Методы композиции и декомпозиции исполняемых UML моделей

Е. Д. Волкова, А. Д. Страбыкин

Аннотация. В статье[1] рассматриваются способы трансформации исполняемых UML-моделей, которые основаны на композиции или декомпозиции отдельных частей модели и позволяют облегчить понимание модели и упростить работу с ней; предлагаются и обсуждаются новые виды таких трансформаций, иллюстрируется применение преобразований к различным элементам UML-моделей.

1. Введение

Известны три интеллектуальные возможности человека, используемые при разработке программных средств: способность к перебору, способность к абстракции и способность к математической индукции. Однако способность человека к перебору, необходимая при создании и модификации сложных систем, в том числе и программных комплексов, весьма ограничена — в среднем человек может уверенно, не сбиваясь, перебирать в пределах 1000 элементов. Средством преодоления этой ограниченности является способность человека к абстракции, благодаря которой человек может объединять разные предметы или экземпляры в одно понятие, заменять множество элементов одним элементом другого рода, упрощая, тем самым, восприятие. Абстракция лежит в основе моделирования, которое в последнее время получило широчайшее распространение и активно используется в процессе создания программных систем.

Наиболее популярным языком моделирования в области разработки программного обеспечения на данный момент является графический язык UML (Unified Modeling Languag — унифицированный язык моделировании) — язык, предназначенный для визуализации, специфицирования, конструирования и документирования программных систем. Выразительных средств этого языка в совокупности с мощными механизмами расширения достаточно для того, чтобы описать любую программную систему, со всех точек зрения, актуальных на различных этапах жизненного цикла. Более того, все шире становится область применения концепций модельно-

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

ориентированной разработки (MDD — Model Drive Development), отводящих моделям главную роль в процессе создания и поддержки системы.

В рамках таких подходов собственно разработка сводится к последовательному уточнению модели системы, начинающемуся с абстрактной модели и, заканчивающемуся готовой программной системой. Таким образом, при использовании MDD сложность создаваемых моделей будет напрямую отражать сложность системы, а работа с моделью может создавать такие же трудности, как те, которые возникают при чтении и понимании большого количества исходного кода на традиционных языках программирования.

Перечисленные выше факторы обуславливают потребность в методах, позволяющих упрощать восприятие сложных моделей и облегчать работу с ними. В то же время применение таких методов не должно затрагивать важных для пользователей и разработчиков модели свойств системы, например поведенческих, которые в UML задаются при помощи диаграмм конечных автоматов (*statemachine diagrams*), диаграмм последовательностей (*sequence diagrams*) и диаграмм деятельности (*activity diagrams*). Данное свойство сближает такие методы трансформации моделей с известными приемами рефакторинга объектно-ориентированного программного обеспечения. Однако графическая природа языка UML накладывает свою отпечаток на возникающие проблемы и методы их решения. Кроме того, язык UML продолжает развиваться, и последняя на сегодняшний день версия стандарта UML 2.0 включает в себя множество новых элементов, многие из которых могут быть использованы для упрощения восприятия моделей.

2. Конечные автоматы UML

Конечные автоматы UML могут описывать поведение следующих элементов исполняемых моделей: активный класс (*active class*), операция (*operation*), составное состояние (*composite state*). В [19] было проведено количественное исследование моделей, используемых в реальных коммерческих проектах. Была составлена представительная выборка таких моделей, анализ которой показал, что, несмотря на наличие достаточно легко понимаемой структуры у почти 90% исследованных элементов модели (диаграмм, автоматов), оставшиеся 10% элементов, как правило, весьма сложны и, более того, именно они специфицируют основную логику работы системы. Таким образом, любые попытки понять и проанализировать поведенческие стороны сложной системы будут наталкиваться на необходимость изучения сложных конечных автоматов, описанных посредством громоздких диаграмм состояний. А так как для любой относительно нетривиальной модификации, количество которых за время жизни программной системы может исчисляться сотнями тысяч, требуется понимание логики работы системы, возникает потребность в методах, упрощающих восприятие системы и облегчающих ее понимание.

Суть предлагаемого подхода состоит в том, что исходная модель сложной системы посредством описываемых преобразований трансформируется и дополняется элементами, образуя иерархическое представление, удобное как в случаях, когда нужно получить представление об общих принципах работы системы, так и в случаях, когда необходим детальный анализ конкретных аспектов. Такие преобразования могут быть классифицированы как методы композиции и декомпозиции автоматных моделей.

Под композицией в данной работе понимается такое преобразование модели, результатом которого является создание единой новой целостной сущности из нескольких имеющихся сущностей. Например, объединение в одной диаграмме элементов всех диаграмм, описывающих конкретный автомат, есть примитивное преобразование-композиция. Следует отметить, что такая трансформация является примером графической композиции, так как затрагивает только элементы, описывающие графические сущности модели. Как правило, применение композиции способствует получению целостного представления о каком-либо элементе системы; при этом должны быть приняты меры, предотвращающие чрезмерное усложнение создаваемого элемента.

Под декомпозицией следует понимать преобразование, обратное композиции, то есть такое преобразование, результатом которого является создание нескольких новых сущностей на основе единственной исходной сущности. Для приведенного примера графической композиции обратным будет преобразование, в результате которого элементы одной диаграммы распределяются между несколькими другими диаграммами.

Применяя композицию и декомпозицию, можно построить иерархическое описание сложных элементов системы, облегчающее ее восприятие. При этом понимание работы системы будет происходить сверху вниз по иерархии описания. Сначала пользователь знакомится с основными компонентами описываемой части модели и принципами их взаимодействия, и далее по необходимости углубляется, переходя к более детальному описанию интересующего его компонента. На практике даже для достаточно сложных конечных автоматов редко требуется больше 4-5 уровней детализации.

Помимо упомянутого объединения и разбиения на диаграммы, другие способы композиции-декомпозиции автоматов могут быть основаны на использовании такого элемента языка UML, как метки. Их использование позволяет 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. Однако, так же, как и при написании программ на традиционных языках программирования, использование меток при спецификации автоматов заметно усложняет понимание принципов работы автомата. Особенно затруднительным становится анализ отдельного перехода в автомате,

например, с целью выяснить, в какие состояния автомат может перейти при срабатывании этого перехода, если в последовательности действий этого перехода встречаются команды перехода на метки, а описания этих меток распределены по многочисленным диаграммам. Поэтому, по нашему мнению, использование меток при описании конечных автоматов оправдано лишь в тех исключительно редких случаях, когда действия, выполняемые при срабатывании перехода, настолько сложны, что требуются отдельные диаграммы даже для описания их частей.

3. Методы композиции и декомпозиции для конечных автоматов UML

В результате проведённого исследования разработано несколько методов композиции (декомпозиции) конечных автоматов UML. Предложенные методы базируются на двух основных идеях: объединение части состояний конечного автомата в структурную единицу конечного автомата — метод, и формирование иерархической единицы конечного автомата — составного состояния.

3.1. Трансформация «Выделение метода» для конечных автоматов UML

Идея трансформации «Выделение метода» состоит в создании нового метода и переносе части исходного автомата в добавленный метод. Данная трансформация во многом аналогична известному рефакторингу «Выделение метода» для объектно-ориентированных языков программирования, описанному в каталоге Фаулера [6]. 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 «Выделение метода» 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 “Выделение метода”, может быть применена к конечным автоматам несколькими способами.

- Для конечных автоматов UML можно применить традиционную трансформацию «выделение метода», которая состоит из выделения в метод подпоследовательности действий одного из переходов конечного автомата.
- 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 возвратной части конечного автомата

Назовём *возвратной частью конечного автомата для состояния X* группу состояний конечного автомата, в которую есть только один вход из некоторого состояния *X*, и все переходы, выходящие за пределы группы, попадают обратно в *X*. Между состояниями, входящими в группу, возможны любые переходы, но существует ровно одно состояние, из которого выполняется переход в группу, и куда попадают все переходы, ведущие за пределы выбранной группы состояний.

Согласно исследованию, проведённому в [19], многие конечные автоматы, используемые в промышленных системах, содержат в себе несколько возвратных частей для различных состояний. Многие из них состоят из 1–3 состояний, однако в каждом третьем конечном автомате с большим количеством состояний встречаются возвратные части из 4 и более состояний.

Большое количество состояний затрудняет восприятие общей схемы работы конечного автомата. Зачастую, такой конечный автомат приходится описывать на нескольких диаграммах, что нарушает целостность картины. Однако все состояния, входящие в возвратную часть конечного автомата, можно перенести в отдельный метод, что сократит общее количество состояний автомата. Назовём такую трансформацию «выделением в метод возвратной части конечного автомата».

Рассмотрим определение части модельного конечного автомата, представленное на Рис. 1. Это автомат допускает выделение метода, так как

для состояния *X* существует возвратная часть конечного автомата, состоящая из одного состояния *Y*.

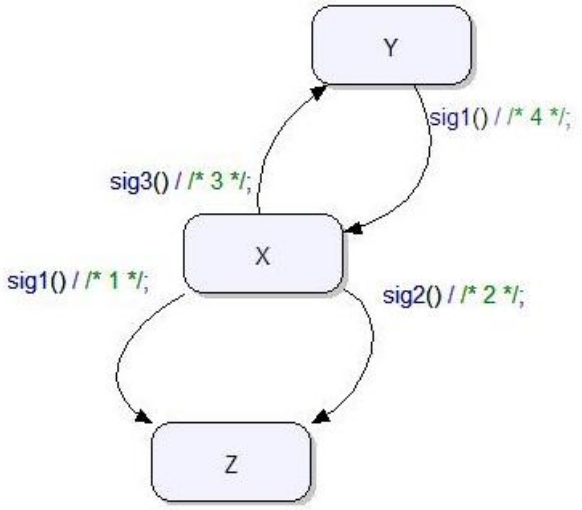

Рис. 1. Часть автомата, допускающая выделение метода

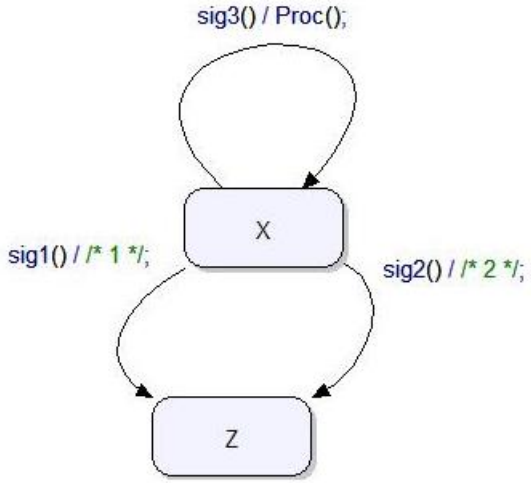

Рис. 2. Часть автомата после проведения преобразования Extract Method

В результате применения трансформации «выделение метода», создаётся метод *Proc* () с реализацией в виде конечного автомата. Все состояния из возвратной части конечного автомата (в данном примере только *Y*) переносятся в выделенный метод. Действия, приписанные переходу в *Y*, становятся действиями, приписанными начальному переходу конечного автомата метода *Proc* (). Вместо них в исходный конечный автомат вставляется вызов метода *Proc* () и команда перехода в исходное состояние *X* (Рис. 2).

Все команды перехода в состояние *X* в созданном конечном автомате заменяются командами возврата из метода (*return*) (Рис. 3).

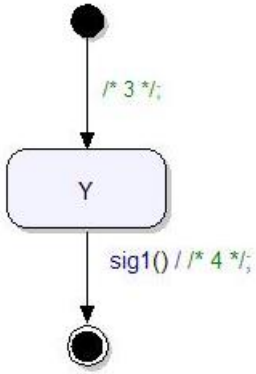

Рис. 3. Описание выделенного метода Proc ()

Часть автомата, выделенная в метод, обладает следующей семантикой: получив сигнал *sig*3 (), 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. Выделенный метод можно использовать повторно для уменьшения дублирования кода.

Существует частный случай трансформации — выделение в метод группы состояний автомата, все переходы из которых завершают работу автомата, то есть ведут в символ `stop`. В этом случае выделение метода корректно, если все команды завершения работы автомата в выделенном методе (`stop`) заменяются командами возврата из метода `(return)`, а вместо действий,

приписанных исходному переходу, добавляется вызов метода *Proc* () и команда завершения работы автомата (`stop`).

В таком случае преобразованный автомат будет выглядеть так, как показано на Рис. 4:

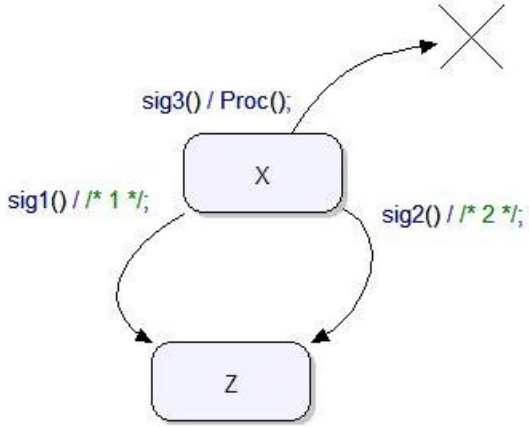

Рис. 4. Результаты применения второго варианта трансформации

3.3. Выделение в метод последовательной части конечного автомата

Во многих конечных автоматах встречаются последовательные части — несколько состояний, вытянутых в цепочку, с одним входом и одним выходом для каждого состояния (Рис. 5). Согласно исследованию, проведённому в [19], 25 % рассмотренных крупных автоматов содержат последовательные цепочки длинной 3 и более состояний.

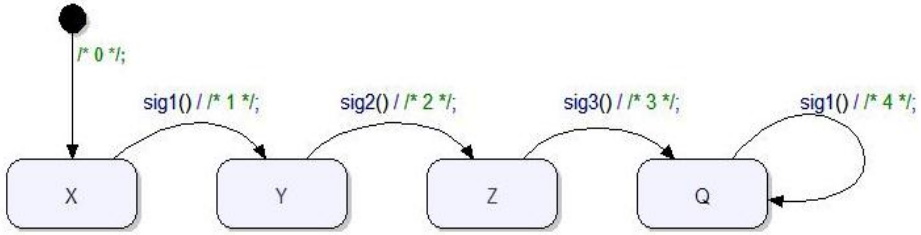

Рис. 5. Исходный автомат

В таких ситуациях все состояния, входящие в цепочку, можно перенести в метод. Это позволит упростить общий вид конечного автомата, уменьшить количество состояний и переходов, и скрыть детали поведения автомата при последовательном прохождении нескольких состояний. Назовём такую трансформацию «выделением в метод последовательной части конечного автомата».

В результате применения трансформации «Выделение метода», создаётся метод *Proc* () с реализацией в виде конечного автомата (Рис. 6).

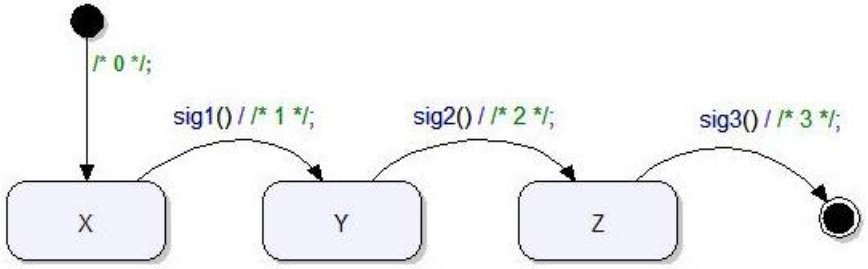

Рис. 6. Описание выделенного метода Proc ()

Все состояния из последовательной цепочки состояний (в данном примере *X*, *Y* и *Z*) переносятся в выделенный метод. Последний переход цепочки состояний, перенесённый в метод, завершается возвратом из метода (`return`). Действия, приписанные началу цепочки переходов, становятся действиями, приписанными начальному переходу конечного автомата метода *Proc* (). Вместо них в исходный конечный автомат вставляется вызов метода *Proc* () и команда перехода в состояние, следующее за цепочкой (*Q*) (Рис. 7)

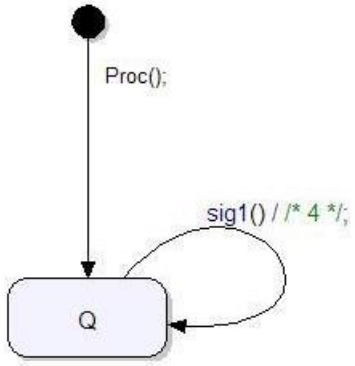

Рис. 7. Результат трансформации

Последовательные цепочки состояний, используемые в конечных автоматах, являются неким многошаговым переходом, определяющим последовательность действий и состояний, которые необходимо пройти, чтобы попасть в очередную часть автомата. Таким образом, все состояния и переходы, формирующие цепочку, являются логически связанными и допускают группировку в отдельную структурную единицу автомата — метод.

3.4. Необходимые условия для выделения части конечного автомата в метод

Автоматизация проведения преобразований для крупных конечных автоматов является исключительно важной задачей, ибо сложность проводимого анализа способствует возникновению ошибок. Одним из этапов процесса автоматизации является возможность автоматического поиска частей конечного автомата, которые можно поместить в отдельный метод согласно описанным выше подходам. После выделения одного метода могут появляться новые группы состояний, удовлетворяющие условиям выделения метода; поэтому необходимо иметь возможность автоматически выполнять итеративный поиск с возвращением для выявления наиболее оптимальной последовательности трансформаций.

Всё выше сказанное подтверждает актуальность и востребованность следующей задачи: сформулировать достаточно формальные необходимые условия для выделения части конечного автомата в метод, общие для всех предложенных трансформаций. Ниже приводятся такие условия.

Обозначим переход из состояния `a` в состояние `b` по событию `sig`, выполняющий действия `act`, через `t (a->b; sig; act)`. Обозначим конечное состояние автомата через `stop`, ветвление по условию `cond` через `decision (cond)`, а конечное состояние процедуры (возврат из процедуры) через `return`. Тогда, например, переход из ветвления по условию `cond` в конечное состояние автомата будет обозначаться как `t ( decision (cond)->stop; sig; act)`.

На диаграмме конечного автомата, записанной в нотации UML, кроме стрелок, обозначающих переходы, могут присутствовать следующие элементы: состояния автомата, символы ветвления по условию (`decision`), начальное состояние, конечное состояние (`stop`) и символы возврата из процедуры (`return`). Назовём все эти элементы узлами конечного автомата.

Рассмотрим множество узлов *N* конечного автомата. Назовём узел $s \in N$ *входным для множества узлов N*, если существует переход `t (a->s; sig; act)` и $a \notin N$. Назовём узел $b \notin N$ выходным для множества узлов *N*, если существует переход `t (s->b; sig; act)` и $s \in N$.

Сформулируем необходимые условия для выделения части конечного автомата в метод:

> *Множество узлов N, не содержащее stop, и все переходы между ними можно перенести в отдельный метод, если для множества узлов N существует ровно один входной узел и не более одного выходного узла.*

Для поиска частей автомата, которые можно перенести в отдельный метод, должно использоваться то же самое условие. Необходимо искать множества узлов с одним входным и одним выходным узлом. На Рис. 8 приведён пример автомата, удовлетворяющего заданному условию. Данное условие не ограничивает количество переходов, по которым можно попасть во входной или выходной узел, и, как видно на рисунке, для входного узла `decision (x)` и для выходного узла `Q` существует по два перехода, ведущие в них.

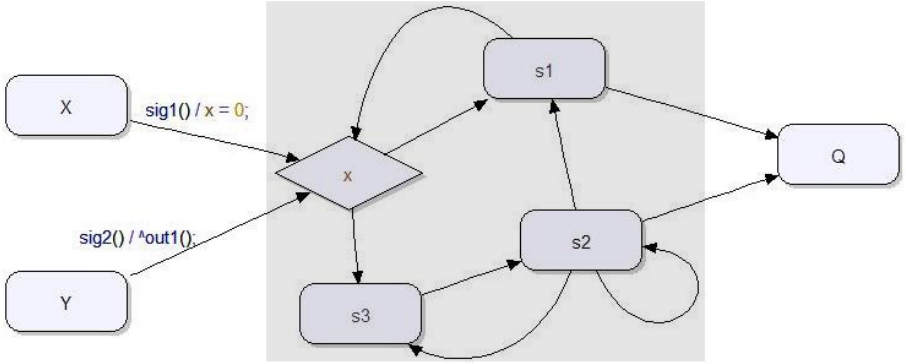

Рис. 8 Общий вид конечного автомата, удовлетворяющего условию выделения метода

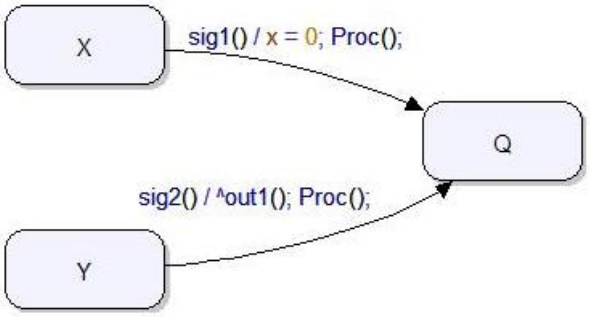

Рис. 9. Результат трансформации

Все узлы из выделенного множества, `decision (x)`, s_1, s_2, s_3 перемещаются в новый метод `Proc ()`. Переходы, ведущие во входное состояние, после трансформации напрямую направляются в выходное состояние `Q`. К действиям, выполняемым в этих переходах, добавляется вызов метода `Proc ()` (Рис. 9).

Переходы, ведущие в выходное состояние, переносятся в метод и направляются в символ возврата из процедуры (Рис. 10).

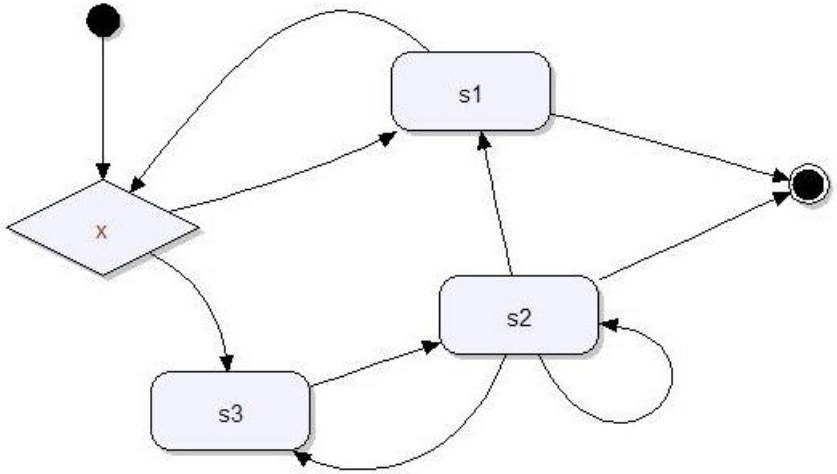

Рис. 10. Выделенный метод Proc ()

В основе необходимого свойства для выделения части конечного автомата в метод лежит следующая идея. Состояния, переносимые в выделяемый метод, перестают принадлежать исходному автомату, и, следовательно, некорректны команды перехода, которые приводят из состояний исходного автомата в состояния, перенесенные в выделенный автомат. Такие команды перехода (смены состояния) должны быть заменены командами вызова выделяемого метода. Однако у автомата, реализующего метод, может быть только одна входная точка, поэтому все такие команды должны осуществлять переход в одно и то же состояние. Аналогичные рассуждения касаются и обратных переходов из состояний выделенного метода в состояния исходного автомата.

Трансформации «выделение метода для конечного автомата», описанные в предыдущих двух параграфах, являются частными случаями трансформации общего вида, базирующейся на необходимом условии. В случае, когда входной узел совпадает с выходным, получаем возвратную часть конечного автомата, которую можно выделить в метод. Последовательные части

конечного автомата, очевидно, также удовлетворяют необходимому условию для выделения части конечного автомата в метод.

Существуют и другие частные случаи приведённого необходимого условия:

- Множество узлов не имеет ни одного выходного узла (и по необходимому условию не содержит `stop`), то есть возврат из созданного метода невозможен. Это значит, что в конечном автомате найден бесконечный цикл, который, возможно, представляет собой «серверную составляющую» исходного автомата.
- В случае, когда выходной узел — это `stop`, выделенный метод выполняет действия, после которых автомат завершает свою работу, так что, возможно, новый метод является неким аналогом деструктора в объектно-ориентированном программировании.

3.5. Трансформация «выделение составного состояния» для конечных автоматов UML

Автоматы, содержащие составные состояния, составляют менее 2 % от всех исследованных в [19] автоматов, 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. Таким образом, можно предположить, что многим разработчикам моделей, работавшим ранее на языке SDL, недостаточно известна эта конструкция.

Тем не менее, введение «составных» или «иерархических» состояний является достаточно важным методом композиции, используемым для создания легко воспринимаемого описания автомата.

В следующих параграфах описываются способы выделения составных состояний в конечных автоматах UML.

3.6. Выделение составного состояния для скрытия деталей реализации

В промышленных моделях для спецификации поведения объектов достаточно часто используются 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”, объект ожидает запрос на выполнение какой-либо операции. Получение запроса инициирует срабатывание перехода, в процессе которого выполняются необходимые действия. По завершении обработки объект вновь возвращается в исходное состояние. Частота использования подобного приема позволила даже выделить на его основе типичный образец проектирования, получивший название «Ромашка», за счет визуальной схожести описания автомата с цветком. Однако даже в тех случаях, когда

автомат, описывающий сложный компонент системы, не удовлетворяет целиком описанному образцу, зачастую удается выделить некоторые элементы, описание которых ему соответствует. Обособить такие части можно за счет использования описанной в этом параграфе трансформации.

При наличии нескольких «основных» состояний детали, описания каждого из них могут значительно усложнять диаграмму, как это показано на Рис. 11. Перегруженные символами диаграммы затрудняют понимание и должны быть преобразованы к более простому виду.

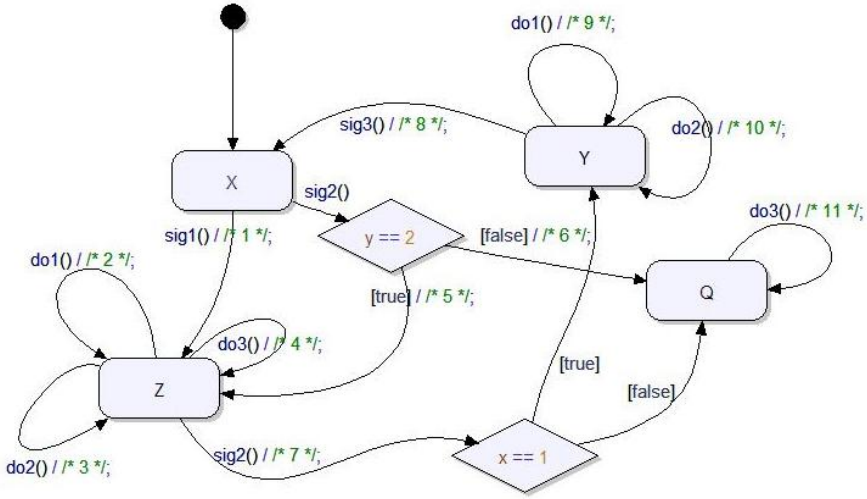

Рис. 11. Исходный конечный автомат

Для выполнения трансформации выбирается состояние, имеющее возвратные переходы, например *Z*. Далее это состояние заменяется составным состоянием *Z*, содержащим одно состояние *Z_internals*. Для спецификации внутренностей составного состояния *Z* создается новая диаграмма, на которой определяется начальный переход, ведущий в *Z_internals*. Все возвратные переходы состояния *Z* переносятся внутрь этого составного состояния и приписываются состоянию *Z_internals*. Результаты применения трансформации представлены на Рис. 12 и Рис. 13.

В результате применения данного преобразования описание автомата становится более иерархичным и поэтому более удобным для восприятия; структурируются и упрощаются диаграммы, специфицирующие автомат. Выполнение трансформации оправдывается в том случае, когда имеется состояние с большим количеством переходов в само себя. Как правило, наличие многих переходов из некоторого состояния в него же говорит о наличии особой семантической роли у функциональности, реализуемой

автоматом только в этом состоянии. Выделение составного состояния позволит в данном случае подчеркнуть эту функциональность не только семантически, но и графически посредством переноса ее описания на новую диаграмму.

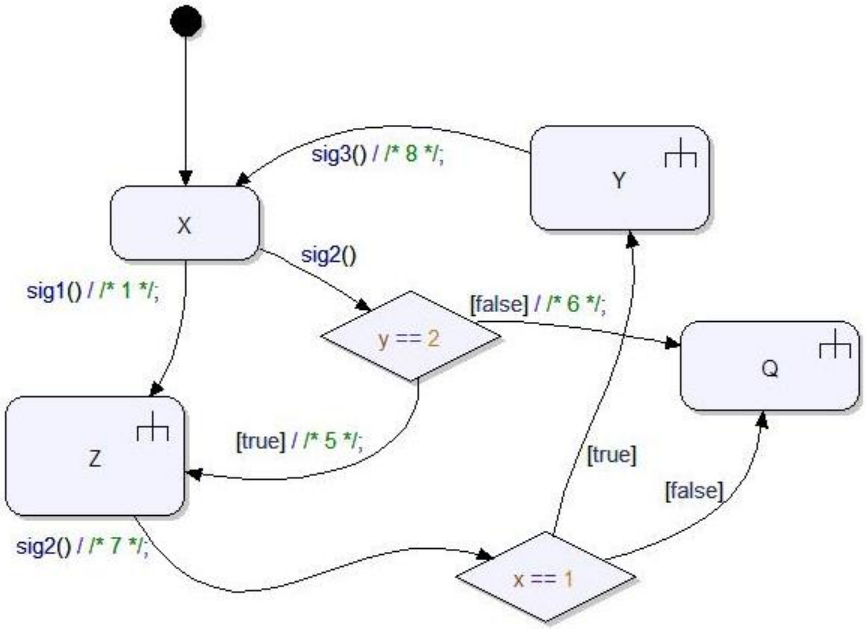

Рис. 12. Результат трансформации

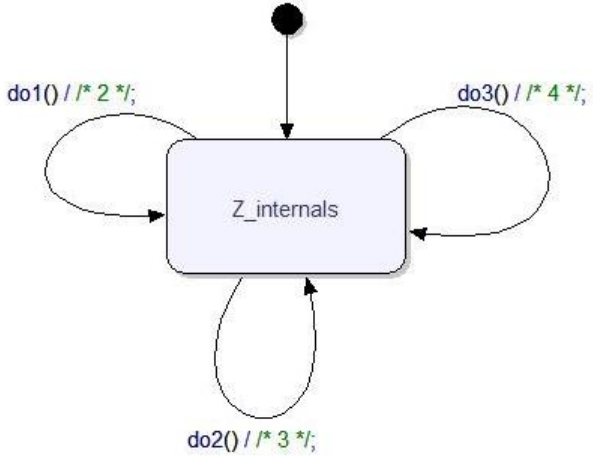

Рис. 13. Результат трансформации

В результате применения данного преобразования описание автомата становится более иерархичным и поэтому более удобным для восприятия; структурируются и упрощаются диаграммы, специфицирующие автомат. Выполнение трансформации оправдывается в том случае, когда имеется состояние с большим количеством переходов в само себя. Как правило, наличие многих переходов из некоторого состояния в него же говорит о наличии особой семантической роли у функциональности, реализуемой автоматом только в этом состоянии. Выделение составного состояния позволит в данном случае подчеркнуть эту функциональность не только семантически, но и графически посредством переноса ее описания на новую диаграмму.

Данное преобразование может быть применено к любому состоянию автомата, если это оправдывается с точки зрения семантики.

3.7. Метод текстового сравнения переходов

Как и при написании программ на традиционных языках программирования, при создании UML-моделей крайне желательно избегать дублирования элементов модели, повторно используя вместо этого существующие элементы, так как наличие нескольких копий каких-либо элементов модели может осложнять понимание и модификацию. В контексте исполняемых моделей, специфицированных посредством конечных автоматов, на практике наиболее часто встречаются дублирование общей архитектуры автомата и дублирование переходов. В первом случае в модели будет встречаться несколько похожих на вид, но необязательно эквивалентных автоматов. Избежать дублирования в данной ситуации весьма затруднительно, можно лишь описать повторяющуюся часть автомата как типичный прием проектирования автоматов (*pattern*), сохранив информацию об использовании этого образца в комментарии к автомату.

В случае наличия в автомате эквивалентных переходов дублирование можно устранить за счет применения трансформаций, описанных в следующих параграфах. Однако задача автоматического обнаружения эквивалентных переходов оказывается достаточно сложной и весьма актуальной для больших автоматных моделей.

Эквивалентность переходов в автомате означает совпадение списка стимулов, инициирующих эти переходы, а также эквивалентность действий, выполняемых при срабатывании заданного перехода. Если автоматическая проверка эквивалентности списков сигналов, инициирующих сравниваемые переходы, не представляет никакой трудности, то проблема определения эквивалентности двух программ, к которой сводится проверка эквивалентности последовательности действий, выполняемых в переходах, алгоритмически неразрешима. Кроме того, сложно определить критерии эквивалентности двух автоматов.

Несмотря на это, для конечных автоматов весьма хорошо зарекомендовал себя следующий простой подход, суть которого состоит в сравнении текстовых строк, описывающих переход. При очевидной простоте реализации этот способ позволяет достаточно эффективно находить в автомате эквивалентные переходы. Эквивалентность текстуально совпадающих действий, приписанных переходам, практически очевидна. Достаточно учесть, что оба перехода описаны в контексте одного конечного автомата, и поэтому все использованные при их описании идентификаторы, например, имена функций и переменных, означают в обоих переходах одни и те же объекты. Действительно, каждое имя в переходе либо относится к объекту, объявленному локально, и тогда его определения текстуально совпадают в обоих переходах, и для них можно повторить это рассуждение; либо относится к объекту, определенному в контексте автомата и выше, но в этом случае правила, по которым идентификатору сопоставляется элемент модели, дадут одинаковый результат для обоих переходов, так как их применение в обоих случаях можно начинать с автомата, содержащего оба рассматриваемых перехода, поскольку сами переходы не содержат подходящего объекта. Отсюда следует, что текстуальное совпадение переходов гарантирует их эквивалентность.

Тем не менее, отсутствие текстуального совпадения отнюдь не обязательно означает отсутствие эквивалентности: к примеру, если в одном из переходов выполняются действия {*a*++; *b*++}, а в другом {*b*++; *a*++}, где и *a* и *b* являются атрибутами класса, содержащего автомат, то эти переходы можно считать эквивалентными, хотя текстуально они различны. Однако текстуальное сравнение может быть использовано и позволяет получать приемлемые результаты.

Метод текстуального сравнения может быть усовершенствован следующим образом: перед сравнением из строк, описывающих переходы, удаляются фрагменты, не имеющие отношения к выполняемым действиям, например, комментарии. Кроме того, все локальные идентификаторы (в случае, если их количество в обоих переходах одинаково) следует переименовать так, чтобы они имели одни и те же имена в порядке их объявления.

Автоматизация поиска эквивалентных переходов позволяет упростить процесс применения трансформаций, устраняющих дублирование переходов.

3.8. Выделение составного состояния по общему переходу

Рассматриваемую здесь трансформацию следует применять, когда в исходном автомате достаточно много состояний, и при этом можно выделить группу семантически связанных состояний, для каждого из которых существует переход, эквивалентный некоторому заданному переходу.

Достаточно часто при создании промышленных моделей возникают ситуации, когда невозможно знать заранее точные время и условия осуществления какого-либо события, например, прихода сигнала от окружения системы. Если

событие достаточно важно и не допускает отложенной обработки, то разработчик конечного автомата будет вынужден описать реакцию на это событие сразу в нескольких возможных состояниях, поскольку он не может знать, в каком именно состоянии будет находиться автомат в момент осуществления события.

Такие описания затрудняют понимание и приводят к ненужному дублированию кода, что повышает вероятность возникновения ошибок в случае изменения реакции на это событие или добавление в автомат новых состояний.

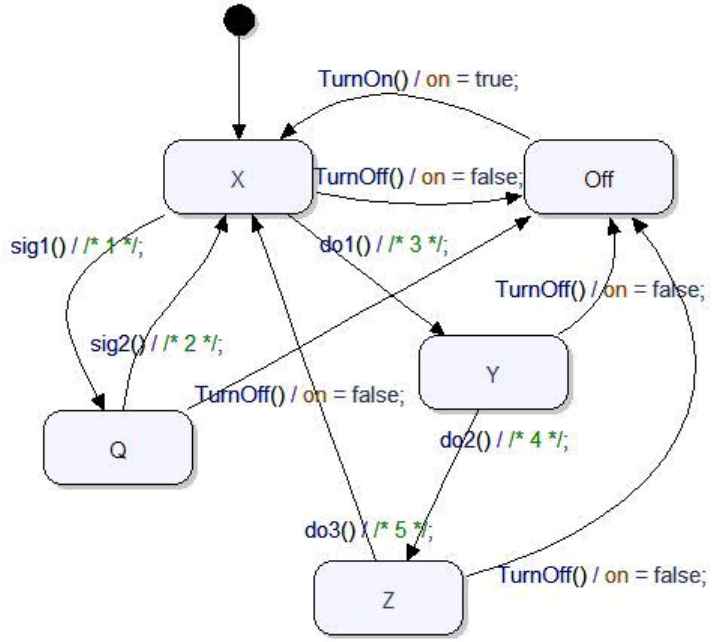

Рис. 14. Исходный конечный автомат

Данная трансформация состоит из двух этапов, каждый из которых является композиционным преобразованием. Перед проведением преобразований должен быть выбран переход, копии которого определены сразу для нескольких состояний в исходном автомате. На первом этапе создается новое иерархическое состояние, в которое включаются все состояния, имеющие переход, эквивалентный выбранному. На втором этапе для всех состояний удаляются переходы, эквивалентные выбранному, а для вновь созданного иерархического состояния такой переход добавляется.

Применение данной трансформации к автомату, изображенному на рис. 14, позволяет выделить составное состояние *On*, содержащее состояния *X*, *Y*, *Q*, *Z*, если в качестве выделенного перехода использовать переход по сигналу *TurnOff ()*. Для спецификации внутренностей составного состояния создается диаграмма, приведенная на рис. 15, а модифицированный исходный автомат показан на рис. 16. Введение иерархического состояния позволило упростить исходную диаграмму за счет удаления дублировавшихся переходов и сделать ее более наглядной и понятной. Кроме того, замена множества одинаковых переходов одним помогает избежать ошибок при изменении спецификации действий, выполняемых при срабатывании перехода.

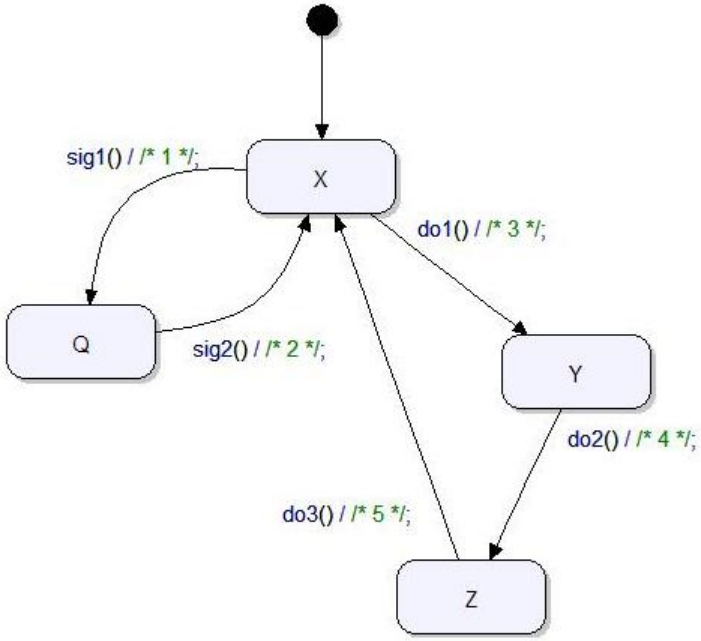

Рис. 15. Результат трансформации: спецификация составного состояния On

Создание в автомате *S* нового составного состояния *B*, явно содержащего состояния $A_1 \ldots A_n$, корректно в том и только том случае, когда это добавление не нарушает строгую иерархию состояний по включению. Иными словами, новое состояние, будучи изображенным на любой диаграмме состояний, описывающей автомат не должно пересекать границы уже существующих состояний. Перенос перехода *T* из состояний $D_1 \ldots D_n$ в состояние *E*, корректен только в том случае, если *E* является составным состоянием и не содержит явно никаких состояний кроме $D_1 \ldots D_n$, а в каждом из состояний $D_1 \ldots D_n$ есть

либо переход эквивалентный *T*, либо переход срабатывающий при тех же условиях, что и *T*.

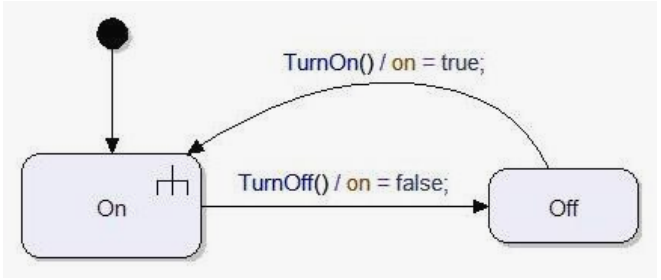

Рис. 16. Результат трансформации: модифицированный исходный автомат

3.9. Выделение составного состояния по общему возвратному переходу

По причине описанного ранее негативного влияния дублирования переходов желательно устранять эквивалентные действия, даже если они находятся на неэквивалентных переходах. В этом параграфе будет описана трансформация, позволяющая устранить дублирование действий в переходах отличающихся лишь состоянием, в которое они переводят автомат.

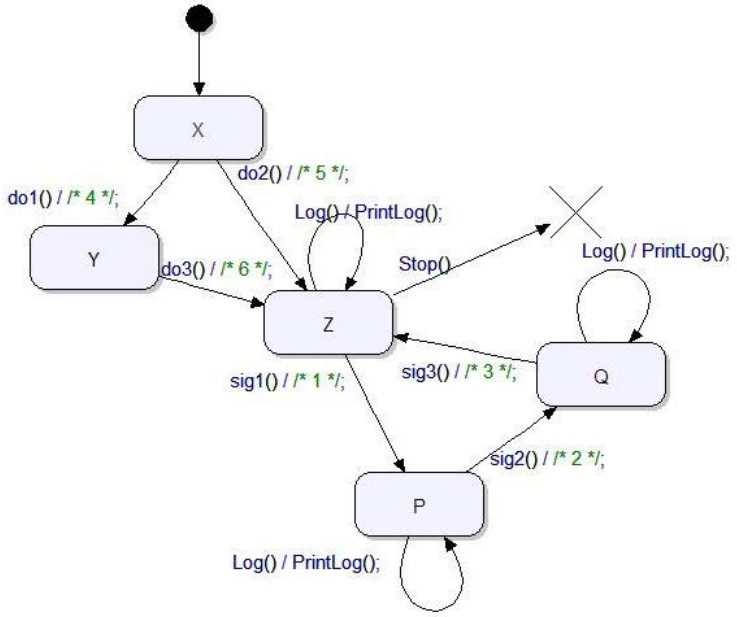

Рис. 17. Исходный конечный автомат

Для сложных автоматов достаточно типична ситуация, когда какое-либо событие должно быть одинаковым образом обработано в нескольких состояниях, при этом его обработка не должна прерывать основной ход событий, то есть по его окончании автомат должен возвратиться в исходное состояние. Примером таких событий может быть протоколирование какой-либо активности либо другие «фоновые» для автомата задачи. Моделирование такого поведения вынуждает разработчиков определять для каждого из состояний, в котором должна быть обработана «фоновая» активность, переход, содержащий эквивалентную последовательность действий и возвращающий автомат в исходное состояние. Такие переходы избыточны, загромождают описание автомата и мешают его восприятию, как, например, на Рис. 17.

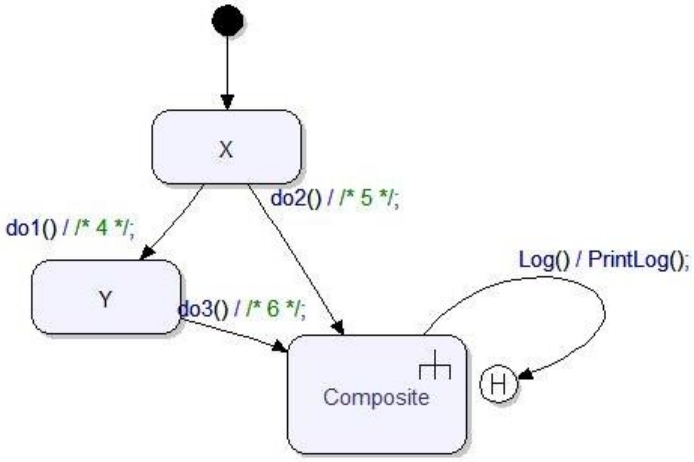

Рис. 18. Результат трансформации

Предлагаемая трансформация выполняется во многом аналогично предыдущей. На первом шаге выделяется множество состояний $A_1 \ldots A_n$ автомата S, для каждого из которых существует переход T_i, обладающий следующими свойствами: он срабатывает при условии t, в нем выполняются действия act, и после его завершения автомат возвращается в исходное состояние A_i. К автомату S добавляется новое составное состояние B, содержащее состояния $A_1 \ldots A_n$, а для спецификации состояния B создается отдельная диаграмма D. К автомату S добавляется переход T из состояния B по стимулу t, выполняющий действия act и завершающийся оператором перехода в состояние истории (*deep history state*). Если у множества состояний $A_1 \ldots A_n$ имеется только одна входная точка A_1, то возможно дальнейшее преобразование автомата. Все операторы перехода в состояние A_1 заменяются операторами перехода в состояние B, а к состоянию B добавляется стартовый переход, ведущий в состояние A_1. После этого становится возможным

переместить состояния $A_1 \ldots A_n$ с исходной диаграммы на диаграмму D (Рис. 19), добавив вместо них состояние B и переход T к исходной диаграмме (Рис. 18).

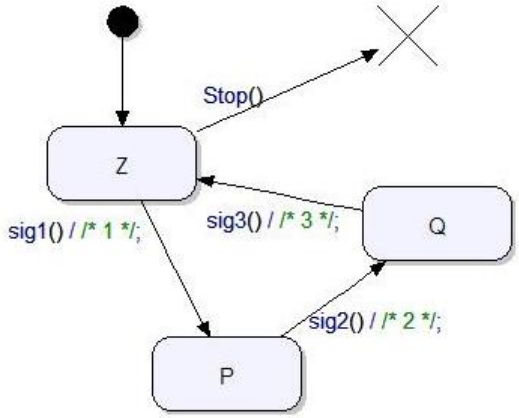

Рис. 19. Результат трансформации

Данная декомпозиция позволяет упростить описание исходного автомата, сделав его более удобным, а также улучшает структуру автомата, устраняя дублирование действий, выполняемых при срабатывании переходов.

«Выделение составного состояния по общему возвратному переходу» может быть применено, если в автомате можно выделить такую группу состояний с единственной входной точкой, что для каждого состояния группы определен возвратный переход по одинаковому для всех состояний стимулу и с эквивалентными для всех состояний действиями.

3.10. Выделение составного состояния для моделей, импортированных из SDL

Несмотря на то, что сейчас язык UML де-факто является стандартным языком моделирования, среди используемых сегодня в реальных промышленных проектах моделей можно выделить класс систем, которые изначально были описаны на других языках моделирования — преимущественно SDL. В основном, язык SDL использовался для спецификации телекоммуникационных систем. Однако преимущества UML подтолкнули многих разработчиков к преобразованию их моделей в UML. При этом, если описание статической структуры системы на SDL достаточно легко моделируется в UML, то для некоторых элементы SDL, используемые для спецификации конечных автоматов, отсутствуют UML-аналоги, например, понятие «мультисостояния» (*multistate*).

В SDL в символе состояния можно перечислить несколько имен состояний, и тогда все переходы, изображенные выходящими 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 показывают, что символ «*» присутствует в 12 % символов состояния, что свидетельствует о достаточно активном использовании этой подстановки и необходимости ее более детального изучения [19].

Отсутствие методов трансформации таких конструкций заставило разработчиков инструментальных средств моделирования пойти по пути внедрения в свои продукты для работы с UML недостающих элементов языка SDL. Так, например, в среде моделирования Telelogic Tau 3.0 поддерживаются «мультисостояния», в том числе, и «*-состояния». Однако для создания моделей, соответствующих стандартному языку UML, необходимо преобразовать подобные состояния в другие элементы, сохранив при этом функциональность автомата.

Предлагаемая трансформация упрощает мультисостояния следующим образом. Для каждого символа мультисостояния к исходному автомату добавляется новое составное состояние B, которое содержит в точности все состояния $A_1 \ldots A_n$, перечисленные в символе мультисостояния. Все переходы, приписанные мультисостоянию, переносятся в созданное составное состояние. После этого мультисостояние может быть безболезненно удалено из автомата. Кроме того, если у множества состояний $A_1 \ldots A_n$ имеется всего одна входная точка A_1, то можно определить отдельную диаграмму для спецификации внутренностей созданного составного состояния B, переместив на нее все описания, касающиеся множества состояний $A_1 \ldots A_n$, заменив при этом в исходной диаграмме все операторы перехода в A_1 операторами перехода в состояния B.

Пример исходной диаграммы приведен на Рис. 20, а результат преобразования – на Рис. 21. Выделение иерархического состояния также возможно, когда в качестве символа мультисостояния указан «*», то есть определяется переход для всех состояний автомата.

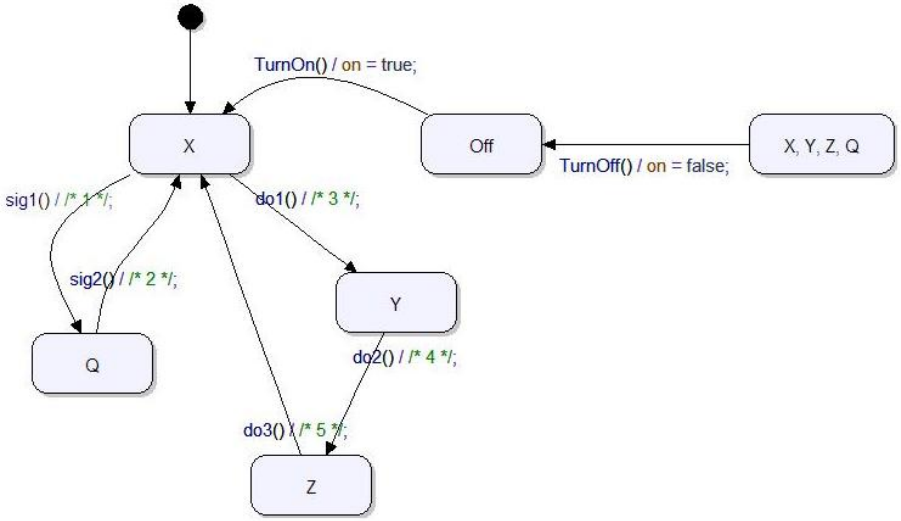

Рис. 20. Исходный конечный автомат

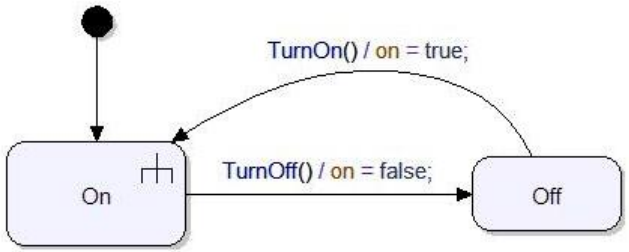

Рис. 21. Результат трансформации

3.11. Пример «Мобильный телефон»

Продемонстрируем применение предложенных трансформаций 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. 22).

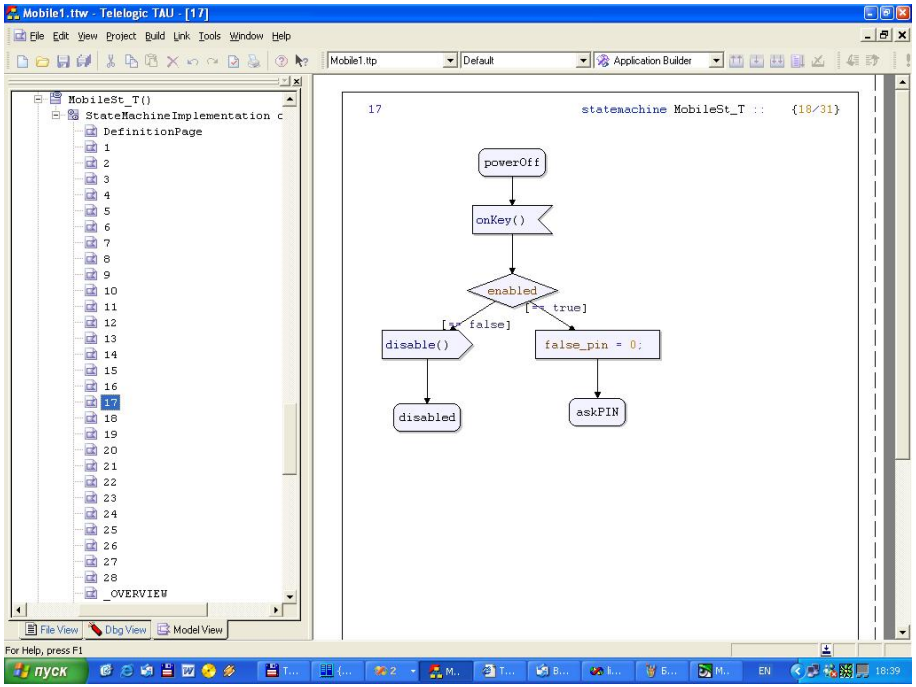

Рис. 22. Исходный вид конечного автомата

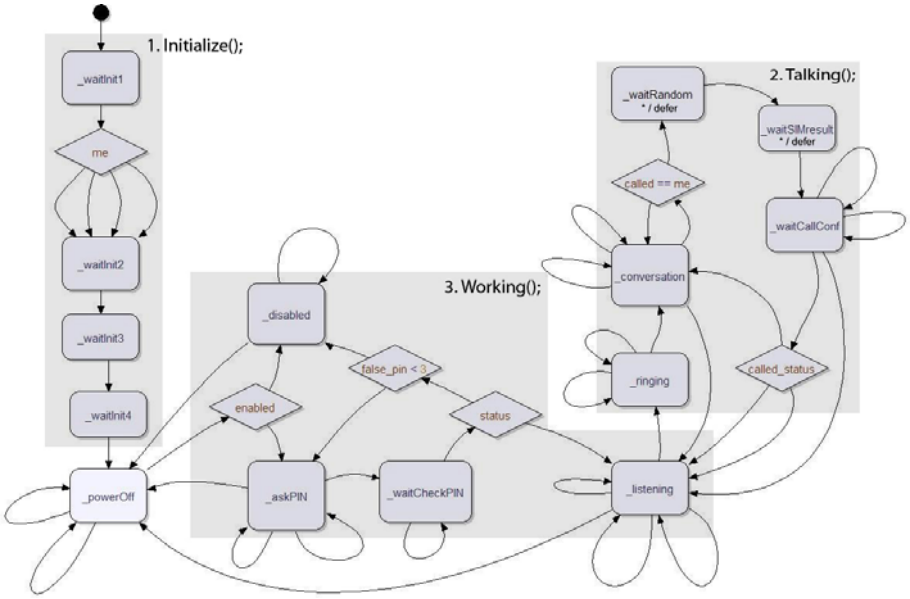

Рис. 23. Краткое описание всего конечного автомата

Такое представление, в котором используется нотация SDL для описания переходов, не позволяет понять цельную структуру конечного автомата. Для упрощения понимания была сгенерирована дополнительная OVERVIEW-диаграмма, описывающая весь конечный автомат в нотации UML (Рис. 23).

Приведённый алгоритм позволяет найти и выделить из данного конечного автомата три метода. На первом шаге в метод *Initialize* () выделяются четыре последовательных состояния. На втором шаге выделяется метод *Talking* (), после чего становится возможным выделить ещё один метод, который назовём *Working* ().

Обратим внимание на то, что выделение метода *Working* () возможно только после выделения метода *Talking* (). 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. 24). Теперь он содержит только одно состояние (вместо четырнадцати состояний в исходном автомате) и вызов двух методов.

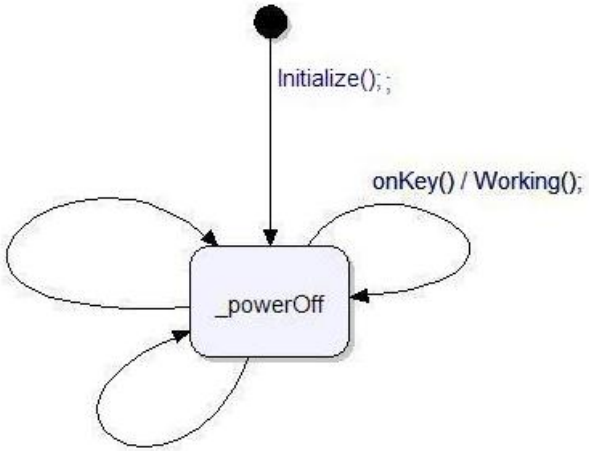

Рис. 24. Результат трансформации

Выделены три метода *Initialize* (), *Talking* () (Рис. 25) и *Working* () (Рис. 28), содержащие 4, 5 и 4 состояния соответственно.

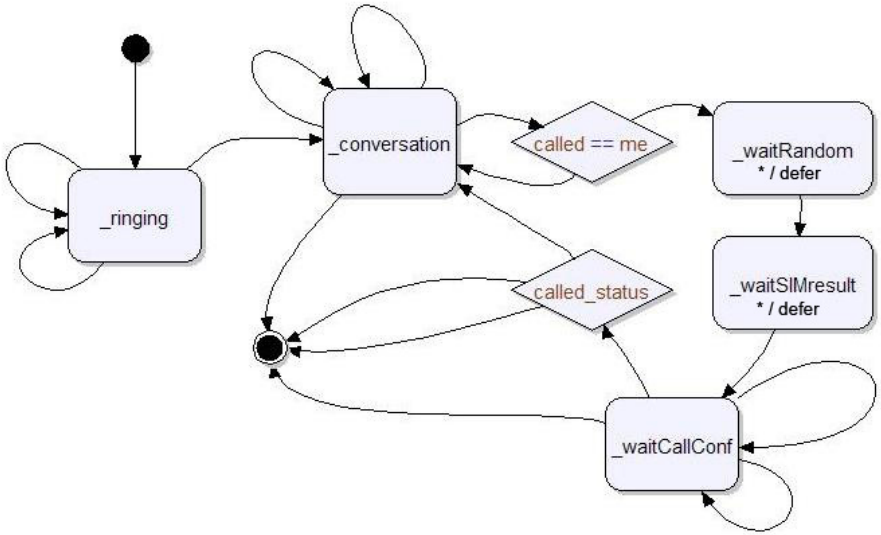

Рис. 25. Метод Talking ()

В методе *Talking* () три состояния из пяти содержат пары одинаковых возвратных переходов, а остальные два состояния определяют реакцию (*defer*) на те сигналы, которые инициируют возвратные переходы. Таким образом, к данному методу можно применить трансформацию «выделение составного состояния по общему возвратному переходу». В результате пара общих переходов будет вынесена с основной диаграммы и применена к внешнему составному состоянию (Рис. 26).

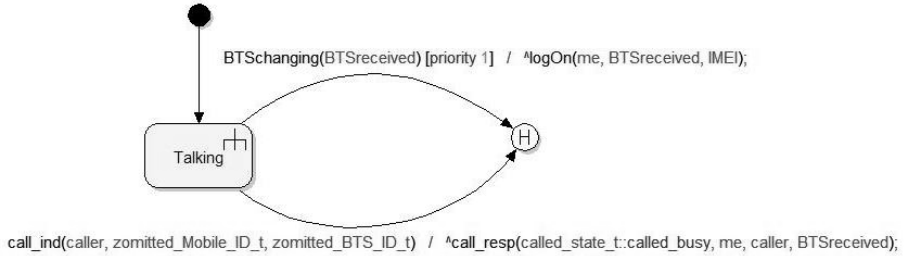

Рис. 26. Результат трансформации метода Talking ()

После данной трансформации, конечный автомат, описывающий поведение метода *Talking* (), будет перенесён в составное состояние *Talking*, и из него будут удалены три пары возвратных переходов, что повысит наглядность (Рис. 27).

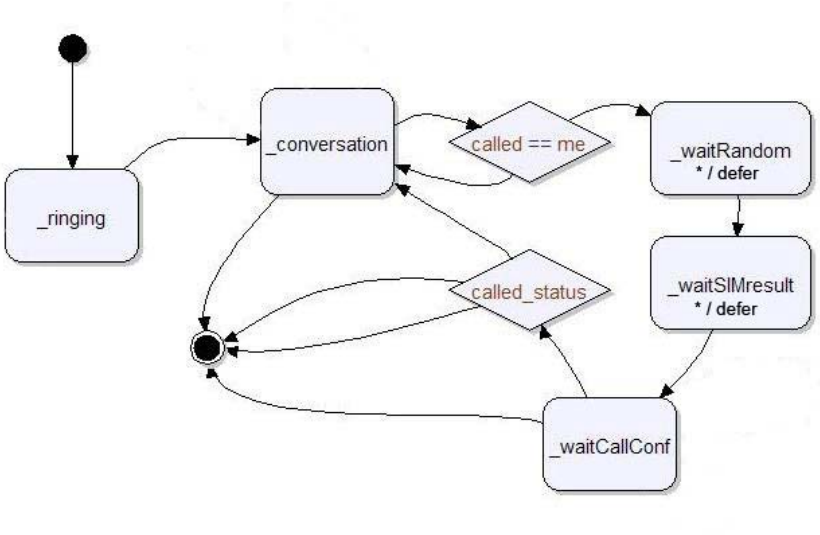

Рис. 27. Составное состояние Talking

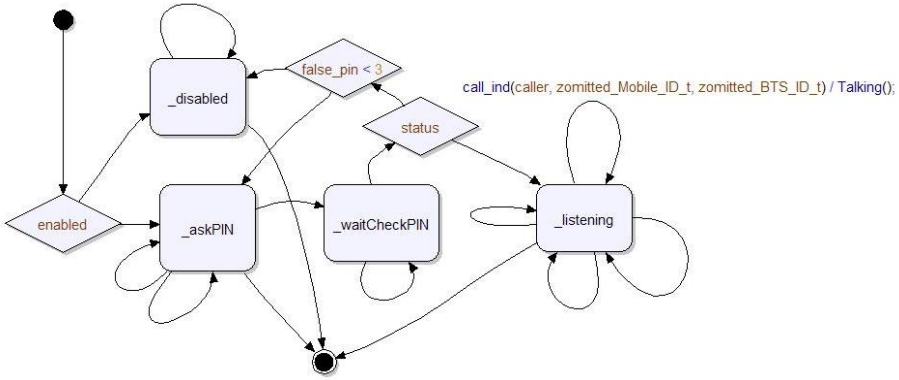

Рис. 28. Метод Working ()

Применив трансформацию «выделение составного состояния для скрытия деталей реализации» для метода *Working* () и выделив три составных состояния, получим более компактную диаграмму с меньшим количеством переходов (Рис. 29).

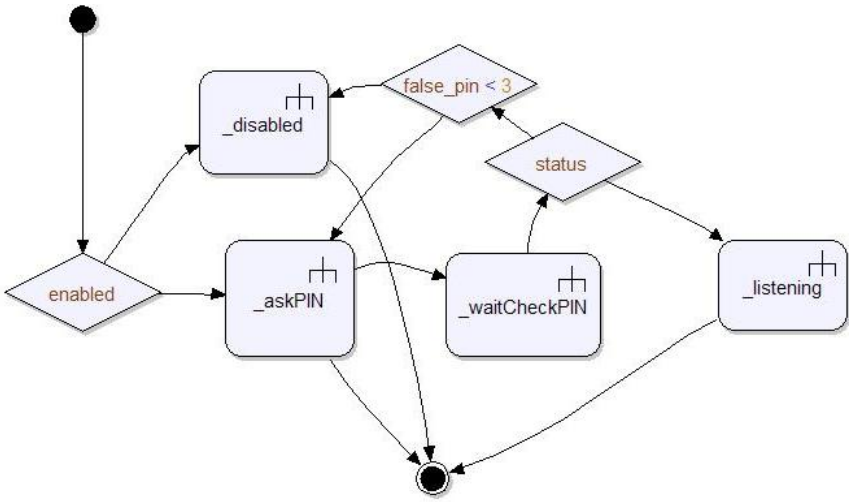

Рис. 29. Результат трансформации метода Working ()

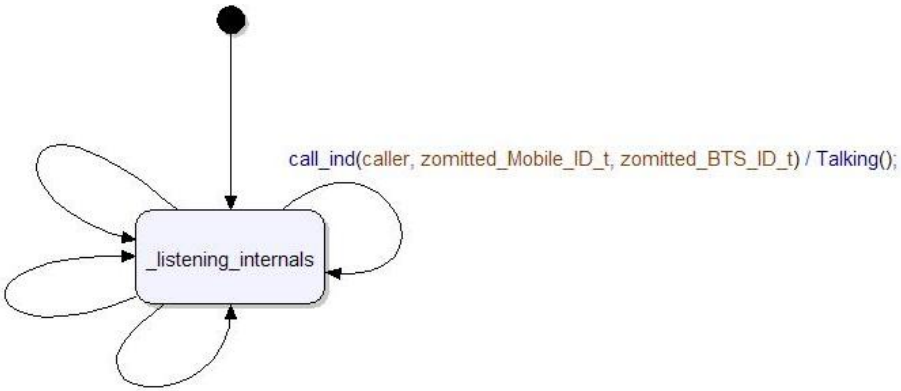

Рис. 30. Составное состояние listening_internals

4. Заключение

В статье были предложены 6 новых методов трансформации исполняемых моделей на языке UML, основанных на композиции и декомпозиции. Была продемонстрирована практическая применимость предлагаемых методов как средства упрощения восприятия и улучшения структуры сложных автоматов.

Каждый из предложенных методов допускает как автоматизацию выполнения преобразования, так и автоматический поиск частей модели, к которым это преобразование применимо и, вероятно, его применение целесообразно. Приоритетными направлениями дальнейшей работы являются реализация

модуля автоматического поиска и применения трансформаций в среде моделирования, наряду с продолжением работ по разработке и исследованию новых методов трансформации.

Литература

[1] Меллор С., Кларк Э., Футагами Т. Разработка на базе моделей // Открытые системы. 2003. № 12 [HTML] (http://www.osp.ru/os/2003/12/030.htm).
[2] Селич Б. Практические аспекты разработки на базе моделей // Открытые системы. 2003. № 12 [HTML] (http://www.osp.ru/os/2003/12/033.htm).
[3] Буч Г., Рамбо Д., Джекобсон А. UML Руководство пользователя. — М.: ДМК Пресс, 2001. 432 с.
[4] Schmidt D. Model-Driven Engineering // J. Computer. 2006. N 2. P. 25-31.
[5] Рамбо Дж., Якобсон А., Буч Г. UML: специальный справочник. — СПб.: Питер, 2002. 656 с.
[6] Фаулер М., Бек К., Брант Д., Робертс Д., Апдайк У. Рефакторинг: улучшение существующего кода. — СПб.: Символ-Плюс, 2002. 432 с.
[7] Opdyke W. Refactoring Object-Oriented Frameworks. Illinois: University of Illinois, 1992. 151 p.
[8] Object Management Group. UML 2.0 Superstructure Specification [PDF, PS] (http://www.omg.org/cgi-bin/doc?formal/05-07-04).
[9] Object Management Group. MOF 2.0/XMI Mapping Specification, v2.1 [PDF] (http://www.omg.org/docs/formal/05-09-01.pdf).
[10] Object Management Group. OCL 2.0 Specification [PDF] (http://www.omg.org/docs/ptc/05-06-06.pdf).
[11] Kovse J., Harder T. Generic XMI-Based UML Model Transformations [PDF] (http://wwwdvs.informatik.uni-kl.de/pubs/papers/KH02.OOIS.pdf).
[12] International Telecommunication Union. Specification and description language (SDL) [PDF] (http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf).
[13] Boger M., Strum T., Fragemann P. Refactoring Browser for UML // Revised Papers. International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World. London, UK: Spinger-Verlag, 2002. P. 366-377.
[14] Sunye G., Pollet D., Traon D., Jezequel J.-M. Refactoring UML models // Proceedings. UML'01—Modeling Languages, Concepts, and Tools Fourth International Conference. Toronto, Canada: LNCS, 2001. P. 134-148.
[15] Samek M. Practical Statecharts in C/C++. San Francisco: CMP Books, 2002. 387 p.
[16] Van Gorp P.; Stenten H.; Mens T., Demeyer S. Towards Automating Source Consistent UML Refactorings [PDF] (http://www.lore.ua.ac.be/refactoringProject/publications/Towards_automating_source-consistent_UML_Refactorings.pdf).
[17] Astels. D. Refactoring with UML // 3rd International Conference on eXtreme Programming and Flexible Proceses in Software Engineering. Marchesi, Italy: M and Succi, 2002. P. 67-70.
[18] Mens T., Van Eetvelde N., Janssens D., Demeyer S. Formalising refactorings with graph transformations // J. of Software Maintanence and Evolution. 2004. N 3. P. 1001-1025.
[19] Волкова Е. Д., Страбыкин А. Д. Анализ и трансформации исполняемых UML моделей // Труды Института системного программирования. — М.: ИСП РАН, 2006. С. 171-192.