для подсчета вероятности достижения заданного состояния $s_x \in S$ из начального, прежде всего, нужно обеспечить, чтобы одна из случайных траекторий работы системы проходила через нужное состояние s_x . Это можно реализовать при помощи задания предиката, который предполагается истинным во всех состояниях: $state \neq s_x$ (здесь $state$ — глобальная переменная, в которой сохраняется текущее состояние, для каждого главного потока компонента и любого побочного потока она своя; в предикате используется переменная для именно того компонента и потока, в котором находится состояние s_x), и запуска верификатора в режиме доказательства правильности, при этом данный предикат становится ложным при очередном выборе траектории с достижением состояния s_x , и верификатор Spin возвращает данную траекторию как контр-пример. Далее, с этой траекторией, обязательно содержащей заход в состояние s_x , запускается верификатор уже в режиме симуляции. Модель для верификатора генерируется таким образом, чтобы после каждого перехода или операции производить подсчет условной вероятности в каждый текущий момент методом произведения текущей вероятности на накапливаемую, притом накапливаемая вероятность сохраняется для каждого из потоков, являющихся родительскими для потока, содержащего состояние s_x . При достижении состояния s_x выводится произведение накопленной вероятности текущего потока и всех его родительских, эти данные читаются из потока вывода верификатора и становятся доступными как результат.

б) Для подсчета вероятности достижения заданного состояния $s_x \in S$ из другого заданного $s_y \in S$ необходимо действовать аналогичным спосо-

бом что и в а), только необходимо установить траекторию, проходящую через оба состояния. Предикатом, который должен быть опровергнут, является $(state \neq s_y) U (G state \neq s_x)$, где U — темпоральный оператор Until, а G — темпоральный оператор «всегда». Сама вероятность вычисляется как частное накапливаемых вероятностей для s_x и s_y .

3. Проверка достижимости всех состояний.

Такая проверка осуществляется для каждого состояния $s_i \in S$, путем ожидания контр-примера для предиката $(state \neq s_i)$. Если верификатор возвращает ошибку, то состояние s_i достигается, при этом как контр-пример можно найти путь из начального состояния в s_i .

4. Проверка связности через точки сопряжения.

Связность через точки сопряжения — ожидаемый пользователем набор соответствий подавтомата (набора состояний) одного компонента или потока с подавтоматом (состояниями) другого.

Пусть имеется точка сопряжения, связывающая некоторые логические наборы состояний A и B , $cp \in CP: \{s_{A1}, s_{A2}, \dots, s_{An}\} \leftrightarrow \{s_{B1}, s_{B2}, \dots, s_{Bm}\}$.

Тогда проверка связности представляет собой проверку предиката

$$G ((state_A == s_{A1} \parallel state_A == s_{A2} \parallel \dots \parallel state_A == s_{An}) \rightarrow$$

$(state_B == s_{B1} \parallel state_B == s_{B2} \parallel \dots \parallel state_B == s_{Bm})).$ Здесь $state_A$ и $state_B$ — те состояния, которые идентифицируют либо текущее состояние потока main компонента, где находится группа состояний A или B соответственно, либо одного из побочных потоков, где эти группы состояний могут быть объявлены.

При ошибке верификации выдается контр-пример, в котором показывается, как можно нарушить неправильно описанную связность между точками сопряжения.

5. Проверка, что для каждое отправленное сообщение получено.

Для каждого сообщения генерируется переменная $status_message_i$, которая может содержать два значения – $sent$ (сообщение отправлено) и $received$ (получено), значения изменяются после отправки и после получения сообщения из канала. Проверяется истинность предиката

$$G ((status_message_i == sent) \rightarrow F (status_message_i == received)),$$

где G и F — темпоральные операторы соответственно «всегда» и «хотя-бы раз в будущем» (forall и global соответственно).

6. Проверка, что каждый заблокированный ресурс разблокируется.

Ресурсы res_i задаются переменными, принимающими значение 1 (ресурс заблокирован) или 0 (ресурс разблокирован, приложение Д.4), соответственно, предикат для проверки условия будет иметь вид

$$G ((res_i == 1) \rightarrow F (res_i == 0)).$$

Средствами, которые могут помочь пользователю в процессе обнаружения ошибки и понять ее причины, получаемые в процессе статического тестирования, являются диаграмма сообщений и стек вызовов.

- Диаграмма сообщений (message sequence chart) — стандартная диаграмма языка UML, показывает изменение состояний процессов с течением времени и их связи через сообщения.
- Стек вызовов выводит все состояния, применение операций в том порядке, в котором они произошли в результате симуляции по модели.

На рисунке 3.8 показан разработанный прототип подключаемого модуля в среде Eclipse для статического тестирования по модели с окнами диаграммы сообщений и стеком вызовов.

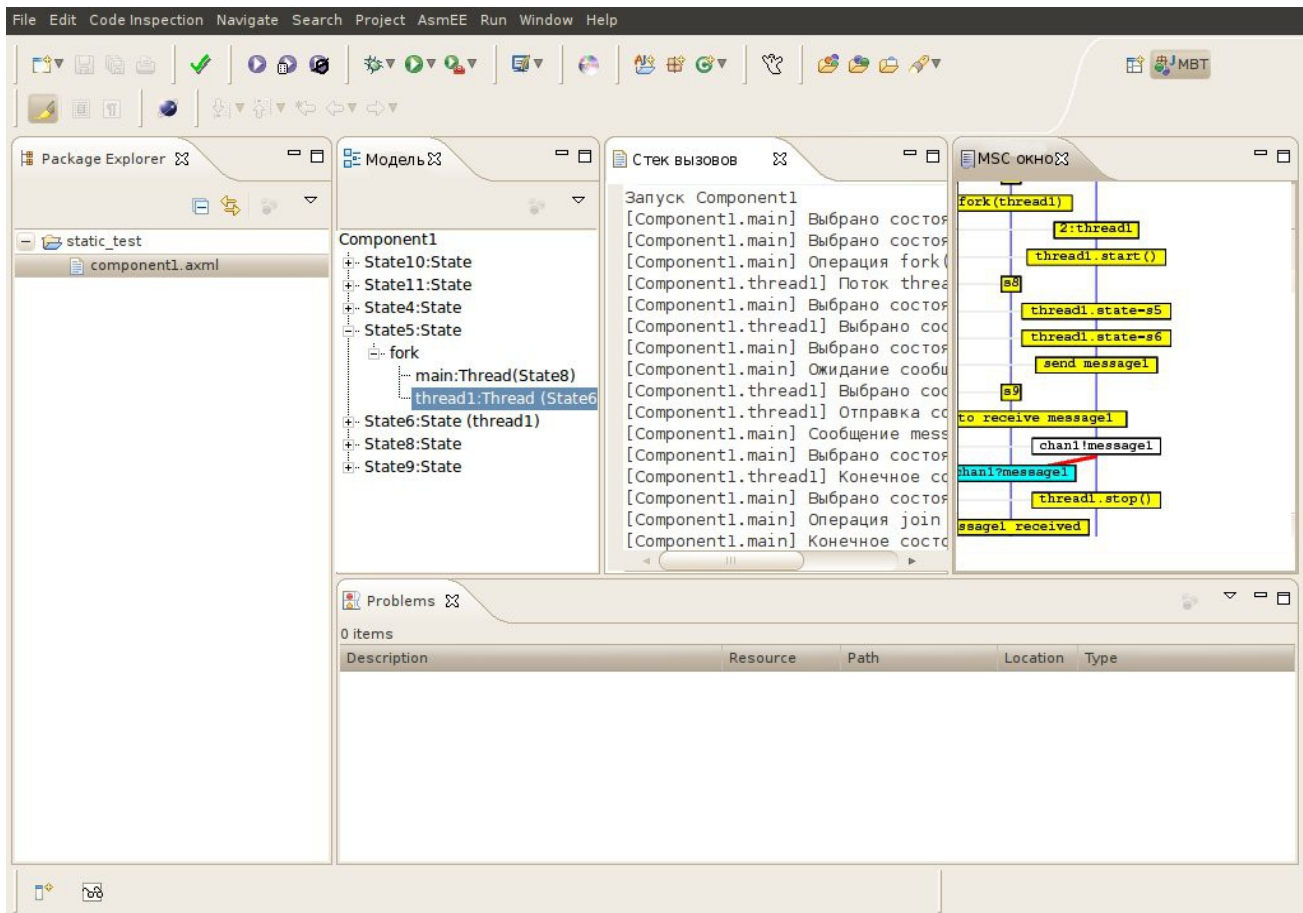


Рисунок 3.8 – Стек вызовов и диаграмма сообщений

3.6 Динамическое тестирование по модели

Динамическое (on-line) тестирование предполагает построение динамической модели ($M_{дин}$) работающей системы и сравнение ее с описанной моделью ($M_{стат}$), причем если $M_{дин} \subseteq M_{стат}$, то данный сеанс тестирования прошел успешно, иначе модели не соответствуют друг другу, что связано либо с найденными ошибками в программе, либо с ошибками в описании модели. Динамическое построение модели позволяет отследить момент начала несоответствия.

Комплексное построение модели всех компонентов системы осуществляется динамически на сервере системы тестирования с использованием протокола TCP/IP. Для обеспечения отсылки данных о проходящих событиях в модели реализовано два метода:

- а) если код системы был написан отдельно от модели, а модель была определена в комментариях к коду — выполняется дополнительный

сгенерированный код на целевом языке программирования в точках описания модели (рисунок 3.9), он взаимодействует с тестирующим сервером в процессе работы компонента, посылая все события на сервер, где они регистрируются в базе данных. Вставка дополнительного кода осуществляется препроцессором при сборке проекта.

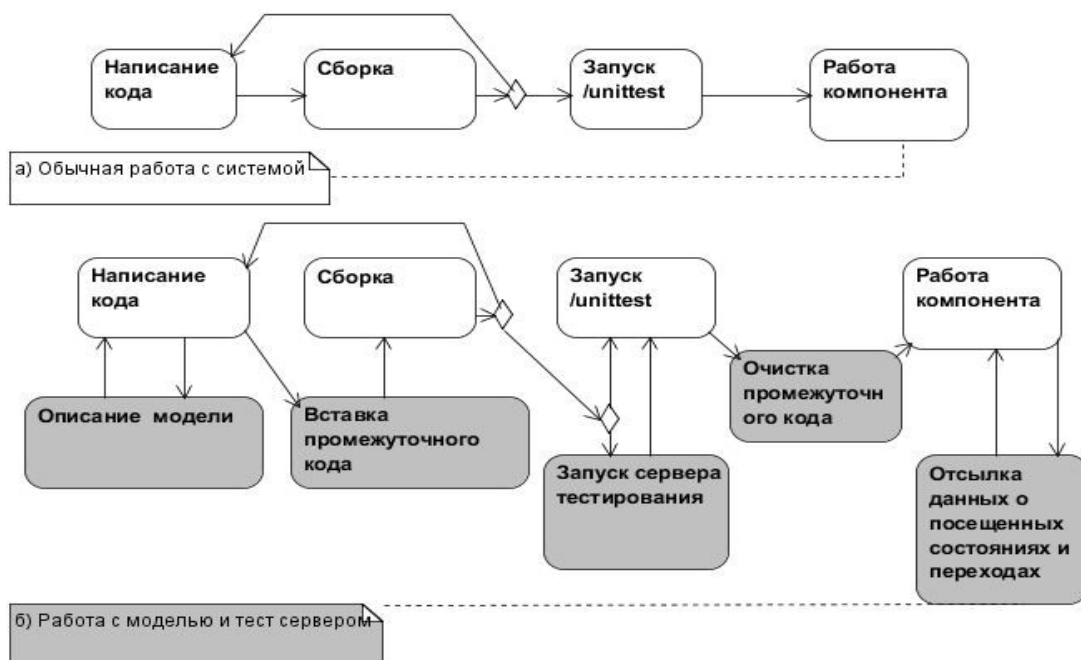


Рисунок 3.9 – Дополнения для on-line тестирования

б) если код системы реализован с использованием классов объектно-ориентированной модели, представленными в приложении В, то вставка промежуточного кода не требуется, поскольку отсылку состояние на сервер в процессе работы системы способны передавать сами классы прозрачно для пользователя.

В процессе тестирования после разворачивания тестируемой системы по тестовым узлам (см. *BoundNodes*), запуска сервера, задания для него активной статической модели $M_{стат}$ и начала взаимодействия, на сервере начинается построение динамической модели системы $M_{дин}$.

Динамическая модель представляет собой коллекцию данных о посещенных состояниях и проведенных операциях:

$$M_{\text{дин}} = \{ (time, A \in A^*, s \in S, thread \in T, ot \in (Op \cup Trans), data) \},$$

где

time – время, когда событие произошло;

A – автомат, в котором происходит событие, компонент распределенной системы;

s – текущее состояние, в котором находился компонент на момент начала события;

t – текущий поток;

ot – производимая операция или переход;

data – данные операции или перехода (посылаемое/принимаемое сообщение, блокируемый/разблокируемый ресурс, создаваемые/ожидаемые потоки, возникающее событие или исключение при переходе).

Естественно, предварительно часы сервера и всех компонентов системы должны быть синхронизированы.

Методика сбора данных показана на рисунке 3.10, задачи тестирования — на рисунке 3.11.

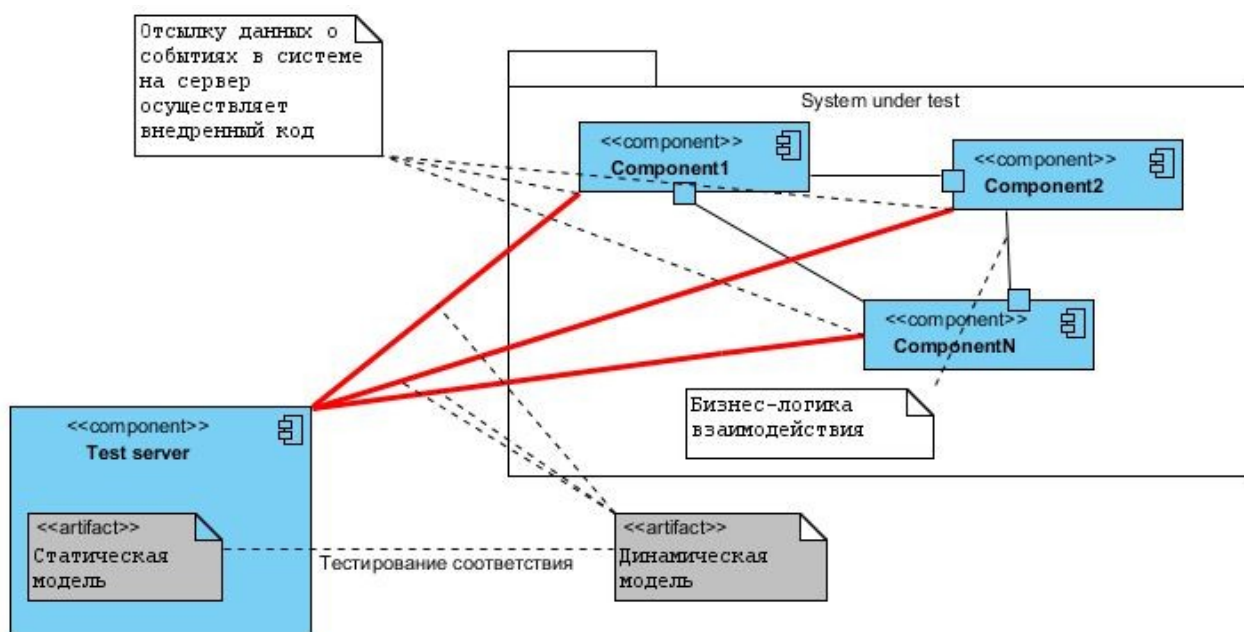


Рисунок 3.10 – On-line тестирование

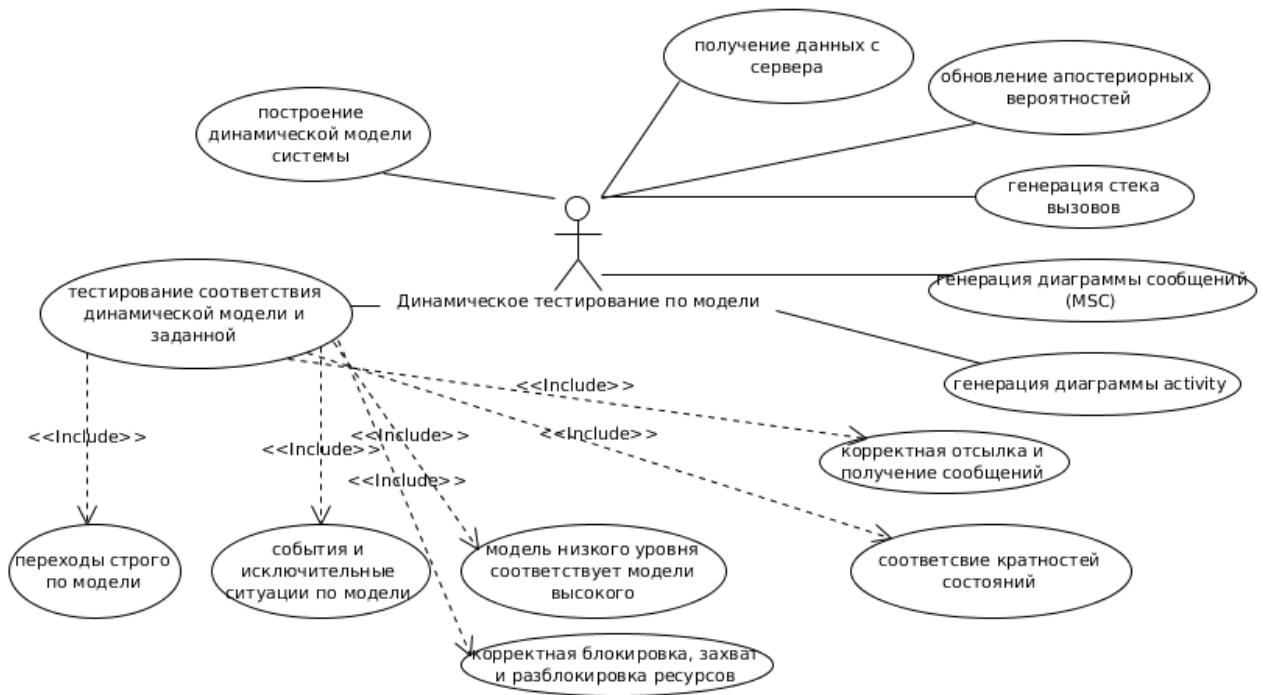


Рисунок 3.11 – Задачи On-line тестирования

Тестирование соответствия (conformance-testing), включает проверку следующих условий:

- Переход из заданного состояния в одно из заданных:

$$\forall A \in A^* : S(A) = S \ \& \ \forall s_1 \in S \ \& \ Next^1(s_1) \stackrel{\delta}{=} (S_{next} \subseteq S) \Rightarrow s_1 \rightarrow s_2, s_2 \in S_{next}.$$

Переходы из состояния в состояние производятся строго по модели. Осуществление перехода в состояние, которое не соответствует функции переходов автомата, говорит об ошибке в потоке управления.

- Срабатывание событий и исключительных ситуаций.

При переходах:

$$\forall A \in A^* : S(A) = S \ \& \ \forall s_1 \in S \ \& \ Next^1(s_1) \stackrel{\gamma}{=} (S_{next} \subseteq S) \Rightarrow s_1 \rightarrow s_2, s_2 \in S_{next};$$

при отсылке сообщений:

$$M_{stat} : \exists msg \in Msg, \exists S_1 \in S, \exists S_2 \in s, send \subseteq (\delta \wedge \gamma) : S_1 \times T \times P \rightarrow (S_2 \times msg) \vee E \Rightarrow \\ M_{din} : S_1 \rightarrow S_2 \vee S_1 \rightarrow S_3, S_3 = first(A_E);$$

при блокировке ресурса:

$$M_{stat} : \exists S_1, S_2, block \subseteq (\delta \vee \gamma) : S_1 \times T \times N \times P \times Res \rightarrow (S_2 \times 1 \times Res) \vee E \Rightarrow \\ M_{din} : S_1 \rightarrow S_2 \vee S_1 \rightarrow S_3, S_3 = first(A_E).$$

При переходе из состояния в состояние может осуществляться генерация событий и исключительных ситуаций. События могут происходить при невозможности отправить сообщение, тайм-ауте при ожидании освобождения ресурса в соответствии с моделью.

- Соответствие модели верхнего уровня моделям нижнего уровня по сообщениям и кратностям.

$$\forall c_i \rightarrow c_j, c_i \in Cn, c_j \in Cn \Rightarrow \exists msg = (s_{from}, s_{to}, msgid, type, txt), s_{from} \in A_i, \\ s_{to} \in A_j, c_i = (A_i, n_i), c_j = (A_j, n_j).$$

Модель верхнего уровня ($A_{structured}$) описывает возможные взаимодействия через отправку/прием сообщений, модели компонентов более низкого уровня описывают обмен конкретными сообщениями между заданными состояниями. Если заданный порядок обмена нарушен, регистрируется ошибка взаимодействия.

- Корректность создания и завершения потоков, кратности состояний

$$\forall T_1 \dots T_k, fork : P \times S_{fork} \rightarrow (S_1 \times T_1) \times \dots \times (S_k \times T_k) \Rightarrow N(S_1) = N(T_1) = N(S_1) + 1 \dots \\ N(S_1) = N(T_k) = N(S_k) + 1;$$

$$\forall T_1 \dots T_k, join : (S_1 \times T_1) \times \dots \times (S_k \times T_k) \rightarrow S_{join} \times T_{join} \Rightarrow N(S_1) = N(T_1) = N(S_1) - 1 \\ \dots N(S_1) = N(T_k) = N(S_k) - 1.$$

При создании потока и моделировании этого действия на низком уровне абстракции проверяется число ожидаемых и созданных потоков. Превышение заданной кратности для состояний говорит либо о неправильности потока управления, либо о возможных высоких нагрузках.

- Правильность отсылки и приема сообщений

$$\forall msg, send : P \times S_{from} \times T_{send} \times P \rightarrow (S_{sent} \times msg) \vee E_{not_sent}, \\ msg = (S_{from}, S_{to}, type, text) \exists receive : P \times T_{receive} \times S_{to} \times msg \rightarrow S_{received} \vee E_{not_sent}.$$

Для любого отправленного сообщения проверяется, дошло ли оно до получателя и правильно ли обрабатывается потеря сообщения.

- Критические секции, связанные с блокировкой внешних ресурсов

$$\forall S, S_1 \geq S \geq S_2, S_1 : block(res), S_2 : unblock(res), res \in Res \Rightarrow N(S) = 1.$$

Если блокируется ресурс одним потоком /компонентом и производится попытка доступа к нему другим потоком/компонентом, то проверяется, что доступ к ре-

сурсу имеет только его владелец. Кратность состояния в критической секции равна 1.

- Контроль возникающих блокировок — бесконечных ожиданий (deadlocks).

$$deadlock \Rightarrow \exists T, T_1, Host(R) = T_1, T_1 \neq T, block : T \times R \& \\ block : T_1 \times R_1, Host(R_1) = T.$$

Дедлок — это бесконечное ожидания объектом *Obj1* освобождения ресурса, когда захвативший этот ресурс *Obj2* сам ожидает освобождения *Obj1*. Цикл в графе ожидания ресурсов сигнализирует о наличии дедлока.

- Связность компонентов через точки сопряжения (соответствие состояний, в каких находится один компонент, состояниям, в которых находится другой компонент) не нарушается.

$$\forall CP(A_1, A_2) \Rightarrow \forall S_1 \in S(A_1) \exists S_2 \in S(A_2) : \exists \text{ время } t, S_t(A_1) = S_1 \& S_t(A_2) = S_2.$$

Если определены точки сопряжения, то проверяется, действительно ли выполняется соответствие связанных состояний.

При обнаружении несоответствия поведения динамической и статической модели системы необходимо анализировать причины ошибки. Для этого тестировщику-аналитику предоставляются следующие средства:

- Детальная трассировка посещенных состояний и примененных в процессе операций (statetrace). В процессе тестирования запоминается информация о всех действиях в модели. Допустим, несоответствие модели и системы выявлено в состоянии s_{err} у компонента A_{err} . При этом все компоненты, моделируемые автоматами A_i , начинают работу с состояний $first(A_i)$ в соответствии с функциями переходов, каждое действие в каждом компоненте при этом запоминается, пока ошибка не будет выявлена и процесс сравнения не будет остановлен в состоянии s_{err} у компонента A_{err} .
- Диаграмма последовательности обмена сообщениями (statechart), являющаяся одной из стандартных в языке графического моделирования UML,

строится автоматически и отражает записанную тестирующим сервером последовательность обмена сообщениями между компонентами системы и их потоками.

Кроме того, на тестирующем сервере производится сбор статистики работы системы и вычисление апостериорных вероятностей всех возникающих действий. Вычисляются реальные вероятности уже совершенных переходов и операций, которые могут быть использованы далее при статическом тестировании как исходные.

Также на сервере собирается статистика по наиболее часто совершаемым переходам и использованию межкомпонентных связей и задержек при взаимодействиях (время перехода из состояния в состояние), и аналитик может отслеживать эти параметры как в режиме реального времени, так и после окончания сеансов тестирования.

Задание к главе 3

1. Проанализировать и обсудить адекватность предлагаемой модели
2. Предложить способы оценки адекватности своей предложенной модели
3. Оценить способы динамического и статического тестирования по модели
4. Предложить алгоритмы реализации своих способов тестирования как по предложенной в пособии модели, так и по собственной
5. Оценить эффективность тестирования предложенными способами
6. Оценить способность практического внедрения предложенных методологий в процесс реальной разработки.

Список использованной литературы

1. Armstrong J. Programming Erlang: Software for a Concurrent World / J. Armstrong. – Raleigh, North Carolina Dallas, Texas: The Pragmatic bookshelf, 2007. – 519 p.
2. Baker P. Model-Driven Testing. Using the UML Testing Profile / P. Baker, Z. Dai, J. Grabowski, Ø. Haugen, I. Schieferdecker, C. Williams. – Berlin: Springer-Verlag, 2008. – 176p.
3. Barnett, M. The Spec# Programming System: An Overview [Электронный ресурс] / M. Barnett, K. Rustan, M. Leino, W. Schulte. – Redmond, WA, USA: Microsoft Research, 2004. – 21 p. Режим доступа- <http://research.microsoft.com/~leino/papers/krml136.pdf>.
4. Blass, A. Play to Test [Электронный ресурс] / A. Blass, Y. Gurevich, L. Nachmanson, M. Veanes. Technical Report MSR-TR-2005-04. – USA: Microsoft Research, 2005. – 26 p. Режим доступа – <http://research.microsoft.com/research/pubs/view.aspx?type=Technical%20Report&id=851>.
5. Blass, A. Abstract State Machines Capture Parallel Algorithms: Correction and Extension [Электронный ресурс] / A. Blass, Y. Gurevich. – USA: Microsoft Research, 2007 – 29 p. Режим доступа – <http://www.math.lsa.umich.edu/~ablass/corr-final.pdf>.
6. Boerger E. Abstract state machines. A method for high-level system design and analysis / E. Boerger E., R. Staerk. – USA: Springer, 2003. – 448 p.
7. Böllert, K. On Weaving Aspects / K. Böllert, A. Moreira, S. Demeyer. In Proceedings of the Workshop on Object-Oriented Technology (June 14 - 18, 1999) Eds. Lecture Notes In Computer Science, vol. 1743.– London: Springer-Verlag, 1999. – p. 301-302.
8. Brown A. An introduction to Model Driven Architecture [Электронный ресурс] / A. Brown. Режим доступа –

<http://www.ibm.com/developerworks/rational/library/3100.html>.

9. Campbell, C. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer [Электронный ресурс] / C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, M. Veanes. Technical Report MSR-TR-2005-59. – USA: Microsoft Research, 2005. Режим доступа – <http://research.microsoft.com/research/pubs/view.aspx?type=Technical%20Report&id=912>.

10. Campbell, C. Testing Concurrent Object-Oriented Systems with Spec Explorer. Extended Abstract [Электронный ресурс] / C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, M. Veanes. – Redmond, USA: Microsoft Research, 2005. – 5 p. Режим доступа – [http://www.win.tue.nl/ipa/archive/springdays2006/Veanes\(FM2005\).pdf](http://www.win.tue.nl/ipa/archive/springdays2006/Veanes(FM2005).pdf).

11. Gerth R. Concise Promela Reference [Электронный ресурс] / Rob Gerth, 1997. Режим доступа – <http://spinroot.com/spin/Man/Quick.html>.

12. Eclipse Modeling Framework Project (EMF) [Электронный ресурс] / Режим доступа – <http://www.eclipse.org/modeling/emf/>.

13. Eclipse.org home [Электронный ресурс] / Режим доступа – <http://eclipse.org/>.

14. El-Far I. Model-based Software Testing [Электронный ресурс] / I. El-Far, J. Whittaker, Florida Institute of Technology, Encyclopedia on Software Engineering (edited by J.J. Marciniak). – USA: Wiley, 2001. – 22 p. Режим доступа – http://www.geocities.com/model_based_testing/ModelBasedSoftwareTesting.pdf.

15. Extensible Markup Language (XML) 1.0 [Электронный ресурс] / W3C Recommendation of 26 November 2008.

16. Gargantini A. Metamodel-based Simulator for ASMs [Электронный ресурс] / A. Gargantini, E. Riccobene, P. Scandurra, Italy – 21p. Режим доступа — <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.153.2512&rep=rep1&type=pdf>.

17. Klein G. Formal verification of an OS kernel /G. Klein, G. Heiser, J.

Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwoodse. Proceedings of the 22nd ACM Symposium on Operating Systems Principles. USA: ASM, 2009. – 336 p.

18. Google Chrome: новый веб-браузер для Windows [Электронный ресурс] Режим доступа – <http://www.google.com/chrome/index.html?hl=ru>.

19. Graphical Modeling Project (GMP) [Электронный ресурс] / Режим доступа – <http://www.eclipse.org/modeling/gmp/>.

20. Gronback R. Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit / R. Gronback. – USA: Addison-Wesley Professional, 2009. – 736 p.

21. Hutton G. Programming in Haskell / G. Hutton. – Cambridge: Cambridge University Press, 2007. – 170 p.

22. JavaTESK: Быстрое знакомство. Документация [Электронный ресурс] / Режим доступа – www.unitesk.ru.

23. Java™ Platform Debugger Architecture (JPDA) [Электронный ресурс] / Oracle Java SE Documentation. Режим доступа – <http://download.oracle.com/javase/6/docs/technotes/guides/jpda/>.

24. Kiczales, G. Аспектно - ориентированное программирование/ G. Kiczales, J. Lamping, A. Mndheker, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. European conference on Object-Oriented programming, June. – Finland: Springer-Verlag, 1997 – 19 p. Режим доступа- <http://www.javable.com/columns/aop/workshop/02/index.pdf>.

25. Knudsen J. Parsing XML in J2ME [Электронный ресурс] / J. Knudsen. Sun developer network. Article. Режим доступа - <http://developers.sun.com/mobility/midp/articles/parsingxml/>.

26. Laddad, R. AspectJ in Action. Practical aspect-orientic programming/R. Laddad. – USA: Manning Publications Co., 2003. – 513 p.

27. LINQ. .NET framework developer center [Электронный ресурс] / Режим

доступа – <http://msdn.microsoft.com/en-us/netframework/aa904594.aspx>.

28. McConnell S. Code complete. Second Edition / S. McConnel. – USA: Microsoft press, 2004 – 960 p.

29. Microsoft Visual Studio [Электронный ресурс] / Режим доступа – <http://www.microsoft.com/visualstudio/ru-ru/default.mspx>.

30. Nachmanson, L. Optimal Strategies for Testing Nondeterministic Systems [Электронный ресурс] / L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, W. Grieskamp. – Redmond: Microsoft Research, 2004. – Режим доступа — [http://research.microsoft.com/~schulte/Papers/OptimalStrategiesForTestingNondeterministicSystems\(ISSTA2004\).pdf](http://research.microsoft.com/~schulte/Papers/OptimalStrategiesForTestingNondeterministicSystems(ISSTA2004).pdf).

31. Object Management Group - UML [Электронный ресурс] / Unified modeling language. Режим доступа – <http://www.uml.org/>.

32. OMG Unified Modeling Language (OMG UML), superstructure [Электронный ресурс] / OMG Document Number: formal/2009-02-02. Режим доступа — <http://www.omg.org/cgi-bin/doc?formal/2009-02-02.pdf>.

33. On-the-fly, LTL model checking with Spin [Электронный ресурс] / Режим доступа — <http://spinroot.com/spin/whatispin.html>.

34. Reed R. Domain of use SDL and MSC [Электронный ресурс] / R. Reed. Telecommunications Software Engineering Limited. Режим доступа – <http://sdl-forum.org/sdl2000present/sdl2000new.zip>.

35. Schultz S. Overview of TTCN-3 [Электронный ресурс] / S. Shultz. – ETSI Secretariat. Режим доступа – http://www.ttcn-3.org/doc/ETSI_TTCN3_Tutorial.ppt.

36. Schmidt, D. Model-driven Engineering [Электронный ресурс] / D. Schmidt. – IEEE Computer Society, 2006. – 7 p. Режим доступа – <http://www.cs.wustl.edu/~schmidt/PDF/GEI.pdf>.

37. SDL [Электронный ресурс] / Режим доступа - <http://sdl-forum.org/SDL/index.htm>.

38. State-Driven Game Agent Design [Электронный ресурс] / Режим доступа — http://www.ai-junkie.com/architecture/state_driven/tut_state1.html.
39. Technical WAP 2.0 [Электронный ресурс] / OMA Technical section. Режим доступа — http://www.openmobilealliance.org/tech/affiliates/wap/technical_wap2_0-20021106.zip.
40. The Java TM Tutorials [Электронный ресурс] / Sun microsystems, 1995-2008. Режим доступа — <http://java.sun.com/docs/books/tutorial>.
41. Turing A. On computable numbers, with an application to the Entscheidungsproblem / Turing A. Proceedings of the London Mathematical Society, Series 2, 42. — London, 1936 — pp. 230-265.
42. UML Testing Profile. Version 1.0 [Электронный ресурс] / Object management group. Режим доступа — <http://www.omg.org/docs/formal/05-07-07.pdf>.
43. Zhayong W. Test Advanced Guide. Advanced topics on using Google C++ Testing Framework [Электронный ресурс] / W. Zhayong. Режим доступа — <http://code.google.com/p/googletest/wiki/GoogleTestAdvancedGuide>.
44. Бар Р. Язык Ада в проектировании систем / Р. Бар. пер. с англ. — М.: Мир, 1988. — 315 с.
45. Бейзер Б. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и системы/ Б. Бейзер. — СПб: Питер, 2004. — 320с.
46. Бек К. Расширения Eclipse. Принципы, шаблоны и подключаемые модули/К. Бек, Э. Гамма. — М: Кудиц-образ, 2005. — 384 с.
47. Бек К. Экстремальное программирование: разработка через тестирование / К. Бек. — СПб.: Питер, 2003. - 224 с.
48. Буч Г. Язык UML. Руководство пользователя / Г. Буч, Д. Рамбо, А. Джекобсон. — М.: ДМК пресс, 2007. — 496 с.

49. Буч Г.. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Второе издание [Электронный ресурс] / Буч Г. Rational Санта-Клара, Калифорния. Режим доступа - <http://vmk.ugatu.ac.ru/book/buch/index.htm>
50. Вентцель Е.С. Теория случайных процессов и ее инженерные приложения / Е.С. Вентцель, Л.А. Овчаров. – М.: Наука. Гл. ред. физ.-мат. Лит. – 1991. – 384 с.
51. Воробьёв Н. Числа Фибоначчи / Н.Н. Воробьёв. Популярные лекции по математике. – М.:Наука, 1978. – 144 с.
52. Выдрин А.С. Стохастические матрицы и анализ защищённости автоматизированных систем / А.С. Выдрин, А. В. Михалёв. Фундаментальная и прикладная математика 2007, т. 13, выпуск 1. – М.: МГУ, 2007 – с. 61-99.
53. Гордиенко А.В. Тестирование при оценке динамической корректности программ АСУ / А.В. Гордиенко // Программирование. – 1982. – №6. – с. 48-52.
54. Горнаков С.Г. Программирование мобильных телефонов на Java2 Micro Edition/С.Г. Горнаков. - М.: ДМК Пресс, 2004. - 336 с.:ил.
55. Грязнов Е. Тонкий клиент [Электронный ресурс] / Е. Грязнов. Мир ПК. Режим доступа – http://www.osp.ru/pcworld/2005/02/169733/_p2.html
56. Гуров В. Исполняемый UML из России [Электронный ресурс] / В. Гуров, А. Нарвский, А. Шалыто. – 7 с. Режим доступа - http://is.ifmo.ru/works/_umlrus.pdf
57. Дастин Э. Автоматизированное тестирование программного обеспечения. Внедрение, управление и эксплуатация / Э. Дастин, Д. Рэшка, Д. Пол. Перевод с англ. – М.: ЛОРИ, 2003. – 567 с.
58. Зайцев И. М. Агентно-ориентированный подход к моделированию интеллектуальных распределённых систем / Зайцев И. М., Федяев О. И. Сб. – Донецк: ДонНТУ. – 2008. – с. 337-338.

59. Иртегов Д.В. Событийно-ориентированные архитектуры. Программирование с использованием POSIX thread library. Слайды [Электронный ресурс] / Д. В. Иртегов Д.В. ООО «Сан Майкросистемс СПб» – 23 с. Режим доступа - http://edu.dgu.ru/Res/SUN/Irtegov_Tech/Lecture_9_Event_driven_architectures_slides.pdf.
60. Иртегов Д.В. Событийно-ориентированные архитектуры. Программирование с использованием POSIX thread library /Д. В. Иртегов. – ООО «Сан Майкросистемс СПб», 2006-2007.
61. Карпов Ю.Г. Model Checking. Верификация параллельных и распределенных программных систем / Ю.Г. Карпов. – Спб.: БХВ-Петербург, 2010. – 552 с.
62. Котов В.Е. Сети Петри / В.Е. Котов. – М.: Наука, Главная редакция физико-математической литературы, 1984. – 160 с.
63. Крючкова Е.Н. Математическая логика и теория алгоритмов: Учебное пособие / Крючкова Е.Н. Алт. госуд. технич. ун-т. им. И.И. Ползунова. – Барнаул: Изд-во АлтГТУ, 2003. – 275 с.
64. Кулямин В. В. Тестирование на основе моделей. Лекции [Электронный ресурс] / В.В. Кулямин. Режим доступа – <http://panda.ispras.ru/~kuliamin/lectures-mbt/>.
65. Марков А.А. Теория алгорифмов /А.А. Марков// Тр. МИАН СССР, т. 42. – М., 1954. – с. 1-374.
66. Материалы конференции Microsoft Academic Days 2005 Russia. [Электронный ресурс] / Режим доступа – <http://www.microsoft.com/Rus/AcademicDays2005/Default.aspx>.
67. Мемоизация (memoization) [Электронный ресурс] / Режим доступа – http://www.seoliga.ru/category/.net/memoizaciya_memoization.shtml.
68. Минский М. Вычисления и автоматы / М. Минский, М.: Мир, 1971. –

366 с.

69. Мышкис А.Д. Элементы теории математических моделей / А. Д. Мышкис. Изд. 3-е, исправленное. – М.: КомКнига, 2007. – 192 с.

70. Нейгель К. С# и платформа .NET 3.0 для профессионалов / К. Нейгель, Б. Ивьен, Д. Глинн, М. Скиннер, К. Уотсон. Пер. с англ. – М. ООО "И.Д. Вильямс", 2008. – 1376+416(на CD) с.

71. Новичков А. Метрики кода и их практическая реализация в Subversion и ClearCase [Электронный ресурс] / А. Новичков, А. Шамрай, А. Черников. Режим доступа – http://cmcons.com/articles/CC_CQ/dev_metrics/mertics_part_1/.

72. Основы теории вероятностных автоматов. Бухараев Р. Г./ Р. Г. Бухарев. – М.: Наука. Главная редакция физико математической литературы. 1985. – 288 с.

73. Основы тестирования программного обеспечения: Учебное пособие / В. П. Котляров, Т. В. Коликова. – М.: Интернет Университет Информационных Технологий; Бином, 2006. – 285 с.

74. Пол С. XML. Проектирование и реализация / С. Пол. пер. с англ. – М.: Лори, 2001. – 506 с.

75. Рочкинд М. Программирование под UNIX / М. Рочкинд. 2-е изд. перераб. и под. перевод с англ. – М.: Издательско торговый дом "Русская редакция"; СПб.:БХВ-Петербург, 2005. – 704 с.:ил.

76. Савин Р. Тестирование Дот Ком, или Пособие по жестокому обращению с багами в интернет-стартапах/ Р. Савин. – М: Дело, 2007. – 312с.

77. Серебряков В.А. Теория и реализация языков программирования [Электронный ресурс] / В. А. Серебряков, М. П. Галочкин, Д. Р. Гончар, М. Г. Фуругян. Лекция N4. Синтаксический анализ. Режим доступа – <http://www.intuit.ru/department/sa/pltheory/4/6.html>.

78. Старолетов С.М. Анализ моделирования оптимальных стратегий тестирования современных распределенных недетерминированных систем /

Е.Н. Крючкова, С.М. Старолетов. Технологии Microsoft в теории и практике программировании. Конференция-конкурс работ студентов, аспирантов и молодых учёных. Тезисы докладов. – Н.: Оригинал 2, 2007. – 2 с.

79. Старолетов С.М. Вопросы проектирования системы тестирования современных распределенных недетерминированных систем /Е.Н. Крючкова, С.М. Старолетов. V Всероссийская научно-техническая конференция студентов, аспирантов и молодых ученых "Наука и молодежь – 2008". Секция «Информационные образовательные технологии». Подсекция «Программное обеспечение вычислительной техники и автоматизированных систем» / Алт. гос. техн. ун-т им.И.И.Ползунова. – Барнаул: изд-во АлтГТУ, 2008 – с. 31-35.

80. Старолетов С.М. Конечный автомат с вероятностными переходами как модель распределённой программной системы/Е.Н. Крючкова, С.М. Старолетов. Математическое моделирование и краевые задачи: Труды пятой Всероссийской научной конференции с международным участием Ч.4.: Информационные технологии в математическом моделировании.- Самара: СамГТУ, 2008. – с. 129.

81. Старолетов С.М. Математическая модель для тестирования программ/Е.Н. Крючкова, С.М. Старолетов. VI Всероссийская научно-техническая конференция студентов, аспирантов и молодых ученых "Наука и молодежь – 2009". Секция «Информационные образовательные технологии». Подсекция «Программное обеспечение вычислительной техники и автоматизированных систем». / Алт. гос. техн. ун-т им.И.И.Ползунова. – Барнаул: изд-во АлтГТУ, 2009. – с. 82-92.

82. Старолетов С.М. Методология тестирования программных систем на основе построения и анализа модели программы / Е.Н. Крючкова, С.М. Старолетов. Сборник трудов VII Всероссийской научно-практической конференции студентов, аспирантов и молодых ученых «Молодежь и современные информационные технологии». Томск, 25 - 27 февраля 2009 г., ч.1. Томск: Изд-во СПБ Графикс, 2009. – с. 195-196.

83. Старолетов С.М. Моделирование связной функциональности многокомпонентных систем / Старолетов С.М., Крючкова Е.Н. Тр. VII Всерос. конф. студентов, аспирантов и молодых учёных Центральный регион. Москва, 21-22 апреля 2010г. – М.: Вузовская книга, 2010. – с. 62-63

84. Старолетов С.М. Модель для тестирования ПО / С.М. Старолетов. Материалы двенадцатой конференции по математике "МАК-2009". – Барнаул: Изд-во Алт. Ун-та, 2009. – с. 84-86.

85. Старолетов С.М. Мультиагентная модель распределенной системы для проведения model-based testing современного ПО / С.М. Старолетов, Е.Н. Крючкова. Тр. VI Всерос. конф. студентов, аспирантов и молодых учёных. Центральный регион. Москва, 1-2 апреля 2009г. – М.: Вузовская книга, 2009. – с. 37.

86. Старолетов С.М. Некоторые аспекты тестирования распределенных систем с построением модели / Е.Н. Крючкова, С.М. Старолетов. Научная сессия ТУСУР-2008: Материалы докладов Всероссийской научно-технической конференции студентов, аспирантов и молодых ученых. Томск, 5-8 мая 2008 г.: В пяти частях. Ч.2. – Томск: В-Спектр, 2008. – с. 50-53.

87. Старолетов С.М. Применение моделей при тестировании программ / Е.Н. Крючкова, С.М. Старолетов // Ползуновский альманах. – Барнаул, 2008.- № 4.- с.16-21.

88. Старолетов С.М. Проведение on-line тестирования программного обеспечения на основе построенной модели / Старолетов С.М., Крючкова Е.Н. // Ползуновский вестник № 2, 2010. – Барнаул: изд-во АлтГТУ, 2010. – с. 212-216.

89. Старолетов С.М. Разработка программного комплекса для тестирования недетерминированных распределённых систем с построением моделей / Е.Н. Крючкова, С.М. Старолетов // Ползуновский альманах. – Барнаул: изд-во АлтГТУ, 2008. – № 4. – с. 219-220.

90. Старолетов С.М. Разработка системы тестирования распределенных

приложений на основе математических моделей / С.М. Старолетов, Е.Н. Крючкова. Технологии Microsoft в теории и практике программирования. Конференция-конкурс работ студентов, аспирантов и молодых учёных. Тезисы докладов. – Н.: Оригинал 2, 2008. – 2 с.

91. Старолетов С.М. Свой взгляд на Model Based Testing распределенных недетерминированных систем / Старолетов С.М., Крючкова Е.Н. Сборник трудов региональной конференции-конкурса (Северо-Западный регион) студентов, аспирантов и молодых ученых «Технологии Microsoft в теории и практике программирования 2009» Санкт-Петербург, 17-18 марта 2009. – Изд-во Политехн. ун-та, 2009. - с. 173.

92. Старолетов С.М. Тестирование недетерминированного программного обеспечение на основе моделей / С.М. Старолетов. Тезисы докладов Международной научной конференции "X Белорусская математическая конференция". Минск, 3-7 ноября 2008 года. – с. 90.

93. Старолетов С.М. Тестирование распределенных приложений на основе построения моделей / Е.Н. Крючкова, С.М. Старолетов // Прикладная информатика. – N 6(18), 2008, – М.: Market DS Publishing, с. 124-134.

94. Старолетов С.М. Тестирование распределенных программ с построением модели программы в виде конечного автомата с вероятностными переходами / Е.Н. Крючкова, С.М. Старолетов. Материалы XLVI Международной научной студенческой конференции «Студент и научно-технический прогресс»: Информационные технологии. – Новосиб. гос. ун-т. Новосибирск, 2008. – 2с.

95. Старолетов С.М. Тестирование распределенных систем с помощью построения моделей / Е.Н. Крючкова, С.М. Старолетов. Молодежь и современные информационные технологии. Сборник трудов VI Всероссийской научно-практической конференции студентов, аспирантов и молодых ученых. Томск, 26 - 28 февраля 2008 г. – 2с.

96. Стелтинг С. Java без сбоев. Обработка исключений, тестирование, отладка / С. Стелтинг. Java Exception Handling, Testing and Debugging. – М.:Кудиц-Образ, 2005. – 464 с.
97. Учебник по языку UML - Unified Modeling Language, UML лекции и примеры, Case средства UML [Электронный ресурс] / Режим доступа — <http://www.uml-rus.ru>.
98. Фаулер М. Рефакторинг. Улучшение существующего кода/ М. Фаулер. Символ-Плюс, 2008. – 432 с.
99. Хемраджани А. Гибкая разработка приложений на Java с помощью Spring, Hibernate и Eclipse / А. Хемраджани. Пер. с англ. – М.: ООО "И.Д. Вильямс", 2008. – 352 с.
100. Шаблон Observer [Электронный ресурс] / Режим доступа – <http://javatutor.net/articles/the-observer-pattern>.
101. Шамгунов Н.Н. State Machine — новый паттерн объектно-ориентированного проектирования [Электронный ресурс]/ Н. Н. Шамгунов, Г. А. Корнеев, А. А. Шалыто. //«Информационно-управляющие системы». – 2004. – № 5. с. 13-25. Режим доступа - <http://www.ict.edu.ru/ft/003756/pattern.pdf>.
102. Щипак Ю. Win32 API. Эффективная разработка приложений / Ю. Щипак. – СПб:Питер, 2006. – 576 с.
103. Якобсон А. Унифицированный процесс разработки программного обеспечения (RUP) / А. Якобсон, Г. Буч, Дж. Рамбо. – USA:Addison-Wesley, 2002. – 493 с.

Приложение А. Пример моделирования АОП функциональности

Рассмотрим для примера мемоизацию вычисления чисел Фибоначчи по определению (при помощи рекурсии) и с использованием техники АОП (под взаимодействием тут можно понимать вызов функций).

Исходный код на языке Java, рекурсивно вычисляющий число Фибоначчи:

```
public class fib {  
    public static long f(int n) {  
        if (n==0||n==1) return 1;  
        return f(n-1)+f(n-2) ;  
    }  
    public static void main(String[] args) {  
        System.out.println(f(100)) ;  
    }  
}
```

Исходный код реализации связной функциональности с использованием АОП на языке AspectJ:

```
public aspect test {  
    pointcut point(int n) : call(long *.f(int)) &&args(n);  
    private Map _cache = new HashMap();  
    long around(int n) : point(n) {  
        Long g = (Long) _cache.get(n);  
        if (g != null) {  
            return g.longValue();  
        }  
        long ret=proceed(n);  
        _cache.put(n, ret) ;  
        return ret ;  
    }  
}
```

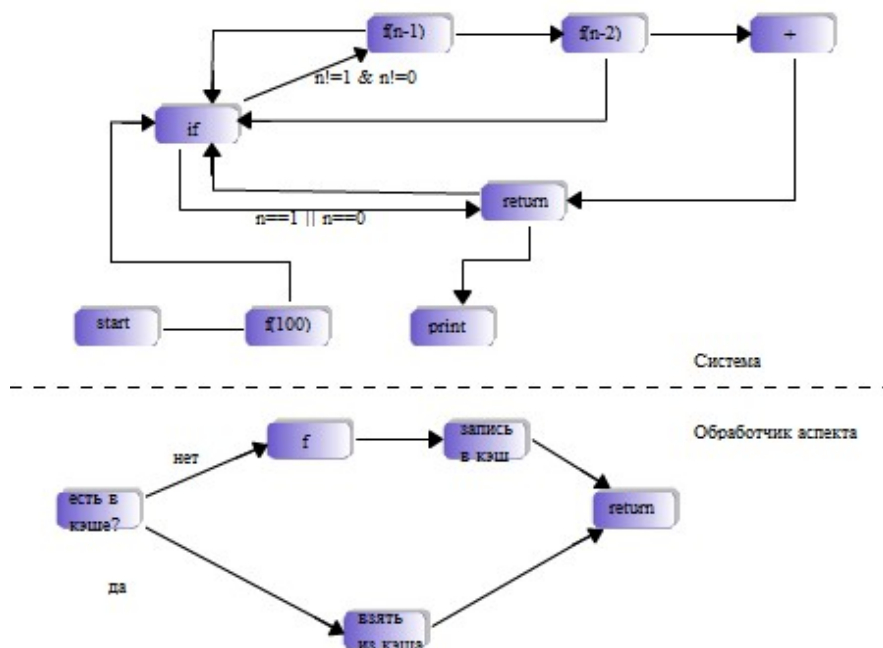


Рисунок А.1 – Модель для вычисления чисел Фиббоначи

В данном случае работает сценарий в соответствии с формулой 2.44. Аспект срабатывает при вызове функции f (поскольку определено **call(long *.f(int))**), при этом вызывается не сама функция, а обработчик аспекта (**long around(int n)**), который проверяет, вычислено ли уже значение для данного аргумента n , если да, возвращает его, а если нет, вызывает функцию f (оператор **proceed(n)**) для его вычисления и записывает вычисленное значение в кэш. После применения связанного аспекта модель будет работать следующим образом:

Приложение Б. Объектная мета-модель

По предложенной в 2.3 математической модели была спроектирована объектная мета-модель (полужирным шрифтом выделены классы, знак '→' означает наследование, после атрибута, через двоеточие приведен тип данных атрибута):

Component - Класс, инкапсулирующий компонент системы	
0..* states: State	Состояния компонента, если этот компонент работает по обычным правилам
0..* aspects: Aspect	Аспекты, если компонент задает множество аспектов, связанных с другими компонентами системы
componentType: ComponentType	Тип компонента
ComponentType - Перечисление, задающее типы компонента	
normal=0	Обычный компонент
aspect=1	Компонент, содержащий аспекты
Operation - Операция, абстрактный класс, определяет суперкласс для различных операций в модели	
p: EDouble	Вероятность срабатывания операции среди других операций и переходов в текущем состоянии.
nextState : ActiveState	Состояние, в которое осуществляется переход при успешном применении операции
SendMessage → Operation - Операция отсылки сообщения	
message: Message	Отсылаемое сообщение
timeoutEvent: TimeoutEvent	Ссылка на событие, возникающее по тайм-ауту
0..* events: Event	Ссылка на другие события, возникающие во время отсылки сообщения
Message - Сообщение	
id: EString	Идентификатор сообщения в модели
text: EString	Текст сообщения
fromState : State	Состояние, из которого посылается сообщение

toState : State	Состояние, в которое посылается сообщение
fromComponent: Component	Компонент, из которого посылается сообщение
NComponent: EInt	Номер экземпляра компонента, если сообщение имеет соответствующий тип и участвует в многокомпонентном обмене
messageType: MessageType	Тип сообщения
MessageType - Перечисление, задает тип сообщения при рассылке между кратными компонентами	
default=0	Обычное сообщение
toN=1	Сообщение экземпляру компонента с заданным номером
toAll=2	Сообщение всем экземплярам компонента
toAllButMe=3	Сообщением всем, кроме текущего
ReceiveMessage → Operation - Операция получения сообщений	
0...* messages: Message	Сообщения, которые могут быть получены данной операцией
0...* events: Event	Другие событие, которые возникают во время приема сообщения
receiveType: InType	Тип приема сообщений с нескольких источников
BlockResource → Operation - Операция блокировки ресурса	
resource: Resource	Ресурс
timeoutEvent: TimeoutEvent	Событие, которое возникает из-за таймаута в попытке заблокировать ресурс
UnBlockResource → Operation - Операция разблокирования ресурса	
resource: Resource	Ресурс
0...* events: Event	События, возникающие при разблокировке
Resource - Внешний разделяемый ресурс	
name: EString	Имя ресурса
Fork → Operation - Операция создания потоков	

0...* createdThreads : Thread	Создаваемые потоки
0...* events: Event	События, возникающие при создании потоков
Join → Operation -Операция редукции	
0...* joinedThreads : Thread	Ожидаемые потоки
timeoutEvent: Event	Событие, возникающие при таймауте ожидания
State - Состояние	
0...* transitionsFromThis : Transition	Переходы из данного состояния
0...* operationsFromThis : Operation	Операции в данном состоянии
threadId : Thread	Ссылка на поток, в котором выполняется состояние
srcFile: EString	Имя файла, где находится исходный код программы, описывающий логику работы компонента системы в данном состоянии
srcLineFrom: EInt	Строка в файле с кодом состояния- начало состояния
srcLineTo : EInt	Строка в файле с кодом состояния- конец состояния
name: EString	Название состояния
N: EInt	Кратность
subAutomatonId : SubAutomaton	Ссылка на подавтомат, в которое может входить данное состояние
firstState: EBoolean	True, если это начальное состояние?
finalState: Eboolean	True, если это заключительное состояние?
Thread - Поток	
name: EString	Имя потока
firstState : ActiveState	Начальное состояние потока
threadBody: subAutomaton	Ссылка на тело потока-подавтомат
parent: Thread	Родительский поток
Transition - Переход	
toState : ActiveState	Состояние, в которое осуществляется переход
0...* events: Event	События, которые могут возникнуть при переходе
p: EDouble	Вероятность перехода

transitionString: EString	Текстовое описание перехода
Event - Событие	
eventState : PassiveState	Состояние, в которое осуществляется переход при обработке события
p : EDouble	Вероятность события
warn: EBoolean	Флаг ошибочного события
eventName : EString	Текстовое название события
iFlag: Eboolean	Флаг перехода в заданное состояние после завершения обработки события
returnState: State	Состояние для перехода из заключительных состояний обработчика события при установленном iFlag
TimeoutEvent → Event	Событие, возникающее по тайм-ауту
timeout: EInt	Значение тайм-аута
ActiveState → State Активное состояние — в него может осуществляться переход при совершении простого перехода или операции	
PassiveState → State Пассивное состояние — переход осуществляется в результате возникновения события	
linkedEvent: Event	Связанное событие
SubAutomaton - Подавтомат	
name : EString	Название подавтомата
firstState:State	Начальное состояние подавтомата
0..* finalStates:State	Заклучительные состояния подавтомата
parent:SubAutomaton	Родительский подавтомат, если подавтоматы вложенные
Node - Узел распределенной системы	
name: EString	Название узла
adress: EString	Адрес узла в сети
type: NodeType	Тип узла
NodeType - Перечисление, определяющее тип узла, зависит от реализации (например, это	

узлы Windows, Linux, мобильные устройства, узлы mpi кластера и др.)	
BoundNodes - Расположение компонентов по узлам распределенной системы, класс определяет связку одного узла с некоторым компонентом (отношение много-ко-многим)	
nodeId : Node	Ссылка на узел
componentId : Component	Ссылка на компонент
N: EInt	Кратность компонента на узле
Aspect - Аспект	
aspectBody: SubAutomaton	Подавтомат, тело аспекта
cp : ConnectPoint	Точка сопряжения, связывает аспект с состояниями, функциональность которых нужно изменить
type: AspectType	Тип применения аспекта
AspectType - Тип применения аспекта, перечисление	
before = 0	вставка функциональности до связанных действий
after = 1	вставка функциональности после связанных действий
around = 2	вставка функциональности с заменой связанных действий
ConnectPoint -Точка сопряжения	
name: EString	название точки сопряжения
ConnectPointRelation - Связь состояний и подавтоматов через точки сопряжения	
linkToStatel : State	Для связи 1..1 или 1..n — состояние, которое связывается с другим состоянием или подавтоматом
linkToSubAutomatonl : SubAutomaton	Для связи n..1 или m..n — подавтомат, который связывается с другим состоянием или подавтоматом
linkToState2: State	Для связи 1..1 или n..1 — состояние, которое связывается с другим состоянием или подавтоматом
LinkToSubAutomaton2: SubAutomaton	Для связи 1..m или n..m — подавтомат, который связывается с другим состоянием или подавтоматом

cp: ConnectPoint	Точка сопряжения для связи
StructureAutomaton - Структурный автомат системы	
0...* components: Component	Компоненты в системе
0...* links : MessageLink	Связи между компонентами через сообщения
MessageLink - Связь в структурном автомате между компонентами системы через сообщения	
componentFrom:Component	Компонент, от которого связь
inputPinTo: InputPin	Вход компонента, к которому идет связь
inType: InType	Тип ожидания сообщений, если их несколько
messageId : Message	Ссылка на соответствующее сообщение низкого уровня
InType - Перечисление, типы сложных входов	
andInput= 0	Вход типа «И»
orInput = 1	Вход типа «Или»
InputPin - Вход компонента	
component: Component	Компонент, к которому относится данный вход
inType: InType	Тип входа
0...* inputComponentConnections: Component	Ссылка на соединенные компоненты

Приложение В. Пример использования объектно-ориентированной реализации многокомпонентной системы на языке Java

Цель данной программной реализации — создание системы классов для моделирования сложных взаимодействующих распределенных систем с многопоточностью и обменом сообщениями. Созданные экземпляры классов будут являться моделью изучаемых программных систем. Все созданные классы основываются на рассмотренной мета-модели, к которым добавлены методы для обеспечения работы и взаимодействия, и реализуют большинство из рассмотренных в 2.3 возможностей модели.

Предлагаемая система классов позволяет как моделировать работу какой-то системы с использованием вероятностных переходов, так и создавать код для реальной программной системы. Поэтому для каждого состояния, перехода и операции может быть задан пользовательский код.

Пример вызова конструкторов состояний, переходов и операций для запуска автоматов системы на выполнение:

```
public static void main(String[] args) {
    //Создать список компонентов
    List<Component> components=new LinkedList<Component>() ;
    //Создать состояния, зададим их параметры позже
    State state1=new State(),state2=new State(),state3=new State(),state4=new
State(),state5=new State() ;
    //Создать компонент
    Component component1=new Component() ;
    //Создать переход в состояние state2, без событий с вероятностью 1.0
    Transition transition12=new Transition(state2, null, 1.0, "transition 1->2");
    //Задать параметры состояния: главный поток компонента, название-state1
    //нет подавтомата, начальное состояние=true, конечное состояние=false
    //переходы из состояния- transition12, операций из состояния нет.
    state1.setParams(component1.getMainThread(),"state1", null, true, false, new Transition[]
{ transition12}, null) ;
}
```

```

//Задать код, выполняющийся в состоянии state1
state1.setCode(new IStateCodeExecutor() {
    @Override
    public void execute(Component currentComponent,
        Thread currentThread, State currentState) {
        System.out.println("Зашли в состояние "+currentState.getName());
    }
},
new IStateCodeExecutor() {
    public void execute(Component currentComponent,
        Thread currentThread, State currentState) {
        System.out.println("Выходим из состояния "+currentState.getName());
    }
});

//в state2: fork и создание потока с первоначальным состоянием state3, и
//продолжением главного потока в state5

//Создать подавтомат для потока с первоначальным состоянием state3
//и заключительным в state4
SubAutomaton subAutomatonThread2=new SubAutomaton("thread_subautomaton", state3,
new State[] {state4},null);
//создать описание потока thread2, с подавтоматом subAutomatonThread2
//и предком в виде главного потока компонента
dissertation.Thread thread2=new dissertation.Thread("thread2",
subAutomatonThread2,component1.getMainThread());
//создать операцию Fork (создание потоков) для потока thread2 с переходом
//основного потока в состояние state5 после создания без событий с //вероятностью
1.0
Fork operationForkState2=new Fork(new dissertation.Thread[] {thread2}, null,1.0,state5);
//установить параметры для state2- передать туда операцию создания потока
state2.setParams(component1.getMainThread(),"state2", null, false, false, null, new
Operations[] {operationForkState2});

```

...

```
//Установить состояния для компонента
component1.setStates(new State[] {state1,state2,state3,state4,state5}) ;

//добавить в компоненты (можно было передать также как массив)
components.add(component1) ;

//Создать структурный автомат и запустить его
StructureAutomaton A=new StructureAutomaton(components) ;
A.start() ;
}
```

В данном примере рассмотрен переход и создание потока. Данный код может быть сгенерирован по модели с переходами автоматически с использованием редакторов модели (см. Рисунок 3.1).

Приложение Г. Контекстно-свободная грамматика языка описания модели

Если модель проектируется как дополнение к программному коду, ее можно описать в качестве языка, было решено сделать это в XML виде (п. 3.2.2). Спроектированная объектная мета-модель (Приложение Б) — по сути представляет собой модель сущность-связь, описывающая классы с полями. XML атрибуты в грамматиках соответствуют этим полям классов в объектной мета-модели. Описание и тип атрибута также соответствуют сущностям из Приложения Б. Структуру XML будем задавать КС-грамматиками.

Основные свойства языка описания модели:

- Согласно нашей методологии «код и модель — одно целое» модель описывается в виде комментариев с XML тегами внутри исходного кода системы. Высокоуровневые элементы модели (компоненты и их связи через сообщения) должны быть описаны в отдельных файлах, поскольку к коду программного модуля они отношения не имеют, они определяются отдельной грамматикой.
- Язык описания модели определяется двумя грамматиками — грамматикой описания модели высокого уровня (компоненты и их связи), а также грамматикой модели низкого уровня (компонент и его состояния).
- Главным тегом для описания элемента модели низкого уровня является тег «состояние» («<state>»). Согласно рассмотренному в 2.3 определению состояния, оно связано с исходным кодом. Если посмотреть на описание атрибутов состояния в мета-модели, там можно увидеть, что там имеются ссылки на позиции в исходном коде. Данные атрибуты не заполняются при описании состояния, а заполняются автоматически, само же состояние описывается в теле программного модуля, где, по мнению разработчика имеется неделимая часть функционала системы (см. определение 2.7).
- Для всех объектов, имеющих связи «0...*» в мета-модели, действует правило — такие объекты описываются внутри соответствующих тегов.

- Модели в виде атрибутов XML и объектов являются взаимнооднозначными, поэтому для XML модели возможна генерация объектов с соответствующими атрибутами и по набору объектов возможна генерация XML модели.
- Если полем класса является объект модели, то при описании в XML виде ссылка на этот объект осуществляется по имени, либо объект описывается как вложенный.

Г.1 Грамматика, описывающая модель высокого уровня (структурный автомат)

```

system: (component|node|boundNodes|inputPin|messageLink)* ;
component: '<component' componentAttribs '>' componentBody '</component>' ;
componentAttribs : (componentAttrib '=' Value) ;
componentAttrib: 'componentType' | 'buildCommand' | 'deployCommand' ;
componentBody: ('<src file=' Value '/src>')* ;
node: '<node' nodeAttribs '/>' ;
nodeAttribs: (nodeAttrib '=' Value)* ;
nodeAttrib: 'name' | 'adress' | 'type';
boundNodes: '<bound' boundAttribs '/>';
boundAttribs: (boundAttrib '=' Value)* ;
boundAttrib: 'nodeId' | 'componentId' | 'N' ;
messageLink: '<messageLink' messageLinkAttribs '/>' ;
messageLinkAttribs: (messageLinkAttrib '=' Value)* ;
messageLinkAttrib: 'componentFrom' | 'inputPinTo' | 'inType' | 'seqNum' |
'messageText' | 'messageId' ;
inputPin: '<inputPin' inputPinAttribs '>' inputPinBody '</inputPin>' ;
inputPinAttribs: (inputPinAttrib '=' Value)* ;
inputPinAttrib: 'component'|'inType' ;
inputPinBody: '<connect inputComponent=' Value '/>' ;

```

Value: "" Name "" ;

Г.2 Грамматика, описывающая модель на уровне компонента системы

```
component: (state)* | (aspect)+ ;
state: oneState|subAutomaton ;
oneState: '<state' stateAttribs '>' stateBody '</state>';
stateAttribs: (stateAttrib '=' Value)* ;
stateAttrib: 'threadId' | 'srcFile' | 'srcLineFrom' | 'srcLineTo' |
'name' | 'N' | 'subAutomatonId' | 'firstState' | 'finalState'|'active';
stateBody: (state|operation|transition)* ;
subAutomaton: '<subAutomaton' subAutomatonAttribs '>' subAutomatonBody
'</subAutomaton>' ;
subAutomatonAttribs: (subAutomatonAttrib '=' Value)* ;
subAutomatonAttrib: 'name'|'firstState'|'parent' ;
subAutomatonBody: (finalState|oneState)* ;
finalState: '<final' 'state' '=' Value '>';
transition: ('<jump' transitionAttribs) ('/>' | ('>' transitionBody '</jump>')) ;
transitionAttribs: (transitionAttrib '=' Value)* ;
transitionAttrib: 'transitionString' | 'p' | 'toState';
transitionBody: (event)* ;
operation: (fork|join|send|receive|block|unblock) ;
operationAttribs: (operationAttrib '=' Value)*;
operationAttrib: 'p'|'nextState' ;
fork: '<fork' operationAttribs '>' forkBody '</fork>' ;
forkBody: (thread|event)* ;
thread: '<thread' threadAttribs ('/>' | ('>' threadBody '</thread>')) ;
threadAttribs: (threadAttrib '=' Value)* ;
threadAttrib: 'name'|'firstState'|'threadBody'|'parent';
threadBody: (state)* ;
```

```

join: '<join' operationAttribs '>' joinBody '</join>';
joinBody: ((('<joined' 'thread' '=' Value '>')|event))* ;
send: '<send' sendAttribs '>' sendBody '</send>';
sendAttribs: ((operationAttribs '=' Value) | (sendAttrib '=' Value) )* ;
sendAttrib: 'message' ;
sendBody: message (event)* ;
receive: '<receive' receiveAttribs '>' receiveBody '</receive>' ;
receiveAttribs: ((operationAttrib '=' Value) | (receiveAttrib '=' Value) )* ;
receiveAttrib: 'receiveType';
receiveBody: ((('<wait message=' Value ( '>' message '</wait>')| '>')|event))* ;
message: '<message' messageAttribs '>';
messageAttribs: (messageAttrib '=' Value)* ;
messageAttrib:
'id'|'text'|'fromState'|'toState'|'fromComponent'|'toComponent'|'NComponent'|'message
Type' ;
block: '<block' blockAttribs ('/>' | ( '>' blockBody '</block>' ));
blockAttribs: ((operationAttrib '=' Value) | (blockAttrib '=' Value) )* ;
blockAttrib: 'resource' ;
blockBody: (event)* ;
unblock: '<unblock' unblockAttribs ('/>' | ( '>' unblockBody '</unblock>' ));
unblockBody: (event)* ;
unblockAttribs: (operationAttrib '=' Value | unblockAttrib '=' Value)* ;
unblockAttrib: 'resource' ;
event: '<event' eventAttribs ('/>' | ( '>' eventBody '</event>' ));
eventAttribs: (eventAttrib '=' Value)* ;
eventAttrib: 'eventState'|'eventSubautomaton'|'p' | 'warn' | 'iFlag' |
'returnState'|'timeout';
eventBody: (state)* ;
aspect: '<aspect' aspectAttribs '>' aspectBody '</aspect>' ;
aspectAttribs: (aspectAttrib '=' Value)* ;

```

```
aspectAttrib: 'cp' | 'type' ;  
aspectBody: (subAutomaton|cpRelation)* ;  
cpRelation : '<cpRelation' cpRelationAttribs '>' ;  
cpRelationAttribs: cpRelationAttrib '=' Value ;  
cpRelationAttrib: 'linkToState1' | 'linkToSubAutomaton1' | 'linkToState2' |  
'linkToSubAutomaton2' ;  
Value: '"' Identifier '"'
```

В приведенных здесь грамматиках нетерминал Name — допустимая согласно спецификации XML последовательность символов атрибута.

Приложение Д. Способы получения эквивалентной модели на языке Promela в целях верификации

Рассмотрим способ создания по нашей модели адекватной модели на языке Promela (входной язык верификатора Spin). Данный язык является по сути функциональным С-подобным языком, описание модели на котором пользователю без специальной подготовки затруднительно. Предлагается подход, при котором пользователем при помощи программных средств описывается разработанная модель на разных уровнях, по которой автоматически генерируются вспомогательные модели, в том числе на языке Promela.

Д.1 Система переходов и недетерминированный выбор состояния

Для проведения алгоритмов верификации на основе моделей система Spin строит специального вида конечные автоматы, однако описание программы на языке Promela происходит по последовательному принципу, а не исходя из состояний. Мы будем реализовывать программу как последовательность переходов между состояниями, как это и описывается в нашей модели. Рассмотрим способ реализации конечного автомата на языке Promela:

```
mtype= {s1,s2,s3};  
active proctype main() {  
    mtype state ;  
    state=s1;  
    do :: (state!=s3) ->  
        if  
            ::(state==s1)-> {  
                printf("state1") ;  
                state=s2  
            }  
            ::(state==s2)->  
            {
```

```

        printf("state2");

        state=s3
    }

fi;
::(state==s3)-> break
od
}

```

Выполнимой единицей в программе является процесс (proctype). В данном примере s_1 - начальное состояние, а s_3 - заключительное, и в теле процесса main работает циклическая смена состояний, пока не будет достигнуто заключительное состояние. Допустимые состояния описываются в перечислении.

Рассмотрим теперь, как осуществить недетерминированный переход в другое состояние из заданного. В языке Promela конструкция условного выбора if выглядит следующим образом:

```

if
:: (выражение1) -> действие1
...
:: (выражениеN) -> действиеN
fi

```

При этом, если выражения истинны одновременно, действие для выполнения выбирается случайным, недетерминированным образом.

Таким образом, код для недетерминированного перехода из состояния s_1 в состояния s_2 или s_3 имеет вид:

```

state=s1;
do :: (state!=s3) ->
    if
    ::(state==s1)->    {//находясь в s1
    if

```

```

:: true -> state=s2 //выбираем s2 или s3
:: true -> state=s3
fi
}

::(state==s2)->
...

```

На этом можно и остановиться, но можно попытаться улучшить адекватность недетерминированного перехода, с учетом вероятностей переходов, которые заданы в нашей модели. Заметим, если написать одинаковые правые части для некоторых состояний, то вероятность перейти в них выше:

```

if
:: true -> state=s2 //выбираем s2 или s3
:: true -> state=s2 //s2 выбираем чаще
:: true -> state=s3
fi

```

Здесь вероятность попасть в состояние s_2 — $2/3$, а в s_3 — $1/3$. Можно предложить и обратный способ генерации числа нужных условий по заданной вероятности (с точностью до $1/10$): умножить каждую вероятность из $[0..1]$ в 10 раз и округлить до целого — столько условных вариантов для каждого перехода и сгенерировать (для уменьшения числа вариантов можно найти их наименьшее общее кратное и поделить все числа на него).

События, возникающие в нашей модели, можно также считать за вероятностные переходы из состояния по формуле условной вероятности.

Д.2 Создание и ожидание потоков

В языке Promela параллельно выполняющиеся сущности называются процессами, но в любой момент любой процесс может запустить другой процесс, так что по сути, это потоки.

Для каждого главного потока компонента в нашей модели нужно сгенерировать свой процесс main в Promela, который будет переходить по состояниям (как например, в Д.1), например, с именем «main_имя компонента»,

который должен быть помечен модификатором `active`, что означает, что процесс будет запущен при запуске моделирования. Другие процессы тоже должны быть сгенерированы по подавтоматам, осуществляющих переходы, отвечающие за логику создаваемых потоков.

Рассмотрим операцию создания потока(`Fork`) в нашей модели — процесса в Promela. Процесс запускается оператором `run`, например, вот так можно, находясь в состоянии s_4 , создать поток `thread1` и перейти в состояние s_8 :

```
proctype thread1() { //побочный процесс(поток)
  mtype state; //локальная переменная для хранения текущего состояния
  ...
}

active proctype main() { //главный процесс компонента
  mtype state ; //локальная переменная для хранения текущего состояния
  pid pid_thread1 ; //PID для создаваемого процесса
  ...
  ::(state==s4)->
    {
      pid_thread1=run thread1(); //запускаем процесс
      state=s8; //меняем состояние
    }
  ::(state==s8)->
  ...
}
```

При создании процесса оператором `run` возвращается `pid` (process identifier), номер процесса, который далее может быть использован при ожидании завершения процесса.

Рассмотрим реализацию операции редукции-завершения потоков(`Join`).

Заметим, что функции, ожидающей, когда процесс завершится, нет в языке Promela. Однако можно использовать `pid` процесса и возможности обмена сообщениями для сигнализации о его завершении.

Обмен сообщениями строится на использовании каналов, которые могут хранить некоторое количество сообщений, и операторах отсылки и приема сообщений «!» и «?». Поскольку, в общем случае, согласно описанию операции редукции, находясь в состоянии, можно ожидать несколько потоков, то используется канал с буфером, устанавливаемым равным числу ожидаемых потоков. На каждую операцию ожидания проще создавать отдельный канал.

Алгоритм операции редукции следующий: перед своим завершением каждый редуцируемый процесс посылает в канал сообщение со своим номером `pid`. Родительский процесс, который осуществляет редукцию, перед этим когда-то создал данный процесс и знает его `pid`, поэтому просто берет из канала данные и удостоверяется, что получен `pid` именно того процесса, что ожидался. Допустим, находясь в состоянии s_{11} , нужно подождать два потока(процесса) `thread1` и `thread2` и перейти в состояние s_{12} :

```
chan chan_join=[2] of {pid}; //глобальное объявление канала с буфером на 2 сообщения
...
pid pid_thread1; //объявление pid-ов в родительском процессе
pid pid_thread2;
pid var ;
...
pid_thread1=run thread1() ;
...
pid_thread2=run thread2() ;
...
::(state==s11)-> //в s11 делаем join
{
    chan_join?var; //читаем из канала 1-й pid
    (var==pid_thread1||var==pid_thread2)->
    chan_join?var; //читаем из канала 2-й pid
    (var==pid_thread1||var==pid_thread2)->
```

```

state=s12 ;//и меняем состояние
}

...
proctype thread1() {
...
chan_join!_pid //отсылка при завершении
}

proctype thread2() {
...
chan_join!_pid
}

```

Д.3 Отправка и получение сообщений

Для приема сообщений прежде всего для каждого участвующего в приеме сообщений процесса необходимо создать глобальный канал для получения сообщений и добавить перечисляемый тип с идентификаторами сообщений:

```

mtype={message1,message2}
chan chan1= [0] of {mtype} ;

```

Для обычных сообщений используются синхронные каналы (указывается 0 в качестве размера буфера). Для приема сообщений по типу «или» необходимо сгенерировать каналы с заданным числом сообщений в качестве буфера (см. пример в Д.2). Сам прием и передача сообщений осуществляются при помощи стандартных средств Promela для отсылки и приема «!» и «?», далее после такой операции осуществляется выбор следующего состояния.

Д.4 Блокировка и разблокировка ресурсов

Разделяемым ресурсом будем считать глобальную переменную. Благодаря особенностям языка Promela, выражение вида $(r==0)$ при $r \neq 0$ останавливает текущий процесс, пока r не станет равным 0 в результате работы другого процесса. На этом и основывается реализация операций BlockResource и

UnblockResource на языке Promela:

```
::(state==s1)->
    {
        //block resource r
        atomic {(r==0)->r=1};
        state=s2 ;
    }
::(state==s2)->
    {
        //unblock resource r
        r=0;
        state=s3 ;
    }
```

В данном случае в состоянии s_1 осуществляется блокировка ресурса r с возможным ожиданием, пока r не освободится. Данные операторы оформлены в блок `atomic {}`, который выполняется атомарно.

Основные положения диссертации были опубликованы и доложены на следующих конференциях и семинарах:

- Конференция-конкурс "Технологии Microsoft в теории и практике программирования" (Новосибирск, НГУ, 2007) [6];
- VI Всероссийская научно-практическая конференция студентов, аспирантов и молодых ученых "Молодёжь и современные информационные технологии" (Томск, ТПУ, 2008г) [23];
- V Всероссийская научно-техническая конференция студентов, аспирантов и молодых ученых "Наука и молодёжь-2008" (Барнаул, АлтГТУ, 2008) [7];
- Конференция-конкурс "Технологии Microsoft в теории и практике программирования" (Новосибирск, НГУ, 2008) [18];
- XLVI Международная научно-студенческая конференция "Студент и научно-технический прогресс (Новосибирск, СО РАН, 2008) [22];
- Всероссийская научно-техническая конференция студентов, аспирантов и молодых ученых "Научная сессия ТУСУР-2008" (Томск, ТУСУР, 2008) [14];
- V Всероссийская конференция "Математическое моделирование и краевые задачи" (Самара, СамГТУ, 2008) [8];
- X Международная Белорусская математическая конференция (Минск, БГУ, 2008) [20];
- Школа-семинар в Сибирском федеральном округе для участников (победителей) программы «Участник молодежного научно-инновационного конкурса» (Барнаул, АлтГТУ, 2008) ;
- Школа – семинар «Менеджмент технико-внедренческой деятельности» для победителей программы «У.М.Н.И.К.» 1 года (Томск, Высшая школа бизнеса ТГУ, 2008);

- Конкурс IT проектов Алтайской торгово-промышленной палаты (Барнаул, 2008);
- VII Всероссийская научно-практическая конференция студентов, аспирантов и молодых ученых “Молодёжь и современные информационные технологии” (Томск, ТПУ, 2009) [10];
- Конкурс-конференция студентов, аспирантов и молодых ученых Северо-Запада «Технологии Microsoft в теории и практике программирования» (Санкт-Петербург, СПбГПУ, 2009)[19];
- VI Всероссийская научно-техническая конференция «Технологии Microsoft в теории и практике программирования» для студентов, аспирантов и молодых ученых Российской Федерации (Москва, МАИ, 2009) [13];
- VI Всероссийская научно-техническая конференция студентов, аспирантов и молодых ученых “Наука и молодёжь-2009” (Барнаул, АлтГТУ, 2009) [9];
- XII Региональная математическая конференция МАК-2009 (Барнаул, АлтГУ) [12] .
- VII Всероссийская научно-техническая конференция «Технологии Microsoft в теории и практике программирования» для студентов, аспирантов и молодых ученых Российской Федерации (Москва, МАИ, 2010)[11] ;

А также в научных журналах:

- «Ползуновский альманах» (г. Барнаул) (статья, опубликованная в результате работы секции «Программное обеспечение вычислительной техники» конференции «Наука и молодежь- 2008»[15]; статья участника школы-семинара по программе УМНИК[17]);
- «Прикладная информатика» (г. Москва) [21]
- Ползуновский вестник (г. Барнаул)-вак [16].

Получено авторское свидетельство на часть разработок (авторское свидетельство на программу для ЭВМ «Редактор состояний для платформы

Eclipse»).