Анализ и трансформации исполняемых UML моделей

Е.Д. Волкова, А.Д. Страбыкин

Аннотация. В статье[1] рассмотрены конечные автоматы языка UML, представлен подход к анализу исполняемых моделей UML. На основании выборки моделей, использованных в промышленных проектах, исследованы их количественные свойства и продемонстрирована актуальность трансформации моделей. Выделены образцы, часто используемые при построении автоматов. Предложены новые трансформации, улучшающие структуру модели, описан процесс их применения к реальной системе.

1. Введение

При создании сложных инженерных систем принято использовать приемы моделирования. Сложность большинства создаваемых сегодня программных систем не уступает сложности многих инженерных сооружений, поэтому моделирование программных систем является весьма актуальной задачей. Более того, в таких концепциях, как MDA (Model Driven Architecture – архитектура на основе моделей) и MDD (Model Driven Development – разработка на базе моделей), моделям отводится центральная роль в процессе создания программного продукта. Основной идеей этих концепций является представление процесса создания программного продукта в виде цепочки трансформаций его исходной модели в готовую программную систему.

Почти во всех инструментальных средствах, воплощающих идеи MDD, в качестве языка моделирования используется язык UML (Unified Modeling Language – унифицированный язык моделирования), целиком или какие-либо его части.

UML – это язык, предназначенный для визуализации, специфицирования, конструирования и документирования программных систем. Слово «унифицированный» в названии языка означает, что UML может использоваться для моделирования широкого круга приложений от встроенных систем и систем реального времени до распределенных web-приложений. Выразительные средства языка позволяют описать систему со всех точек зрения, имеющих отношение к разработке и развертыванию.

[1] Работа выполнена при поддержке РФФИ, проект 05-01-00998-а.

В свете инициатив MDA и MDD роль моделей в жизненном цикле программного обеспечения (ПО) претерпевает значительные изменения. Если ранее моделирование рассматривалось как одно из удобных средств документирования, и, соответственно, жизненный цикл моделей был близок к жизненному циклу артефактов документации, то в последнее время работа с моделями становится все более похожа на работу с исходными кодами. Подобный подход ставит перед исследователями новые задачи исследования применимости к моделям методик и приемов работы, используемых для работы с исходными кодами. Одной из таких методик является рефакторинг.

Рефакторинг – это изменение внутренней структуры ПО, имеющее целью облегчить понимание и упростить модификацию, но не затрагивающее при этом наблюдаемого поведения. Рефакторинг, как набор методик преобразования программ, помогает решать две глобальные задачи: облегчение процесса повторного использования каких-либо компонентов программной системы и снижение расходов на поддержку и сопровождение системы. Первые рефакторинги появились в результате обобщения опыта нескольких экспертов в области объектно-ориентированного проектирования. В этом отношении рефакторинги достаточно близки к широко известным на сегодняшний день паттернам проектирования.

Существует много исследовательских работ и публикаций, посвященных методам и алгоритмам применения рефакторинга. Полноценная поддержка рефакторинга ставит перед производителями следующий ряд задач:

1. Поиск плохо спроектированных участков кода (модели), для которых требуется проведение рефакторинга;
2. Определение рефакторинга (синтез из поддерживаемых базовых рефакторингов), который следует применить;
3. Проверка или доказательство неизменности поведения системы после выполнения преобразований;
4. Реализация применения рефакторинга и, в частности, разработка пользовательского интерфейса и диалогов, поддерживающих процесс применения рефакторинга;
5. Сохранение целостности модели, то есть распространение произведенных изменений на другие части модели (диаграммы, тесты);
6. Оценка эффекта, полученного в результате применения рефакторинга.

По каждому из указанных пунктов ведутся научные разработки, но лишь в немногих из них учитывается специфика UML.

Анализ существующих UML моделей, приводимый в данной статье, показывает, что их структура сложна для понимания и содержит недостатки, которые можно было бы устранить путём проведения эквивалентных трансформаций. Особое внимание уделяется анализу и поиску методов рефакторинга для конечных автоматов языка UML, которые являются основой для полностью автоматической генерации исполняемого кода по UML-

моделям. На базе проведённого анализа и выявленных недостатков описывается новая трансформация, специфичная для UML.

2. Анализ исполняемых UML-моделей

С целью выявления особенностей использования конечных автоматов UML в реальных промышленных проектах было проведено статистическое исследование набора моделей. Все рассмотренные модели описывают поведение системы с использованием конечных автоматов, по которым можно сгенерировать исполняемый код.

Конечные автоматы UML могут описывать поведение следующих элементов исполняемых моделей:

- активный класс (active class);
- операция (operation);
- составное состояние (composite state).

В зависимости от своего происхождения, все исследованные модели UML можно разделить на два класса:

1. Модели, изначально спроектированные на языке UML (например, в таких программных системах, как Rational Rose, Telelogic Tau G2, I-Logix Rhapsody, Borland Together);
2. Модели, изначально спроектированные на языке SDL (например, в таких программных системах, как Telelogic SDL Suite, Verilog ObjectGeode) и трансформированные в UML вручную или при помощи специальных утилит (например, Telelogic Tau G2 — Import SDL).

Исполняемые UML-модели второго класса в основном описывают различного рода коммуникационные системы (то есть такие классы систем, для моделирования которых предназначен язык SDL). Исполняемые модели первого класса в связи с универсальностью языка UML описывают гораздо более широкий спектр систем.

2.1. Характеристика конечных автоматов

Общая статистика по исследованным моделям представлена в таблице 1. Перечисленные модели были заимствованы из реальных проектов коммерческих компаний. Для сбора и анализа необходимой информации был разработан дополнительный модуль к промышленной среде UML-моделирования Telelogic Tau G2.

Ожидалось, что модели, используемые в реальных проектах, будут иметь достаточно высокий уровень сложности. Тем не менее, более 90% от всех описанных автоматов содержат не более трех состояний, а доля автоматов без состояний (включающих только один начальный переход) близка к 75% (Рис. 1). Причем доля таких автоматов растет вместе с размером модели.

Название модели	Предмет моделирования	Происхождение	Объем (Кб)	Кол-во автоматов, реализующих			Общее количество	
				активные классы	опера-ции	составные состояния	пассивные типы данных[2]	Диаграммы
Aircraft Simulator	Симулятор самолета	UML	371	1	2	0	0	34
Central Interface	Система контроля доступа	UML	253	4	0	1	8	24
IOS Algorithms	Система ввода/ вывода	UML	1 611	11	41	2	70	203
Llama Simulator		UML	894	6	2	0	31	126
MMI		UML	3 267	17	20	0	45	112
MV-IOS6		UML	675	5	0	0	14	118
3gN		UML	8 660	12	175	0	2708	нет инф
ATM and Banklib	Банкомат	SDL	144	4	1	0	4	12
Local Exchange		SDL	178	3	5	0	2	13
Access Control	Система контроля доступа	SDL	281	8	1	0	10	34
DEL_REL		SDL	190	3	2	0	21	22
Inres		SDL	121	4	0	0	4	15
Mobile	Мобильный телефон	SDL	772	14	0	0	27	156
Pager	Пейджер	SDL	161	3	4	0	4	14
cc_layer		SDL	1 066	3	19	0	45	39
common Executor		SDL	1 396	5	39	0	12	89
1xevdo		SDL	21 817	21	482	0	350	1457
ATC_ENV		SDL	5 710	17	62	0	867	226
CpCallm		SDL	67 295	9	760	0	2262	1009
S		SDL	380	1	9	0	0	34
SS_RCS		SDL	20 572	3	94	0	175	547
Tarif_c7		SDL	881	2	25	0	4	104
DC2000_5		SDL	19 420	33	226	0	447	1483
23 модели		**7 - UML 16 - SDL**	**152 M**	**189**	**1969**	**3**	**7110**	**5871**
	В среднем:		**6.6 M**	**8**	**86**	**0**	**309**	**267**

Таблица 1.

Следует отметить, что автомат без состояний практически не обладает семантикой автомата и может использоваться только в качестве одной из форм записи некоторой последовательности действий, выполняемой в процессе во время начального перехода. Более того, текстовый синтаксис кажется намного более удобным средством для подобных спецификаций. Таким образом, оказывается, что в промышленных проектах примерно в половине случаев

[2] В колонке «пассивные типы данных» учитывались следующие типы: пассивный класс, тип данных (datatype), перечислимый тип (enum), синоним типа (syntype), объединение (choice)

конечные автоматы используются не по своему прямому назначению. Причиной этому может служить недостаточный уровень владения инструментом у разработчиков модели или же, например, требование унифицировать все описания поведенческих аспектов системы с использованием для этого конечных автоматов.

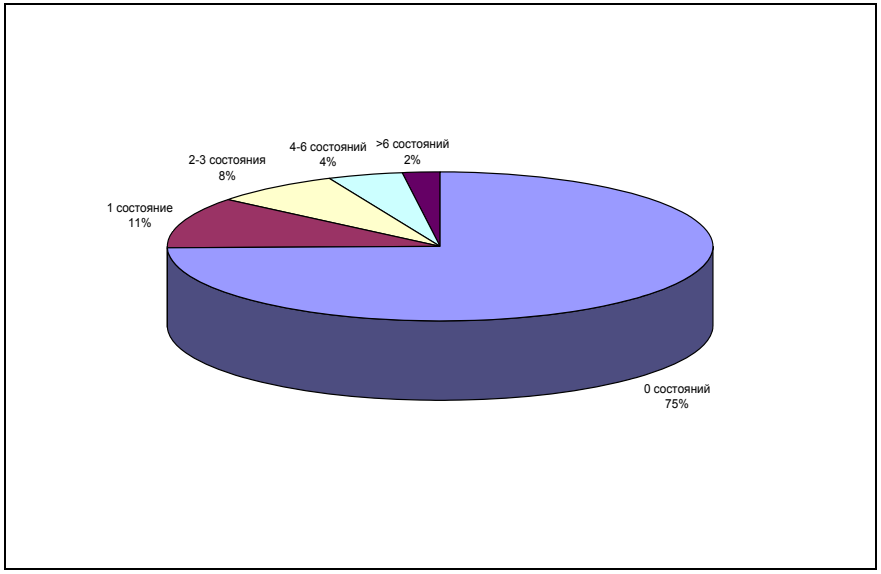

Рис. 1. Количество состояний в конечных автоматах

На основе полученных данных использование конечных автоматов без состояний может быть объяснено следующим образом. В рассмотренных моделях операции практически не обладали семантикой состояний, поэтому 99% операций описывались автоматами без состояний, вырождаясь в императивную последовательность действий. Таким образом, использование автоматов для спецификации операций, как правило, не оправдано, и, тем не менее, широко применяется на практике.

Если рассмотреть автоматы, реализующие классы, то распределение количества состояний значительно изменяется (Рис. 2). Для спецификации классов практически не используются автоматы без состояний, в то время как преобладают автоматы, имеющие одно состояние. Такая структура характерна для классов, не обладающих сложной внутренней логикой, а реализующих некоторый сервис для других компонентов системы. В единственном имеющемся состоянии, которое очень часто носит имя “Idle” или “Wait”, класс ожидает запроса на выполнение какой-либо операции. Получение запроса инициирует срабатывание перехода, в процессе которого выполняются необходимые действия. По завершении обработки класс вновь возвращается в исходное состояние.

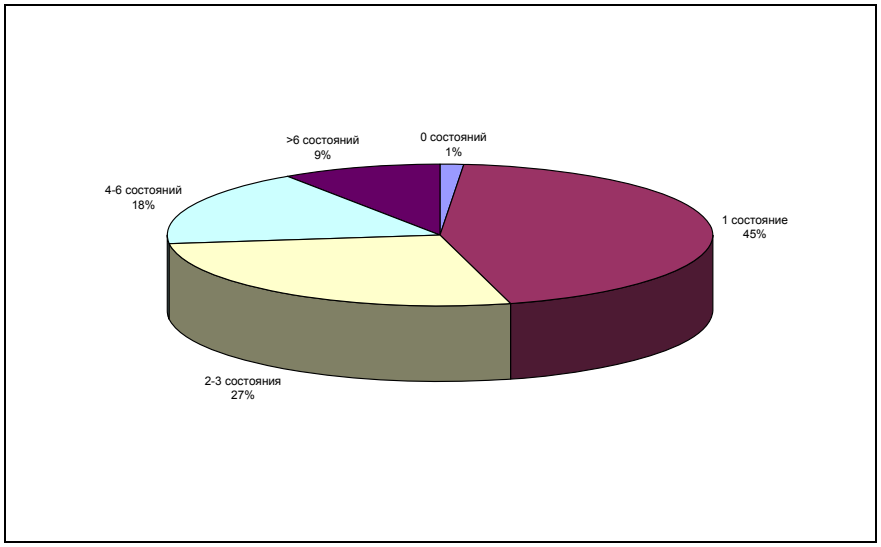

Рис. 2. Количество состояний в конечных автоматах, реализующих классы

Автоматы, специфицирующие иерархические состояния, составили чуть менее 2% от всех обнаруженных автоматов и были найдены всего лишь в нескольких из рассмотренных моделей, что позволяет сделать вывод об их достаточно редком использовании, несмотря на их выразительную мощность. Причиной тому может служить тот факт, что составные состояния не являлись частью языка SDL до его версии SDL-2000. Большинство крупных промышленных моделей SDL, впоследствии трансформированных в UML, было разработано до того, как появился новый стандарт SDL-2000.

На Рис. 3 приведена статистика количества переходов, которые могут сработать в каждом из состояний автомата. И здесь снова 84% процента состояний достаточно просты в понимании, так как имеют не более 6 переходов. Однако состояния с большим числом переходом могут заметно затруднить понимание автомата, а их доля приближается к 15%; более того, как правило, эти состояния являются ключевыми в понимании алгоритмов, заложенных в конкретный автомат.

Таким образом, в среднем, автомат, реализующий класс, содержит 3 состояния и около 12 переходов и 4 диаграмм, при этом около 90% автоматов содержат не более 6 состояний, и, следовательно, их понимание не должно вызывать серьезных затруднений у разработчиков. Однако внутренняя логика работы системы, как правило, реализуется оставшимися 10%, среди которых встречаются автоматы, насчитывающие до 30 состояний. Вполне очевидно, что умственные затраты на понимание такого автомата достаточно велики; соответственно, значительно затрудняется процесс его модификации, поиска

ошибок и проч. Поэтому средства, уменьшающие сложность автоматов, сохраняя их внешние свойства, действительно востребованы на практике.

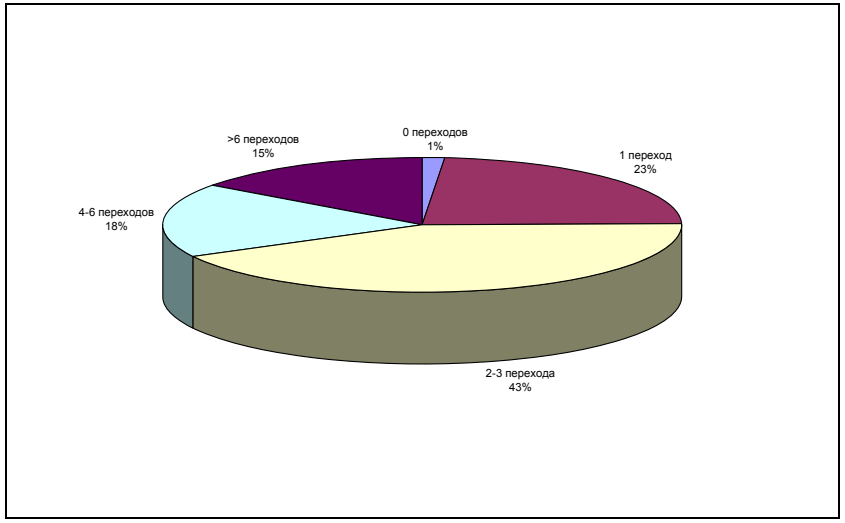

Рис. 3. Количество переходов из состояния

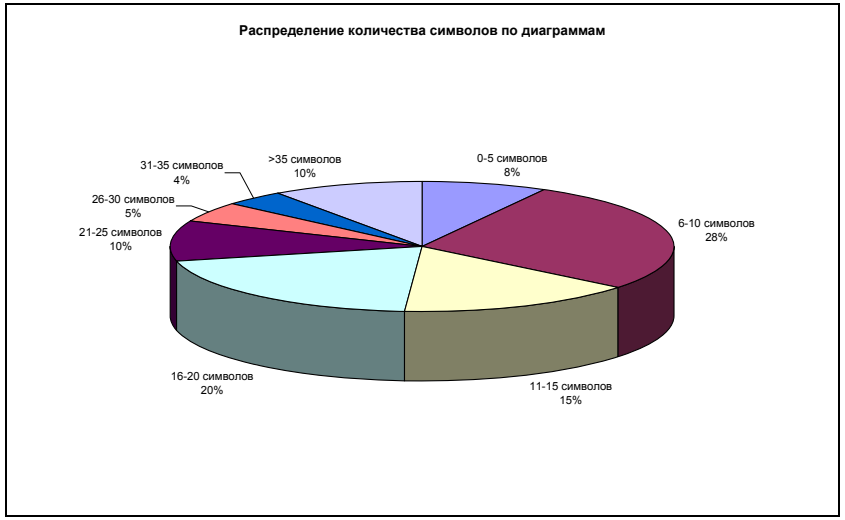

Рис. 4. Распределение количества символов на диаграммах

Анализ диаграмм состояний показал (см. Рис. 4), что, в среднем, автомат, реализующий класс, включает в себя 3-4 диаграммы, каждая из которых содержит около 9 символов и 9 линий, что не должно в значительной степени

препятствовать пониманию. В то же время для 10% автоматов, описывающих внутреннюю логику работы системы и содержащих более 6 состояний и переходов, количество диаграмм, на которых описан автомат, возрастает до пятидесяти, что очень сильно затрудняет понимание целостной картины работы системы.

2.2. Используемые конструкции

Для повышения уровня выразительности и упрощения описания сложных систем в состав средств описания конечных автоматов UML был включен ряд специальных конструкций. Их использование позволяет во многом упростить и сократить описание сложных автоматов, и поэтому одной из целей проведенного исследования было выявление характера использования подобных конструкций. Далее приведен обзор полученных результатов.

За счет использования операторов ветвления в действиях, выполняемых при срабатывании перехода в автомате, один и тот же переход может в различных условиях перевести автомат в различные состояния. Максимально возможное использование ветвления означало бы наличие в каждом состоянии не более чем одного перехода для любого сигнала. В этом случае выбор состояния, в которое перейдет автомат, происходил бы в процессе интерпретации действий, приписанных переходу. Результаты статистического исследования приведены на Рис. 5.

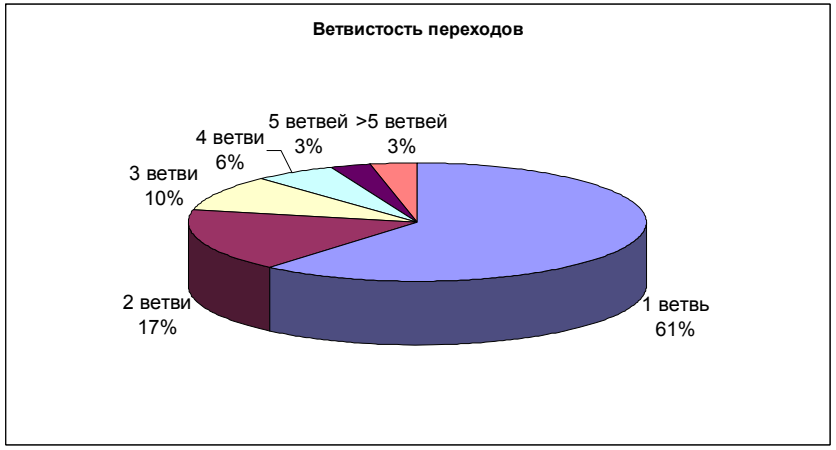

Рис. 5. Ветвистость переходов

Как и следовало ожидать, большинство переходов не разветвляются, а около 90% из них имеет не более трех ветвей. Однако 3% переходов, имеющие более 5 ветвей, могут заметно усложнить понимание логики работы системы. Абсолютный максимум составил 21 ветвь в одном переходе.

Использование графического синтаксиса позволяет проводить графическую декомпозицию диаграмм состояний – распределять сложные автоматы по нескольким графическим сущностям, не упрощая при этом структуру автомата. Этот подход позволяет облегчить процесс понимания деталей работы сложного автомата, однако затрудняет восприятие автомата как единого целого, что немаловажно для понимания логики работы сложной системы.

Одним из средств графической декомпозиции UML являются метки. Они позволяют графически отделить участки диаграммы состояний, чтобы, например, перенести их на другую диаграмму или расположить отдельно на исходной диаграмме. Кроме того, введение меток способствует повторному использованию фрагментов диаграмм, так как переход на единожды описанную метку может быть выполнен многократно из различных частей автомата. Статистика использования меток приведена на Рис. 6.

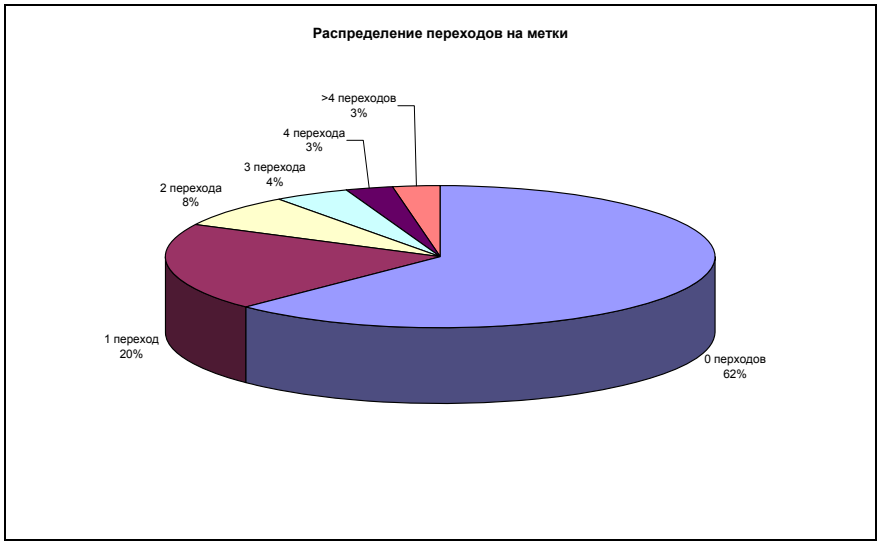

Рис. 6. Распределение переходов на метки

Распределение количества команд перехода на метки очень похоже на распределение количества ветвей. В обоих случаях наиболее простые варианты (одна ветвь и отсутствие переходов на метки) обеспечивают около 60% случаев, а следующие по сложности варианты (две ветви и одна команда перехода на метку) – около 20%, в то время как остальные варианты имеют по 3-4%. Однако в автоматах встречались и переходы, перегруженные командами перехода на метки. Для некоторых переходов в автомате максимальное количество команд перехода на метку превысило 20.

Чтобы избежать дублирования переходов для различных состояний, можно использовать несколько приемов. В UML в символе состояния можно

перечислить несколько имен состояний, и тогда все переходы, выходящие из этого символа, будут относиться ко всем перечисленным состояниям. Кроме того, если в качестве имени состояния указать символ «*», то переходы, выходящие из этого символа, будут относиться ко всем состояниям автомата. Также имеется возможность исключить определенные состояния из множества состояний, описываемого символом «*». Умелое использование этих возможностей позволяет значительно упростить описание переходов, применимых более чем к одному состоянию. Результаты статистического исследования показали, что символ * присутствует в 12% символов состояния, что свидетельствует о достаточно активном использовании этой подстановки и необходимости более детального изучения вариантов ее использования и возможных трансформаций с выделением или заменой символа «*».

Кроме того, при описании состояния, в которое должен быть совершен переход, UML позволяет использовать символ «-», означающий состояние, в котором был инициирован исходный переход. Согласно статистике более трети символов состояния содержит символ «-». Это снова свидетельствует об удобстве и востребованности этой конструкции, а также о необходимости исследовать затрагивающие ее трансформации.

2.3. Типичные способы построения конечных автоматов

Анализ полученной выборки не выявил каких-либо стандартов или «правил хорошего тона» при разработке конечных автоматов. Единственным «паттерном» можно считать применяемую одной из компаний методику, когда при описании автомата для каждого перехода из заданного состояния используется отдельная диаграмма, и еще одна диаграмма используется для всех общих описаний. Естественным недостатком такого подхода является сложность получения целостного представления о моделируемом автомате по причине разрозненности отдельных диаграмм, описывающих состояния.

3. Улучшение структуры конечных автоматов UML

3.1. Трансформация «выделение метода» для конечных автоматов UML

Идея трансформации “Extract method” состоит в создании нового метода и переносе части исходного автомата в добавленный метод. Данная трансформация во многом аналогична известному рефакторингу «Extract Method» для объектно-ориентированных языков программирования, описанному в каталоге Фаулера [1]. Суть традиционной трансформации состоит в выделении участка кода и перемещении его в другой метод. Это позволяет сделать код исходного метода более понятным и повышает вероятность повторного использования выделенного метода.

Для корректного выполнения традиционного рефакторинга “Extract method” требуется тщательный анализ потока данных в выделяемом участке кода, так

как все используемые переменные должны быть переданы в метод в качестве параметров, а все изменения переменных должны быть тем или иным образом возвращены исходному методу, если измененные переменные используются в нем далее.

Для первичного рассмотрения проблемы выделения метода в автомате эту проблему можно обойти следующим образом. Если используемая переменная является атрибутом автомата или сущности, содержащей автомат, то она будет видна и в выделенном методе и, следовательно, ее не нужно передавать в качестве параметра. Если же используемая переменная является локальной для действий, выполняемых в переходе, то при перенесении всех действий перехода в выделяемый метод определение локальной переменной и все ее использования будут также перенесены. Для выделения метода, в который помещаются не все действия, выполняемые в переходе, требуется дополнительный анализ потока данных.

Следует подчеркнуть исключительную важность автоматизированной поддержки рефакторинга при проведении подобных преобразований, ибо сложность проводимого анализа будет способствовать ошибкам.

Идея, лежащая в основе традиционного рефакторинга “Extract method”, может быть применена к конечным автоматам несколькими способами.

- Для конечных автоматов UML можно применить традиционную трансформацию «выделение метода», которая состоит из выделения подпоследовательности действий одного из переходов конечного автомата в метод.
- В рамках описываемого исследования был разработан новый вариант трансформации «выделение метода», специфичный только для конечных автоматов UML, – «выделение в метод части конечного автомата», который подразумевает перенос в выделяемый метод не только действий, связанных с переходом, но и самих переходов и состояний.

3.2. Выделение в метод части конечного автомата

Рассмотрим определение части конечного автомата, представленное на Рис. 7.

Выбрав часть перехода вместе со следующим состоянием, можно выделить метод, в который войдет часть состояний конечного автомата, начиная с состояния Y. Будем называть такую трансформацию Extract Sub State Machine.

Применимость данной трансформации связана со следующим свойством. Состояния, переносимые в выделяемый метод, перестают принадлежать исходному автомату и, следовательно, команды перехода, приводящие из состояний исходного автомата в состояния, перенесенные в выделенный автомат, некорректны. Такие команды перехода (смены состояния) должны быть заменены командами вызова выделяемого метода. Однако у автомата, реализующего метод, может быть только одна входная точка, поэтому либо все

такие команды должны осуществлять переход в одно и то же состояние, либо можно использовать целочисленный параметр для передачи номера того состояния, с которого должно начаться выполнение метода. Но введение такого параметра и добавление его обработки в начальном переходе усложняет выделяемый автомат и затрудняет его понимание.

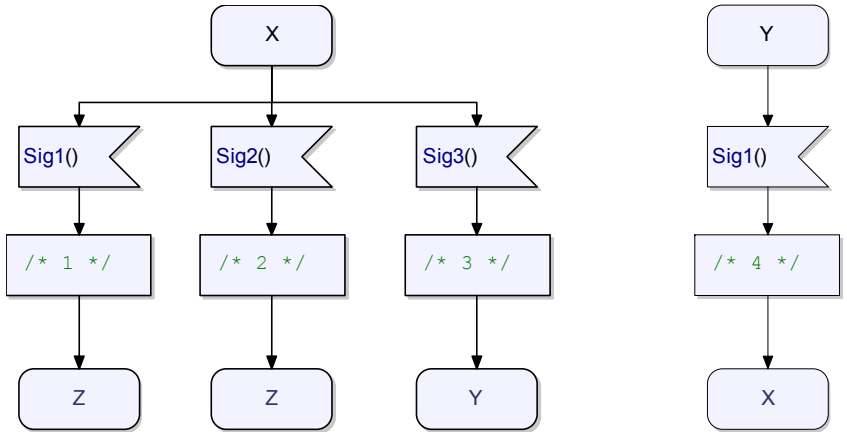

Рис. 7. Часть автомата, допускающая выделение метода

Для обработки обратных переходов из состояний выделенного метода в состояния исходного автомата может быть применен следующий прием. Все состояния исходного метода, в которые можно попасть из выделяемого метода, нумеруются последовательными натуральными числами. Все команды перехода, ведущие из состояний выделяемого метода в состояния исходного, заменяются командами возврата из метода, использующими в качестве возвращаемого значения номер того состояния, в которое должен был бы осуществиться переход. После замены тела выделенного метода его вызовом возвращаемое им значение присваивается новой локальной переменной, и после возврата из метода оно анализируется для определения состояния, в которое должен был осуществиться переход. Описанный прием, хотя и позволяет при выделении метода не накладывать ограничений на количество обратных переходов, на практике зачастую только затрудняет понимание автомата, что противоречит целям проведения рефакторинга.

Таким образом, несмотря на то, что количество вариантов применения предлагаемой трансформации достаточно велико, далеко не все из них служат цели упрощения понимания автомата. Тем не менее, можно выделить несколько специальных случаев, когда все прямые и все обратные переходы ведут в одно состояние.

Введём несколько обозначений. Обозначим через `RS(x, y)` множество, содержащее все состояния автомата, в которые можно попасть из состояния y,

не проходя при этом через состояние x, включая y и исключая x. Специальный символ stop добавляется в множество RS(x, y), если из состояния y за некоторое количество переходов можно дойти до действия, завершающего работу автомата (stop). Обозначим множество всех переходов некоторого конечного автомата A через All_T(A), а переход из состояния a в состояние b по сигналу z – через t(a, z->b).

Определение 1. Множество состояний S ***замкнуто на множестве переходов T****, если не существует перехода*

$$t(x', e->s) \in T : s \in S, x' \notin S$$

Трансформация Extract Sub State Machine для перехода t(x,e -> y) конечного автомата A, может быть применена при выполнении следующих трех условий:

1) x != y, иначе RS пусто и это будет случай выделения автомата без состояний;
2) множество RS(x,y) замкнуто на множестве переходов All_T(A)\t(x,e->y);
3) stop ∉ RS(x, y).

Трансформация Extract Sub State Machine для перехода t(x,e -> y) заключается в следующем.

1) Создаётся и добавляется в активный класс метод P с реализацией в виде конечного автомата.
2) В этот метод перемещаются все состояния из множества RS(x, y) .
3) Действия, приписанные переходу t(x,e -> y), становятся действиями, приписанными начальному переходу конечного автомата метода P(). Вместо них в исходный конечный автомат вставляется вызов метода P() и команда перехода в исходное состояние x.
4) Все команды перехода в состояние x в созданном конечном автомате заменяются на команды возврата из метода (return).

Часть автомата, выделенная в метод, обладает следующей семантикой: получив сигнал Sig3(), автомат выполняет некоторые действия, начиная с состояния y, по завершении которых возвращается в состояние x. Подобная логика близка по смыслу к вызову метода: выполнение задачи с последующим возвратом в исходное состояние. Именно это и служит основанием для выделения метода.

В результате преобразования выделяется структурная единица автомата – метод, а диаграмма, описывающая конечный автомат, уменьшается, что

упрощает его понимание Результат преобразования представлен на Рис. 8. Выделенный метод показан на Рис. 9. Выделенный метод можно использовать повторно для уменьшения дублирования кода.

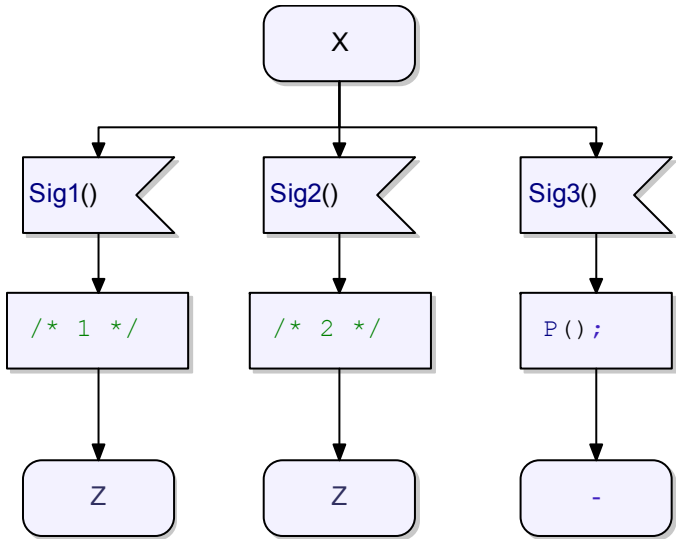

Рис. 8. Часть автомата после проведения преобразования Extract Method

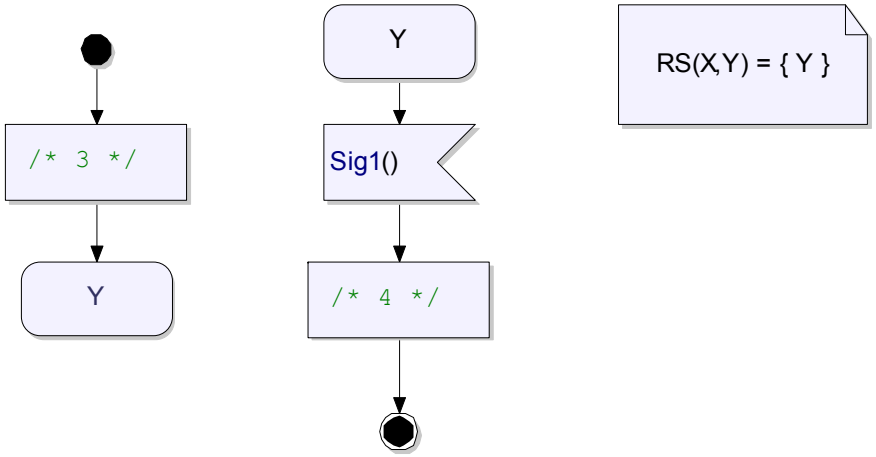

Рис. 9. Описание выделенного метода

Существует несколько частных случаев трансформации Extract Sub State Machine.

1) Ни для одного состояния из `RS(x, y)` нет перехода в `x`. Это значит, что возврат из созданного метода невозможен, в конечном автомате найден бесконечный цикл; возможно, это «серверная составляющая» исходного автомата.
2) Множество `RS(x,y)` содержит символ `stop`, и ни для одного состояния из `RS(x, y)` нет перехода в `x`. Это означает, что выделенная в метод часть автомата рано или могла завершить его работу: либо выделенные действия реализуют необходимую подготовку к завершению работы автомата (аналог деструктора в объектно-ориентированном программировании), либо найдена «серверная составляющая» исходного автомата (только если есть цикл).

Во втором случае выделение метода корректно при выполнении следующих условий:

a) в выделенном методе все команды завершения работы автомата (`stop`) должны быть заменены командами возврата из метода `(return);`
b) вместо действий, приписанных исходному переходу, должен быть добавлен вызов метода `P()` и команда завершения работы автомата (`stop`).

В рассматриваемом случае преобразованный автомат будет выглядеть так, как показано на Рис. 10.

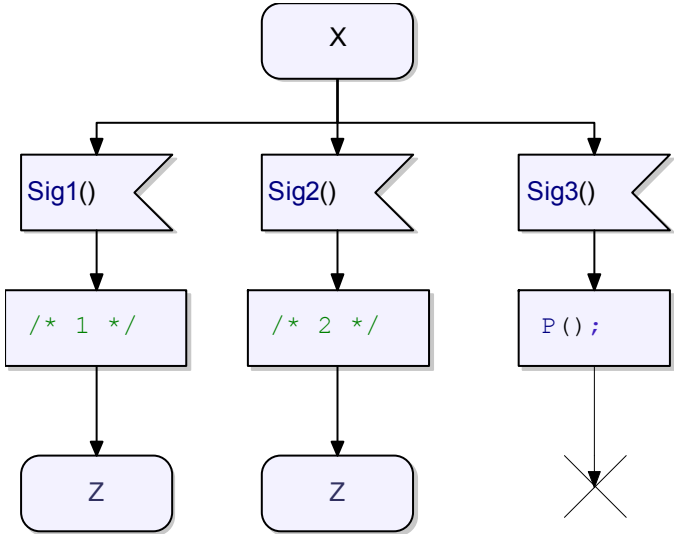

Рис. 10. Результаты применения второго варианта трансформации

3.3. Пример «Мобильный телефон»

Продемонстрируем применение трансформации «Выделение части конечного автомата в метод» на одном из конечных автоматов системы Mobile, моделирующей работу мобильного телефона.

В исходной системе конечный автомат представлен на 28 диаграммах, каждая из которых описывает ровно один переход (Рис. 11).

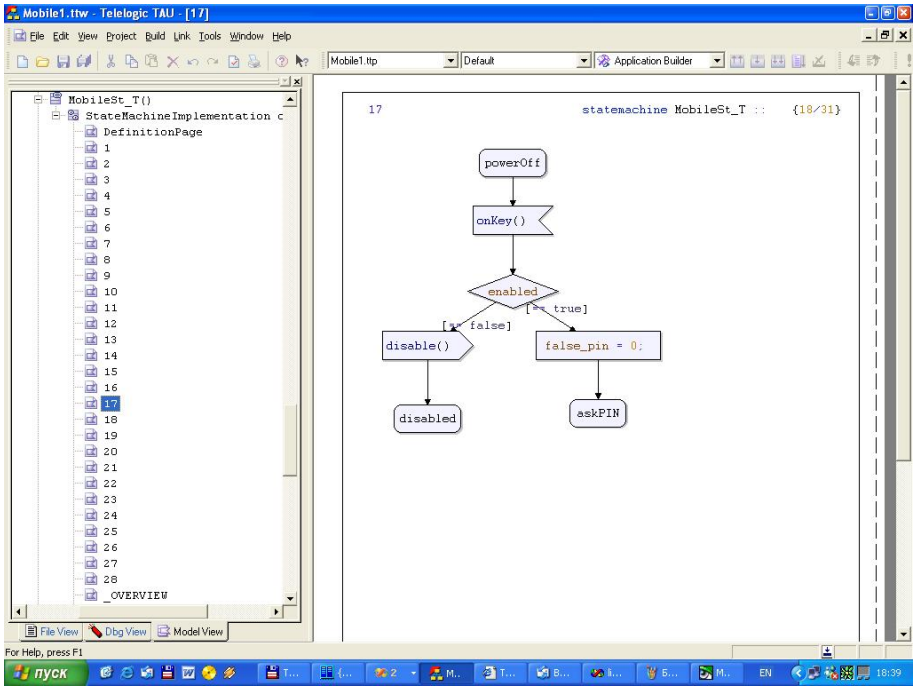

Рис. 11. Исходный вид конечного автомата

Такое представление не позволяет понять цельную структуру конечного автомата. Для упрощения понимания была создана дополнительная диаграмма, схематично описывающая весь конечный автомат, иллюстрирующая все состояния и переходы со всеми ветвлениями (Рис. 12).

Приведённый алгоритм позволяет найти и выделить из данного конечного автомата три метода. На первом шаге в метод Initialize() выделяются четыре последовательных состояния (Рис. 13).

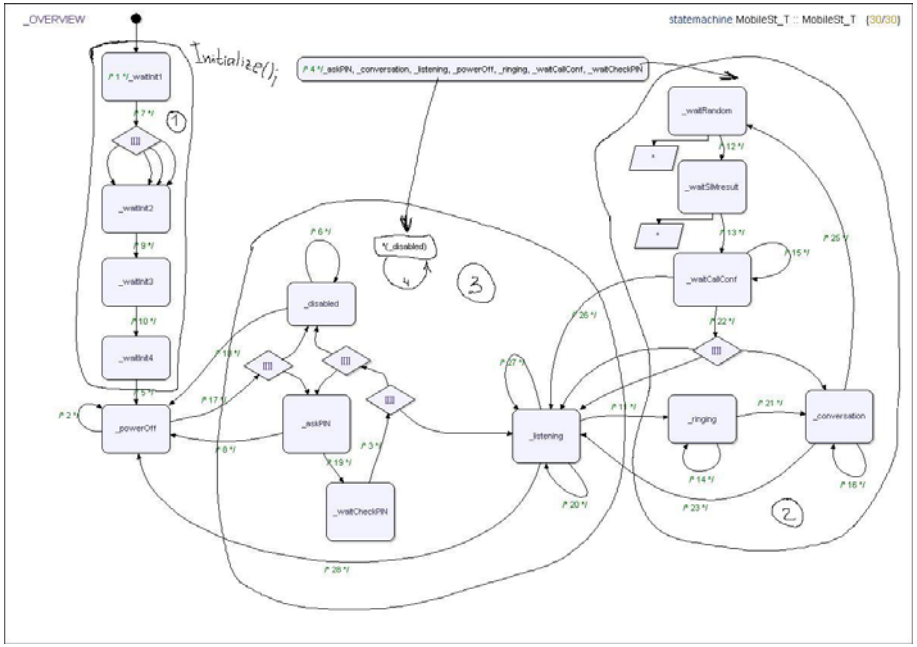

Рис. 12. Краткое описание всего конечного автомата

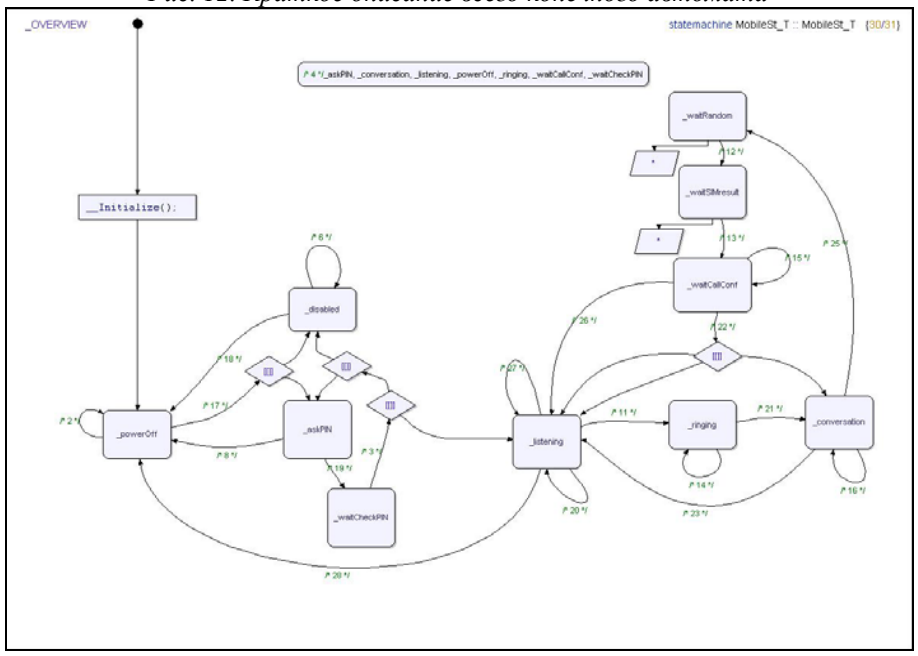

Рис. 13. Выделение метода Initialize

На втором шаге выделяется метод TalkingThePhone() (Рис. 14), после чего становится возможным выделить ещё один метод, который назовём Working().

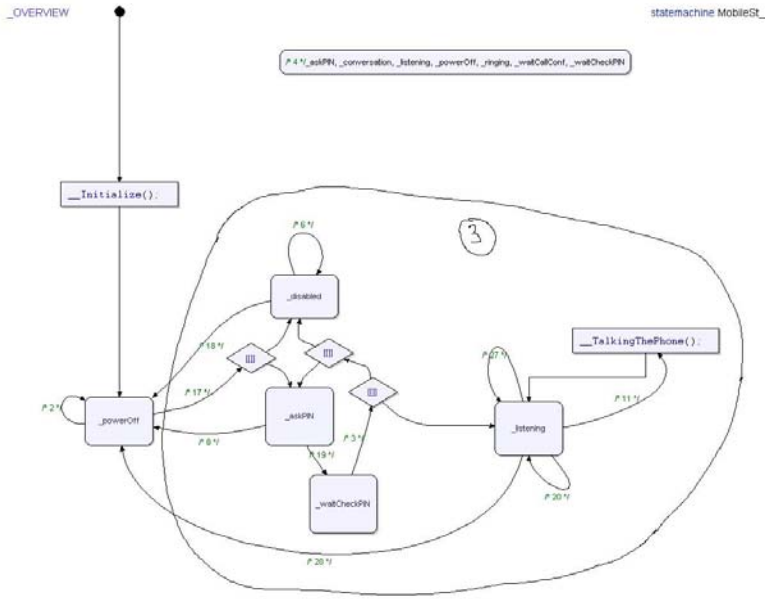

Рис. 14. Выделение метода TalkingThePhone

Обратим внимание на то, что выделение метода Working возможно только после выделения метода TalkingThePhone. Процесс применения трансформации итеративный. Поиск частей конечного автомата, которые можно вынести в отдельный метод, можно автоматизировать.

В результате применения трансформаций исходный конечный автомат сильно упростился и свободно помещается на одной диаграмме (Рис. 15). Теперь он содержит только одно состояние (вместо четырнадцати состояний в исходном автомате) и вызов двух методов.

Выделены три метода Initialize(), TalkingThePhone() (Рис. 16) и Working() (Рис. 17), содержащие 4, 5 и 4 состояния соответственно.

4. Заключение.

Таким образом, задача трансформации моделей UML является достаточно актуальной. Проведенные исследования подтвердили гипотезу о возможности улучшения структурных качеств и упрощения понимания моделей, применяемых в реальных промышленных проектах. Это ставит перед исследователями задачи поиска новых трансформаций, в которых учитывается

специфику моделей UML. Для оценки применимости и полезности трансформаций необходимо продолжение работы по формализации подмножества конечных автоматов UML, описанию семантики их выполнения, а также создание инструментальных средств, автоматизирующих сбор необходимой информации и процесс трансформации моделей.

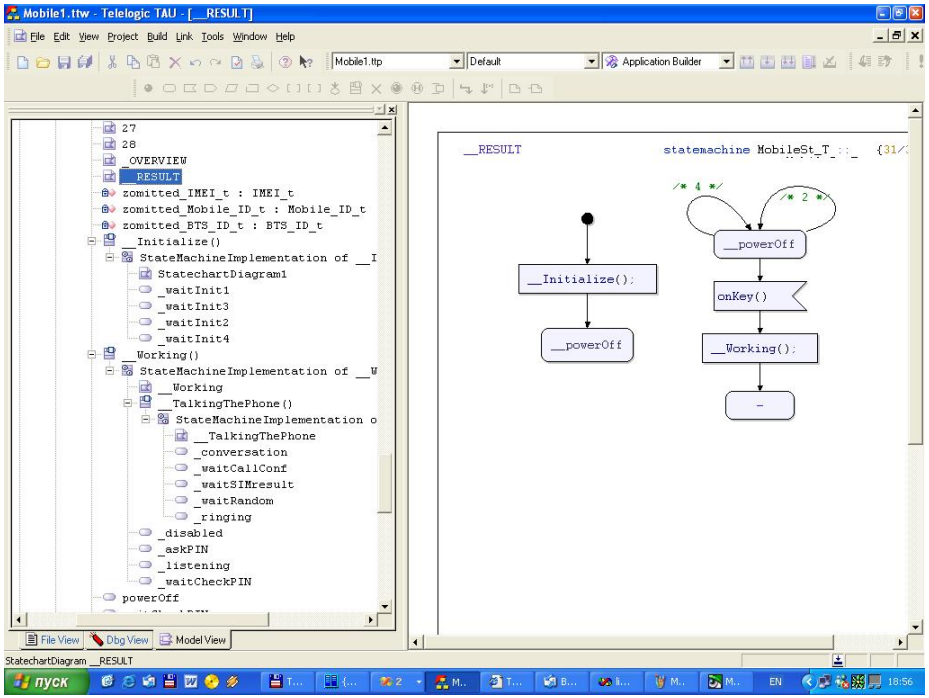

Рис. 15. Результат трансформации

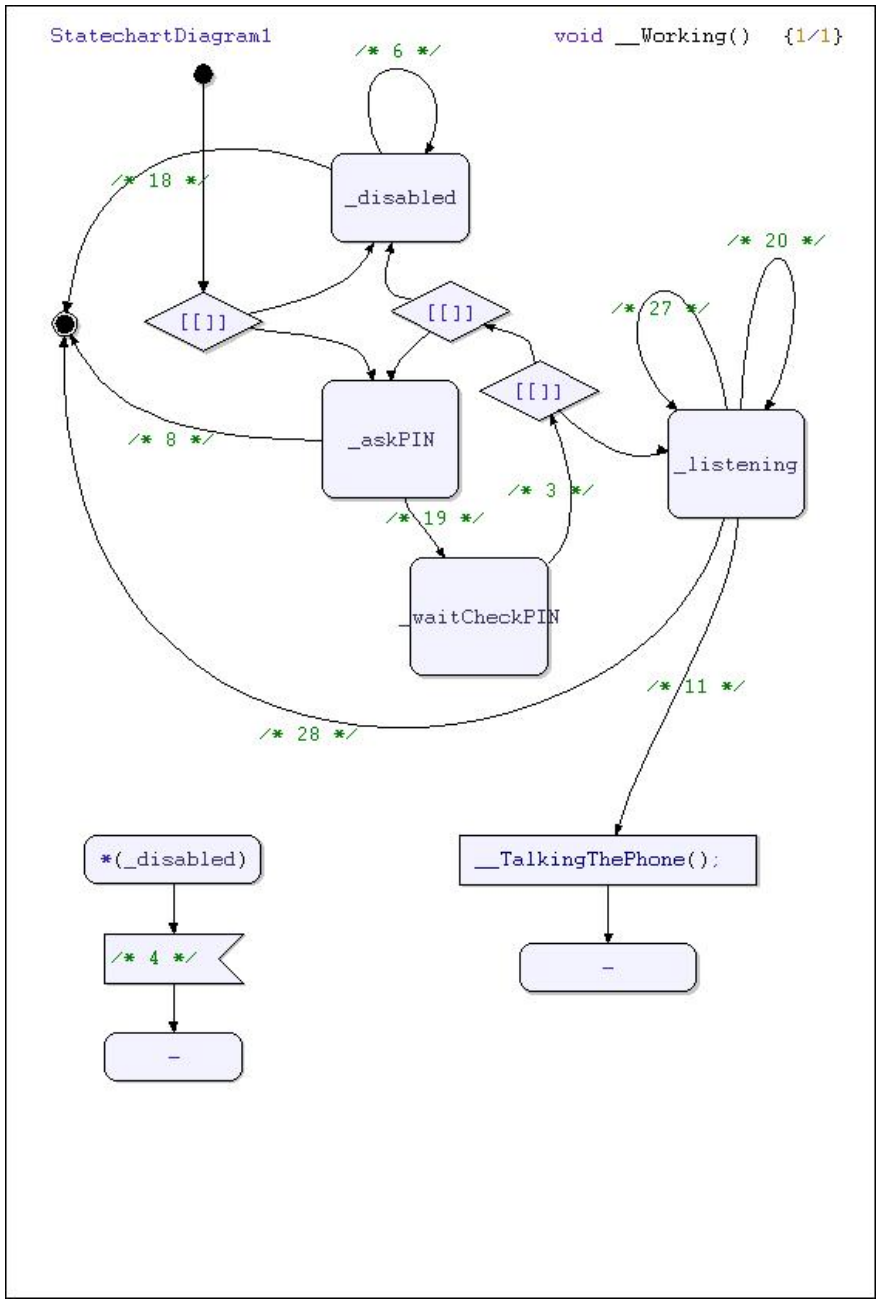

Рис. 16. Метод Working

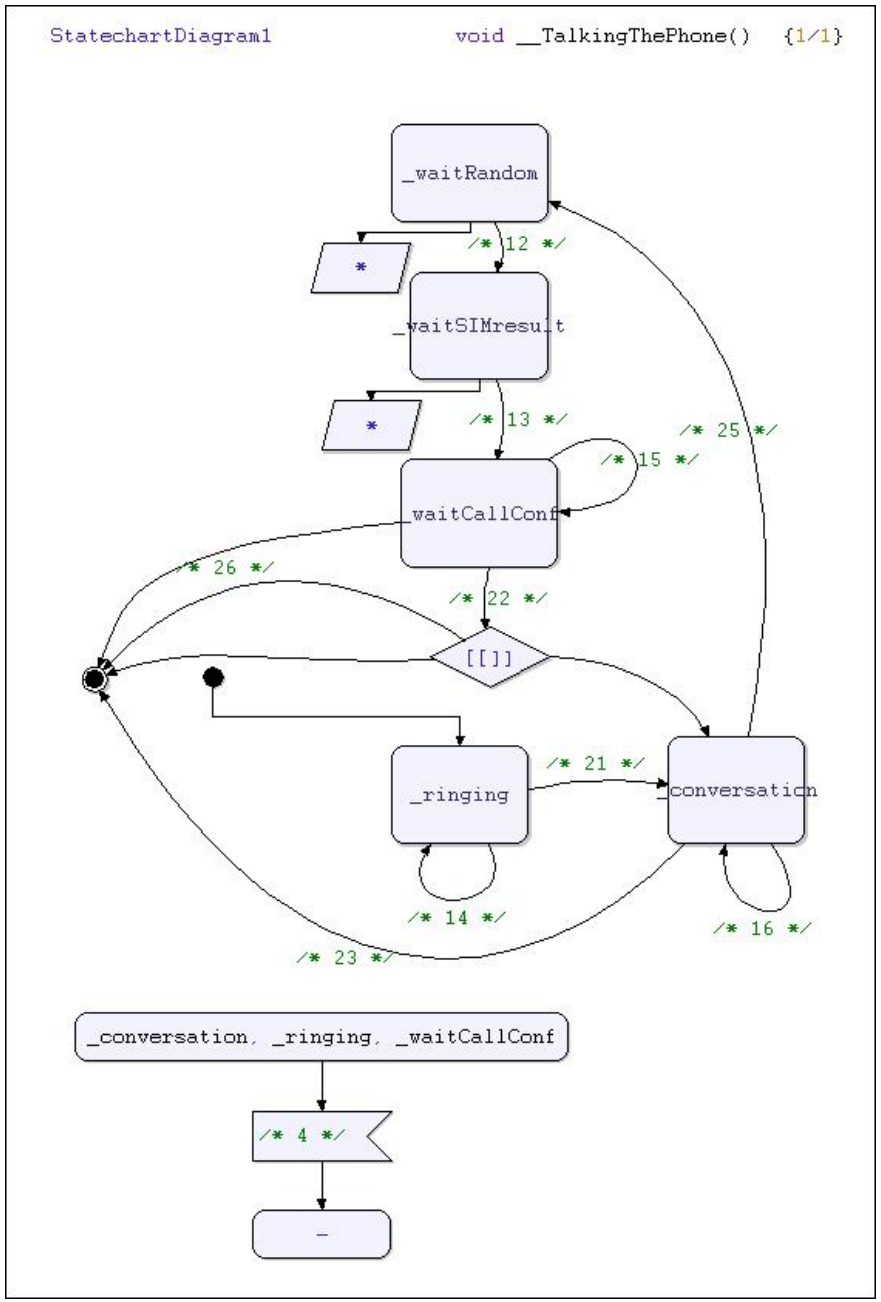

Рис. 17. Метод TalkingThePhone

Литература

1. Фаулер М., Бек К., Брант Д., Робертс Д., Апдайк У. Рефакторинг: улучшение существующего кода. - СПб.: Символ-Плюс, 2002. - 432 с.
2. William F. Opdyke, "Refactoring Object-Oriented Frameworks". PhD Thesis, University of Illinois at Urbana-Champaign. Also available as Technical Report UIUCDCS-R-92-1759, Department of Computer Science, University of Illinois at Urbana-Champaign.
3. Tom Mens. A Survey of Software Refactoring, IEEE Transactions on Software Engineering, Vol. 30, No. 2, February 2004.
4. Van Gorp, P.; Stenten, H.; Mens, T. and Demeyer, S. Towards Automating Source Consistent UML Refactorings, in Proc. Unified Modeling Language Conf. 2003, 2003.
5. Astels. D., 'Refactoring with UML', in Marchesi, M and Succi, G (eds). XP 2002 - Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Proceses in Software Engineering, 2002.
6. Tom Mens, Niels Van Eetvelde, Dirk Janssens, and Serge Demeyer. Formalising refactorings with graph transformations. Fundamenta Informaticae, 2003.
7. Robert France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song, “A UML-Based Pattern Specification echnique,” IEEE Transactions on Software Engineering, Vol.30, No.3, pp. 193-206, March 2004.
8. Marciniak J. J. The Encyclopedia of Software Engineering // Wiley Publishers. 2002. – 2076p.: il.
9. Буч Г., Рамбо Д., Джекобсон А. UML Руководство пользователя // М.: ДМК Пресс. 2001. – 432 с.: ил.
10. Меллор С., Кларк Э., Футагами Т. Разработка на базе моделей // Сайт журнала «Открытые Системы»: URL: http://www.osp.ru/os/2003/12/030.htm (2005. 25 июня).
11. Селич Б., Практические аспекты разработки на базе моделей // Сайт журнала «Открытые системы»: URL: http://www.osp.ru/os/2003/12/033.htm (2005. 25 июня).
12. Zs. Pap, I. Majzij, A. Pataricza, A. Szegi. Completeness and Consistency Analysis of UML Statechart Specifications // I.Maizik homepage: URL: http://home.mit.bme.hu/~majzik/publicat/ddecs2001.pdf (2005. 25 июня).
13. Unified Modeling Language: Superstructure // OMG official website: URL: http://www.omg.org/docs/formal/05-07-04.pdf (2005. 25 июня).
14. UML 2.0 OCL Specification // OMG official website: URL: http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf (2005. 25 июня).
15. H. Eriksson, M. Penker, B. Lyons, D. Fado. UML2 Toolkit // Indiapolis: Wiley Publishing Inc. 2004. – 511 p. il.