Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

А.П.Лукьянова

МОДЕЛИРОВАНИЕ И ВЕРИФИКАЦИЯ ПОТОКОВ ДАННЫХ НА ДИАГРАММАХ СОСТОЯНИЙ

Бакалаврская работа

Научный руководитель: канд. техн. наук Нарвский А.С.

Консультант: докт. техн. наук, профессор Шалыто А.А.

Санкт-Петербург
2005

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Объектно-ориентированная методология проектирования включает в себя три модели [1]. *Объектная (статическая) модель* показывает статическую структуру проблемной области, для которой разрабатывается система. *Динамическая модель* отображает поведение системы, в особенности последовательность взаимодействий. *Функциональная модель* описывает вычисления в системе. Она показывает, каким образом выходные данные вычисляются по входным данным, не рассматривая порядок и способ реализации вычислений.

Существует множество методологий для определения моделей всех трех типов. Разработан ряд технологий для получения кода на целевом языке на основе таких моделей. В частности, в последнее время широкое распространение получили пакеты для проектирования программ на языке *UML* [2, 3]. Они позволяют разработчику создавать модель программы с помощью набора диаграмм, некоторые из которых в дальнейшем могут быть преобразованы в код на целевом языке программирования.

Наибольшее развитие и формализацию в различных методологиях получили диаграммы статической модели. Описание же динамической модели системы часто носит неформальный характер. Диаграммы, описывающие поведение, используются только для пояснения предметной области, а семантическая связь с создаваемым в последствии кодом отсутствует.

Исключением из этого правила является методология, предложенная в статье [4]. В ней авторы формулируют метод проектирования событийных объектно-ориентированных программ с явным выделением состояний. Этот подход предлагает описывать статическую модель системы любыми стандартными средствами и определяет способ задания динамической модели системы,

основанный на *SWITCH*-технологии [5, 6], адаптированной таким образом, чтобы удовлетворять нотациям стандарта *UML*.

Достоинствами этой методологии являются уменьшение разрыва между фазами проектирования и реализации (прямая связь динамической модели с кодом), соответствие стандарту *UML*, а также наличие достаточно удобного инструмента разработки, реализовывающего эту методологию – программного пакета *UniMod* [7] на платформе *Eclipse* [8].

Настоящая работа расширяет описанную в работе [4] методологию, добавляя к ее статической и динамической моделям элементы функциональной модели, а именно, информацию о потоках данных. Обоснованием для такого расширения служат выводы, сделанные на основе опыта применения автоматной технологии с использованием программного пакета *UniMod*.

В первой части работы рассматриваются существующие модели данных (в частности, диаграммы потоков данных структурного анализа и диаграммы деятельности *UML*). По результатам этого рассмотрения формулируется способ описания потоков данных «поверх» диаграмм состояний.

Также в первой части работы формализуется правило верификации потоков данных, описанных предложенным методом. Автоматическая верификация модели по этому правилу позволяет существенно уменьшить время, требующееся на отладку программ. Реализация автоматической верификации описана во второй части работы.

1. ИССЛЕДОВАНИЕ

1.1. Описание исходной методологии

Построение модели системы начинается с анализа предметной области. Стандартными средствами [1, 9 – 13] строится статическая модель системы, и,

если в дальнейшем действовать согласно автоматному подходу, в статической модели выделяются объекты и управляющие ими автоматы.

Рассматриваемая автоматная методология описывает динамическую модель системы с помощью так называемой *модели управления*. Построение этой модели состоит из следующих этапов:

– в статической модели выделяются автоматы, все возможные объекты, являющиеся источниками событий, и объекты управления, которые также могут являться источниками входных воздействий;

– все объекты компонуются на схеме связей автомата, которая представляется с помощью диаграммы классов *UML*;

– каждое входное и выходное воздействие реализуется вручную в соответствии с его функциональностью;

– для каждого автомата описывается поток управления, представленный в виде графа переходов типа *Мура-Мили*, заданного диаграммой состояний *UML*, в которой на дугах указываются события (**eN**), охранные условия (описанные с помощью входных воздействий (**xN**)) и выходные воздействия (**zN**);

– с использованием подходящего инструмента разработки программного обеспечения, например программного пакета *UniMod* и платформы *Eclipse*, диаграмма классов и графы переходов могут быть автоматически преобразованы в исходный код на целевом языке или интерпретированы специальным интерпретатором.

Необходимо отметить, что основным принципом рассматриваемой методологи является сужение разрыва между фазами проектирования и реализации, облегчение и автоматизация кодирования. Если рассматривать

реализацию методологии пакетом *UniMod*, этот принцип достигается с помощью следующих возможностей:

- интерпретация или автоматическая генерация кода;
- запуск и отладка программы внутри среды разработки;
- автоматическое выделение семантических и синтаксических ошибок модели;
- автоматическое завершение ввода и исправление ошибок;
- форматирование и рефакторинг кода.

Как следует из ее описания, данная методология состоит из элементов статической (схемы связей) и динамической (графы переходов) моделей. Рассмотрим теперь различные составляющие функциональной модели и выделим те из них, которые целесообразно добавить в автоматную методологию.

1.2. Функциональная модель и диаграммы потоков данных

Основными составляющими функциональной модели являются диаграммы потоков данных [14, 15], которые показывают потоки значений от внешних входов через операции и внутренние хранилища данных к внешним выходам.

Наибольшее развитие диаграммы потоков данных получили в методологии *SA/SD* (*Structured Analysis/Structured Design*) [16]. В этой методологии ведущей является именно функциональная модель (набор диаграмм потоков данных), на втором месте по важности стоит динамическая модель и на последнем месте – объектная модель. Таким образом, в методологии *SA/SD* проектируемая система описывается с помощью процедур (процессов), что не соответствует объектно-ориентированному подходу и поэтому диаграммы потоков данных и не были включены в состав стандарта *UML*.

Процедурная ориентированность методологии *SA/SD*, и в частности диаграмм потоков данных, является ее недостатком: системы, спроектированные по этой методологии, имеют менее четкую структуру, так как разбиение процесса на подпроцессы во многом произвольно, зависит от реализации и обычно плохо отражает структуру проектируемой системы. При этом диаграммы состояний не отражают реализацию системы, являясь лишь средством пояснения модели для разработчика, а следовательно, не могут быть использованы для генерации кода.

В силу этого недостатка, а также из-за противоречия между объектно-ориентированным подходом и диаграммами потоков данных, внесение их в модель в чистом виде нецелесообразно.

1.3. Функциональная модель и описание операций

Помимо диаграмм потоков данных, функциональная модель описывает смысл операций объектной модели и действий динамической модели, а также ограничения на объектную модель. Именно этот вид информации о данных и предполагается внести в рассматриваемую методологию.

Согласно функциональной модели, все нетривиальные операции можно разделить на три категории: *запросы*, *действия* и *активности*. Запросом называется операция без побочных эффектов над видимым извне объекта его состоянием. Примером запроса может служить входное воздействие автоматной модели. Действием называется операция, имеющая побочные эффекты, которые могут влиять на целевой объект и на другие объекты системы, которые достижимы из целевого объекта. Примером действия является выходное воздействие.

Как следует из примеров, между понятиями функционального анализа и элементами автоматной модели устанавливается соответствие. Следовательно,

включение элементов функциональной модели (описания операций) в автоматную модель логически обоснованно.

Все операции в модели должны быть специфицированы. Отметим, что спецификация операции описывает только те изменения, которые видны вне ее. Операция может быть реализована таким образом, что при ее выполнении будут использоваться некоторые значения, определенные внутри операции. Эти детали реализации скрыты и не участвуют в определении внешнего эффекта операции.

Обыкновенно спецификация операции содержит ее сигнатуру (имя, количество, порядок и типы параметров, количество, порядок и типы возвращаемых значений) и описание ее эффекта (действия, преобразования). Операции рассматриваемой автоматной модели не имеют параметров. Выходные воздействия к тому же не имеют возвращаемых значений. Однако и они, и входные воздействия, могут получать и изменять данные в памяти автомата. Запрашиваемые действием элементы данных будем называть *входными данными* (или параметрами). Сохраняемые в память автомата данные будем называть *выходными данными действия*. Кроме того, события в автоматной модели также несут с собой данные – свои параметры. Для обеспечения общности будем также называть их *выходными данными события*.

Итак, для автоматной модели целесообразно описывать имена операций (действий и событий), входные и выходные данные, а также эффект операций. Для описания эффекта в функциональной модели могут использоваться:

− математические формулы;

− табличные функции: таблицы, сопоставляющие выходные значения входным;

− уравнения, связывающие входные и выходные значения;

- аксиоматическое определение пред- и постусловий;
- таблицы принятия решений;
- псевдокод;
- естественный язык.

Определим наиболее подходящие способы для описания операций автоматной модели. Отметим еще раз, что несмотря на то, что события автоматной модели не являются операциями, они приносят с собой параметры, а следовательно также входят в модель данных и должны быть описаны.

Табличные функции для решения задачи не подходит, так как они не рассчитаны на произвольный набор данных. Более того, достаточно подробная спецификация того, как именно формируются выходные данные, необоснованна. Напомним, что входные воздействия при правильном (robust) автоматном программировании вообще не должны изменять значения входных данных. Выходные воздействия изменять их могут, но основная логика приложения реализуется автоматом (графом переходов), а эти воздействия (с точки зрения автоматного подхода!) представлены как точечные неделимые операции.

Таким образом, для спецификации воздействий и событий автоматной модели достаточно указания списков входных и выходных данных, а также описания эффекта этих действий на естественном языке.

Дополнительной возможностью является описание пред- и постусловий, задающих соотношения между входными данными (для предусловий) и входными и выходными данными (для постусловий). Ограничения, которым должны удовлетворять входные значения функции, называются ее предусловиями, а ограничения, которым удовлетворяют выходные значения функции – ее постусловиями.

Для описания пред- и постусловий могут быть использованы как математический, так и естественный языки. Компромиссом между ними является ставший популярным в последнее время язык описания ограничений *OCL* (*Object Constraint Language*) [18, 19], вошедший в состав стандарта *UML*. Варианты применения ограничений *OCL* для спецификации автоматной модели приведены в приложении 1. Реализация таких ограничений выходит за рамки данной работы [23-29].

Далее рассмотрим и проверим на примере предложенный в этом разделе способ спецификации операций автоматной модели.

1.4. Спецификация потоков данных поверх диаграммы состояний

В качестве примера используем модуль управления партнерами в Интернет-приложении *STEED Options Brokerage Web Site*, разработанном компанией *eVelopers*, в которой работает автор. На рис. 1 представлена схема навигации по сайту для модуля управления партнерами.

Каждому партнеру соответствует один или более контактов (телефон, электронный и физический адреса). Как следует из диаграммы, администратору системы предоставляется возможность просматривать список партнеров, создавать и редактировать партнеров, определять и модифицировать соответствующие контакты.

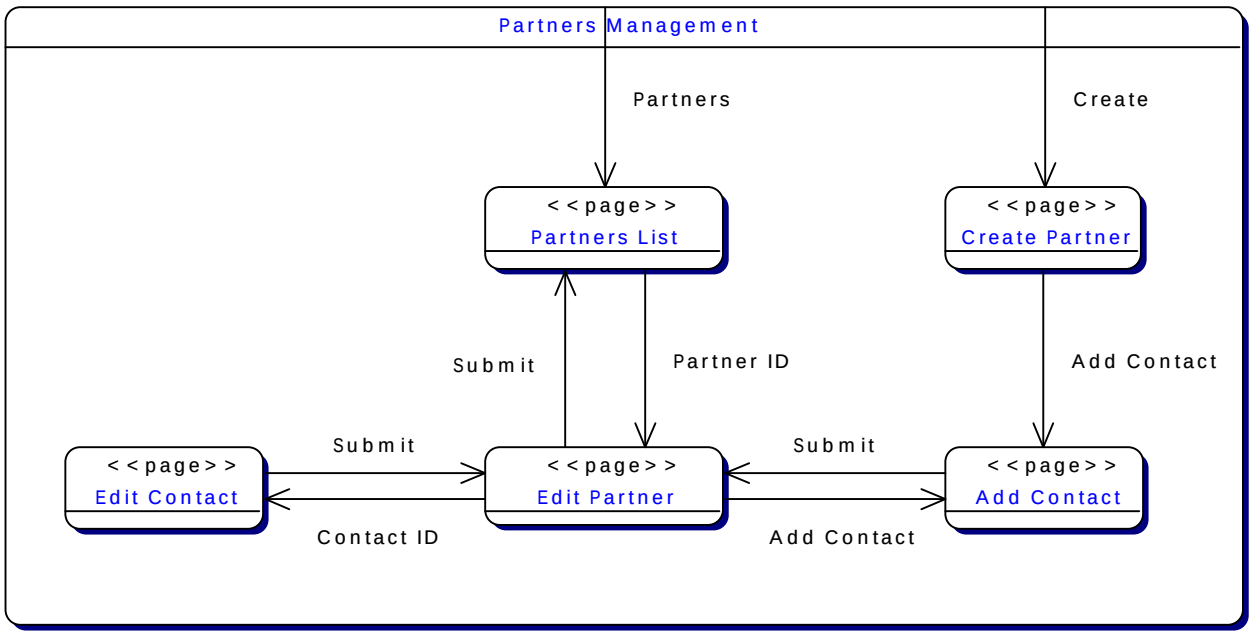

Рис. 1. Навигационная диаграмма управления партнерами

Рассмотрим теперь последовательность действий при добавлении нового контакта для партнера. На странице **«Edit Partner»** пользователь нажимает кнопку **«Add New Contact»** и переходит на страницу **«Add Contact»**. После заполнения формы, он нажимает кнопку **«Submit Contact»** и происходит обратный переход. На рис. 2 изображен фрагмент диаграммы состояний, соответствующий этой часть сценария.

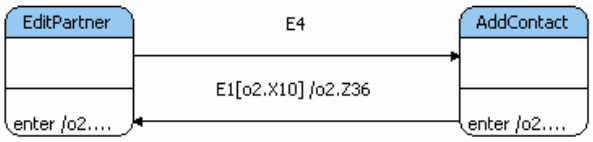

Рис. 2. Создание контакта

Рассмотрим второй (обратный) переход. Событие **E1** означает что пользователь, введя данные нового контакта, нажал кнопку **«Submit Contact»**. Входное воздействие **o2.X10** осуществляет проверку корректности введенных

пользователем данных. В случае успешной проверки, вызывается выходное воздействие **o2.Z36**, добавляющее запись о новом контакте партнера в базу данных. Таким образом, можно определить следующие входные и выходные данные для действий и события этого перехода (табл. 1).

Таблица 1. Пример спецификации событий и воздействий

Событие или действие	Описание	Входные данные	Выходные данные
E1	Нажата кнопка **«Submit Contact»**	–	Данные контакта (телефон, адреса)
o2.X10	Проверить корректность данных контакта	Данные контакта, идентификатор партнера	–
o2.Z36	Сохранить контакт	Данные контакта, идентификатор партнера	Код результата операции

С помощью набора подобных таблиц возможно описание операций в пределах автоматной модели. Назовем такие таблицы *таблицами спецификаций*.

Данные в этой модели сохраняются в контекстах одного из трех видов, различающихся временем жизни и условиями своего создания:

– *контекст приложения* (*application context*) создается при инициализации главного управляющего автомата и существует до момента завершения его работы;

– *пользовательский контекст* (*user context*) ассоциирован с сессией пользователя и существует до тех пор, пока пользователь работает с системой;

− *контекст события* (*event context*) существует только в течение одного перехода автомата между состояниями и используется для передачи данных между выходными воздействиями на переходах. Именно этот последний контекст целесообразно использовать для передачи данных контакта в рассмотренном только что примере.

Благодаря такой организации управления данными, путь их следования, то есть поток данных, совпадает с управляющими потоками, изображаемыми на диаграммах состояний. Следовательно, можно сказать, что таблица вида, изображенного выше, является простейшим способом **спецификации потоков данных «поверх» диаграмм состояний**.

1.5. Визуализация потоков данных

Визуализация потоков данных, определенных поверх диаграмм состояний, становится возможной с помощью приемов (визуальных элементов), вошедших в состав модели диаграммы деятельности *UML*. Диаграммы деятельности имеют много общего с диаграммами состояний. Это подтверждается тем фактом, что диаграммы деятельности в предыдущих версиях *UML* (до версии 2.0) рассматривались как частный случай диаграмм состояний.

Обыкновенно в диаграмму деятельности входят начальный и конечный узлы, операции и поток управления между ними. В таких случаях они напоминают блок-схемы, и единственным существенным отличием здесь является поддержка параллельных процессов. Однако, начиная с версии *UML* 2.0, потоки на диаграммах деятельности могут представлять не только управление, но и данные. Кроме того, в нотацию был включен элемент банка данных.

Изображение потоков данных на диаграмме состояний, в случае если она используется как средство визуального программирования, не обоснованно. Основной проблемой таких диаграмм обычно является их загруженность и добавление дополнительных (необязательных) деталей может только запутать программиста. Однако если использовать диаграммы состояний лишь для пояснения поведения системы, как часть документации, то добавление деталей иногда может повысить понимание системы. В таких случаях можно использовать элементы, показанные на рис. 3: контакты (*pins*), потоки данных, объекты данных, пред- и постусловия. Подробно обо все этих элементах диаграмм деятельности написано в работе [21].

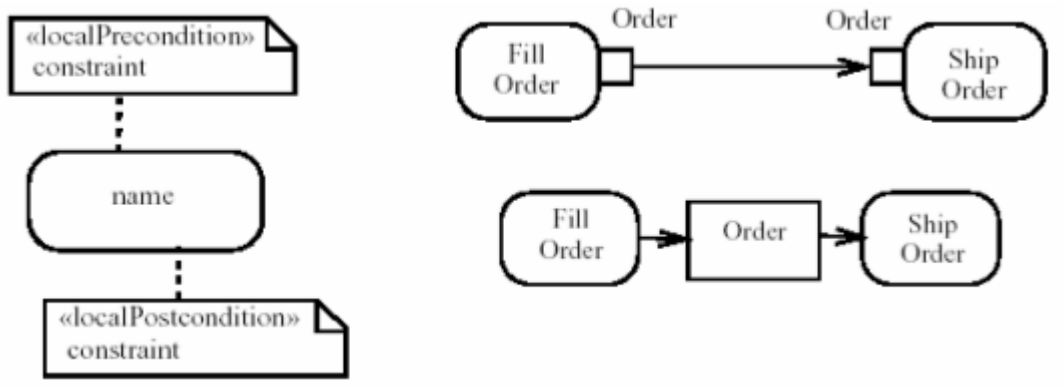

Рис. 3. Способы изображения потоков данных на диаграммах деятельности

Все эти способы можно применять и для диаграмм состояний, с той лишь разницей, что вершины диаграммы деятельности соответствуют не вершинам, но меткам на диаграммах состояний. В *UML* 2.0 для диаграмм состояний протокольных автоматов предлагается возможность подробного рассмотрения одного или нескольких переходов отдельно от диаграммы состояний, когда метка на переходе раскрывается в небольшую диаграмму деятельности, имеющую вместо начального и конечного узлов исходное и целевое состояния рассматриваемого перехода. В случае такой детализации переходов, изображение потоков данных производится согласно модели диаграммы деятельности.

1.6. Верификация потоков данных

Любой запрошенный элемент данных должен быть определен до запроса. Для автоматной модели это общее правило допускает конкретную формализацию, что дает возможность автоматической его проверки.

1.6.1. Пример

Для приведенного в разделе 1.4 примера можно утверждать следующее. «Входное воздействие **`o2.X10`** должно следовать за событием или воздействием, выходными данными которых являются данные контакта и идентификатор партнера». В пределах этого примера очевидным является утверждение, что, так как событие **`E1`** предшествует на диаграмме состояний вызову воздействия **`o2.X10`**, данные контакта присутствуют в контексте, однако идентификатор партнера (в пределах этого примера) не определен – сформулированное правило не удовлетворено.

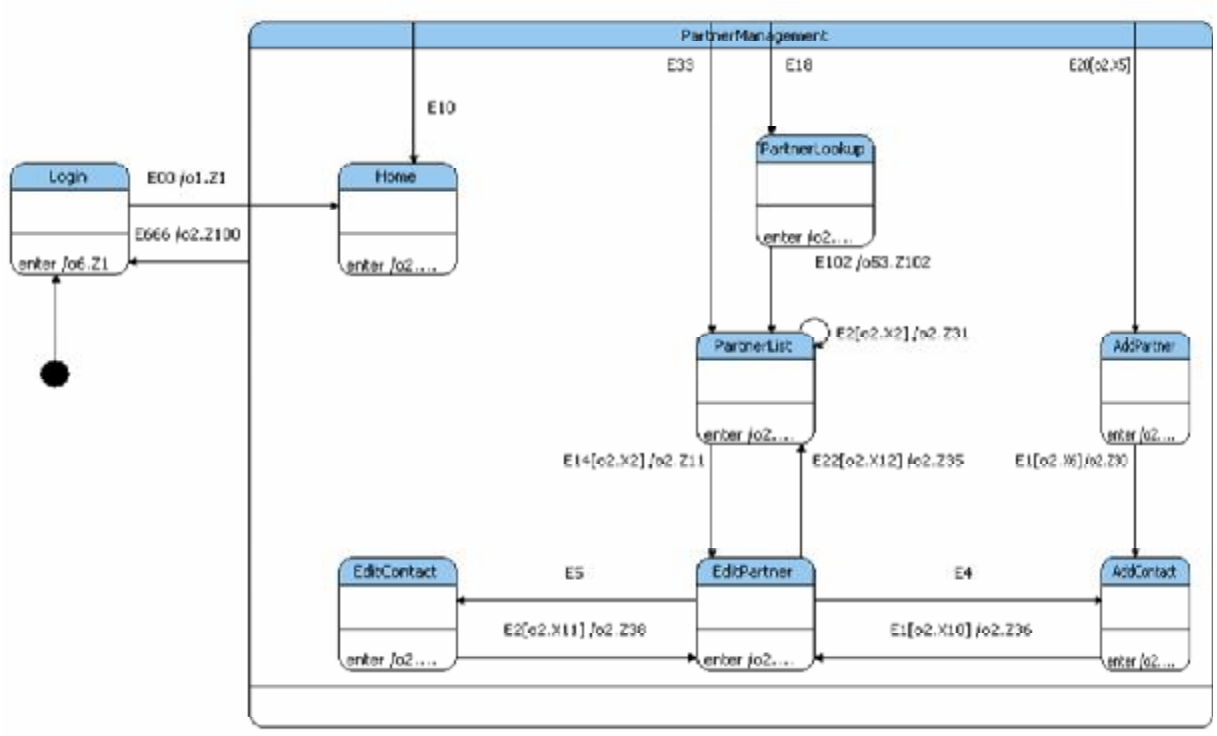

Рис. 4. Диаграмма состояний управления партнерами

Простота такой проверки основана в первую очередь на простоте рассмотренного примера (рис. 2), не отвечающего реальным задачам. Если рассмотреть полный вариант диаграммы состояний автомата, реализовывающего управление партнерами (рис. 4), то сформулированное правило оказывается удовлетворено, однако его проверка существенным образом усложнится. В данном случае информация о партнере (в том числе его идентификатор) определяется за один переход до состояния **«AddContact»** выходным воздействием **o2.Z11** (на переходе от состояния **«PartnerList»** к состоянию **«EditPartner»**).

Такое решение часто появляется в сценариях, требующих последовательного рассмотрения нескольких уровней детализации объекта. Например, расширяя рассматриваемый пример партнерскими программами, получим иерархию объектов, в которой программы содержат, помимо собственной информации, списки партнеров, партнеры, в свою очередь, содержат списки контактов. Сценарии работы с такими сущностями обычно предоставляет пользователю возможность просмотра информации с последовательной ее детализацией (рис. 5). При этом, например, на странице просмотра контакта будет необходимо отобразить не только собственные поля контакта, но и какую-то информацию о партнере и партнерской программе. Часто эта информация сохраняется в контексте (сессии) на «своем» уровне, за несколько переходов до рассматриваемого момента ее использования (в данном случае – до страницы контакта).

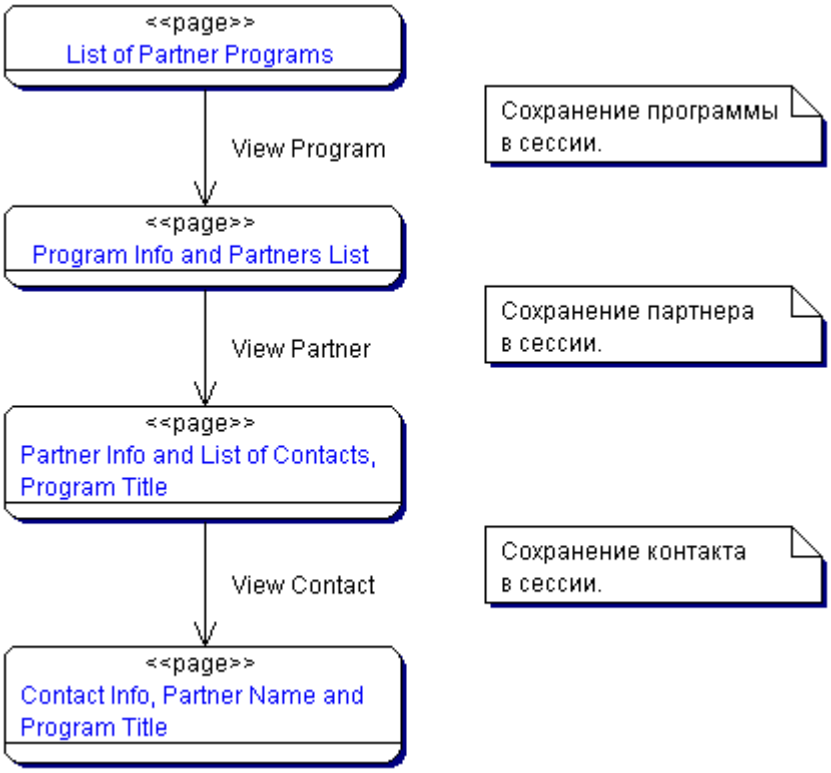

Рис. 5. Сценарий просмотра уровней детализации объекта

Эта схема существенно усложняется, если на «нижние» уровни возможно попадание несколькими путями. Из-за такого усложнения схемы и могут возникать ошибки и именно для таких схем более всего необходима автоматическая проверка потоков данных.

1.6.2. Формализация

Итак, сформулируем правило верификации потоков данных, определенных поверх диаграммы состояний. Для этого введем несколько обозначений.

Обозначим множество всех входных и выходных воздействий участвующих в работе автомата, как A, множество событий как E, а множество данных как D. Далее определим пару функционалов $In : A \to D$ и $Out : A \cup E \to D$, определяющих для каждого действия (или события) входные и выходные данные соответственно. Отметим, что для событий определены только выходные данные.

Правило верификации потоков данных формулируется следующим образом. Для каждого элемента данных $d \in D$ для каждого воздействия $a \in A$, такого, что $d \in In(a)$, и для каждого вызова воздействия a автоматом, должно выполняться следующее условие: на каждом пути от начального состояния диаграммы состояний до рассматриваемого места вызова действия a должно встречаться действие или событие $a' \in A \cup E$, такое, что $d \in Out(a')$.

Ясно, что автоматическая проверка данного правила требует выделения всех возможных последовательностей вызова воздействий и появления событий, что возможно благодаря наличию графа переходов. Описание алгоритма выделения этих последовательностей приведено во второй части данной работы.

1.7. Анализ кода и проверка соответствия спецификации

Ниже приведены шаблоны входных и выходных воздействий из схемы реализации пакета *UniMod*.

```
/* Выходное воздействие */
public void Z36(StateMachineContext smctx) {
    //TODO
}
/* Входное воздействие */
public int X10(StateMachineContext smctx) {
    //TODO
    return -1;
}
```

Оба типа воздействий получают единственным параметром контекст автомата – класс **`StateMachineContext`**. Этот класс предоставляет доступ ко всем трем контекстам данных посредством методов **`getApplicationContext()`**, **`getUserContext()`** и **`getEventContext()`**. Класс возвращаемого этими методами объекта должен расширять интерфейс **`Context`**:

```
public interface Context {

  /**
   * Assigns given data to given parameter name
   *
   * @param name parameter name
   * @param data parameter value
```

```
     */
     public void setParameter(String name, Object data);

     /**
      * Returns value of parameter
      *
      * @param name parameter name
      * @return parameter value
      */
     public Object getParameter(String name);

     /**
      * Returns all values of parameter with given name
      *
      * @param name parameter name
      * @return values array
      */
     public Object[] getParameterValues(String name);

     /**
      * Returns all parameters names in the data context
      *
      * @return all parameters names
      */
     public Enumeration getParameterNames();
}
```

С целью автоматического поиска ошибок программиста по описанному выше правилу в рамках данной работы реализован анализ кода воздействий с автоматическим определением списков входных и выходных данных. Ясно, что для этого необходимо выделить в коде метода, реализовывающего воздействие, обращения к контекстам данных (вызовы методов **`getParameter()`** и **`setParameter()`**).

Результат такого анализа предоставляет две возможности:

− проверку соответствия реализации спецификациям. Такая проверка позволяет избежать некорректной реализации модели (допустим, когда программист забыл добавить в контекст какой-либо элемент). Она особенно актуальна в случае, если объекты управления и логика приложения (автомат) реализовываются разными командами разработчиков;

− верификация потоков данных по описанному выше правилу с использованием автоматически определенных списков входных и выходных

данных. Часто нецелесообразно тратить время на спецификацию деталей (таких, как входные и выходные данные). Тогда можно основой для описанной в предыдущем разделе процедуры верификации взять результаты анализа кода. Ошибки спецификации сами по себе не имеют значения, если не вызывают ошибок реализации. Именно на поиске этих ошибок и стоит фокусировать процесс верификации. В сложных системах и разработчик, и тем более программист, может не учесть какой-либо из путей работы алгоритма, скажем, просто забыв положить какой-нибудь элемент в контекст, что особенно вероятно если этот элемент данных используется «далеко» от места своего определения. В последнем случае вероятной становится и возможность, что такую ошибку не выделят в процессе тестирования.

Именно на поиск таких ошибок и ориентирована данная работа. В следующей ее части описана реализация этого поиска, выполненная в рамках программного пакета *UniMod*.

2. РЕАЛИЗАЦИЯ

2.1. Описание

Описанные в предыдущем разделе модель определения и способ верификации потоков данных были реализованы в пакете *UniMod* версии 1.2.13.

Он предоставляет следующие возможности:

− определение входных и выходных данных (с указанием контекста) для каждого входного или выходного воздействия объектов управления;

− описание параметров (выходных данных – с точки зрения потоков данных) событий;

− анализ кода и автоматическое выделение входных и выходных данных для каждого входного и выходного воздействия;

− верификация потоков на диаграмме состояний. В качестве входных данных для алгоритма верификации могут выступать как спецификация воздействий, так и результаты анализа кода. Результатом верификации служит набор путей, на которых запрашиваемые данные не производятся.

Заметим, что далее в данной работе входные данные часто называются запрашиваемыми, а выходные данные – производимыми или определяемыми. Будем говорить, что набор воздействий *производит* некоторый элемент данный, если в этом наборе есть хотя бы одно воздействие, выходные данные которого содержат указанный элемент.

2.2. Спецификация

Спецификация воздействий и событий в пакете *UniMod* была реализована в виде следующих (табл. 2) тегов *Javadoc*, заданных в области комментариев метода (в случае воздействий) или константы (в случае события).

Таблица 2. Тэги спецификации событий и воздействий

Тэг	Описание	Форма
@unimod.action.descr[1]	Описание воздействия	Текст (в общем случае многострочный) на естественном языке.
@unimod.event.descr[1]	Описание события	
@unimod.application.in	Входные данные воздействия контекста приложения	Параметры через пробел, запятую или другие знаки препинания. Для определения сложных идентификаторов данных (например, содержащих пробелы) используются двойные кавычки. Для определения двойных кавычек в идентификаторе используется замена их на соответствующую escape-последовательность. В случае определения нескольких таких тэгов, их значения объединяются.
@unimod.application.out	Выходные данные воздействия контекста приложения	
@unimod.user.in	Входные данные воздействия контекста пользователя	
@unimod.user.out	Выходные данные воздействия контекста пользователя	
@unimod.event.in	Входные данные воздействия контекста события	
@unimod.event.out	Выходные данные воздействия контекста события	
@unimod.event.parameter	Параметры события	

[1] Поддержка тэгов `@unimod.action.descr` и `@unimod.event.descr` присутствовала и в более ранних версиях *UniMod*.

Ниже приведен пример спецификации выходного воздействия **o2.Z36** из рассмотренного в предыдущей части примера (раздел 1.6.1):

```
/**
 * @unimod.action.descr Save new contact to the database
 * @unimod.event.in CONTACT, PARTNER
 * @unimod.user.out “ERROR CODE”
 */
public void Z36(StateMachineContext smctx) {
    //…
}
```

Пример спецификации события выглядит следующим образом:

```
/**
 * @unimod.event.descr New contact submition
 * @unimod.event.parameter CONTACT
 */
public static final String E1 = "E1";
```

2.3. Анализ кода

Анализ кода воздействий, использованных автоматом, производится по следующей схеме:

– с помощью схемы связей модели определить список объектов управления, связанных с исследуемым автоматом (выделить классы, реализовывающие эти объекты и идентификаторы, с помощью которых автомат обращается к этим объектам);

– для тех объектов управления, для которых определен класс реализации, в случае если этот класс присутствует в проекте, провести начальный анализ кода с использованием дерева *AST* (*Abstract Syntax Tree*). При начальном анализе требуется выделить в этом дереве вершины, соответствующие методам, реализующим входные и выходные воздействия;

– для каждого метода, выделить идентификатор, под которым ему передается (единственным параметром) контекст конечного автомата (**StateMachineContext**). После этого провести исследование кода тела метода. Идентификатор контекста автомата необходим для того, чтобы выделить в теле

метода обращения к контекстам данных, полученным через контекст автомата. Результатом этого исследования должны стать списки входных и выходных данных воздействия.

2.3.1. Первоначальное исследование кода

Необходимо пояснить, что платформа *Eclipse*, на которой базируется современная версия пакета *UniMod*, предоставляет разработчику своих расширений (*plugins*) возможность исследования *Java*-кода, преобразованного в дерево *AST*. Такое исследование проводится с помощью наследников класса **`ASTVisitor`**, предоставляющего пару методов **`visit()`** и **`endVisit()`** для каждого типа вершины дерева *AST*.

Согласно этой схеме, первоначальное исследование кода класса, реализовывающего объект управления, осуществляется с помощью вызова метода **`td.accept(visitor)`**, где **`td`** – вершина типа **`TypeDeclaration`** (соответствует исследуемому классу) дерева *AST*, а **`visitor`** – объект подкласса **`ASTVisitor`**, переопределяющего методы **`visit(MethodDeclaration node)`** и **`endVisit(MethodDeclaration node)`**. В процессе выполнения метода **`td.accept`** эти методы будут вызываться в начале и конце обработки вершины дерева, соответствующей описанию метода. В случае если метод имеет один параметр типа **`StateMachineContext`**, предполагается что этот метод описывает воздействие. Тогда вершина **`node`** добавляется в формируемый список воздействий объекта управления.

Дополнительно на этом этапе (при обработке каждой вершины **`MethodDeclaration`**) определяется идентификатор параметра, под которым контекст автомата передается методу. Этот идентификатор понадобится на следующем шаге, цель которого состоит в том, чтобы, исследовав код метода, выделить обращения к контекстам данных.

2.3.2. Выделение обращений к контекстам данных

Анализ кода тела метода, реализовывающего входное или выходное воздействие, получает в качестве входных данных вершину дерева *AST*, соответствующую телу метода, а также идентификатор параметра – контекста автомата. Рассмотрим следующий пример выходного воздействия:

```
public void Z36(StateMachineContext smctx) {
  Context c = smctx.getUserContext();                                    (1)
  Contact contact =
            (Contact)smctx.getEventContext().getParameter("CONTACT");    (2)
  Partner partner = (Partner)c.getParameter("PARTNER");                  (3)

  // ...

  c.setParameter("ERROR CODE", ErrorCodes.OK);                           (4)
  // smctx.getUserContext().setParameter("ERROR CODE", ErrorCodes.OK);   (5)
}
```

Анализ кода выделяет в теле метода вызовы (вершины дерева *AST* типа **`MethodInvocation`**) трех видов:

- **`get*Context`** (**`getApplicationContext`**, **`getUserContext`** и **`getEventContext`**) – такой вызов либо определяет переменную контекста (1), либо является базой для вызова методов **`getParameter`** (2) или **`setParameter`** (5). Ясно, что для выделения этих методов и используется полученный на вход этого шага идентификатор параметра (в данном случае – **`smctx`**);

- **`getParameter`** – запрос параметра либо с использованием переменной контекста (3), либо с вложенным вызовом **`get*Context`** (2);

- **`setParameter`** – определение параметра. Здесь, как и в предыдущем пункте, возможны два варианта указания контекста (4, 5).

На рис. 6 и 7 изображены схема связей и диаграмма состояний автомата, реализовывающего поиск входных и выходных данных метода.

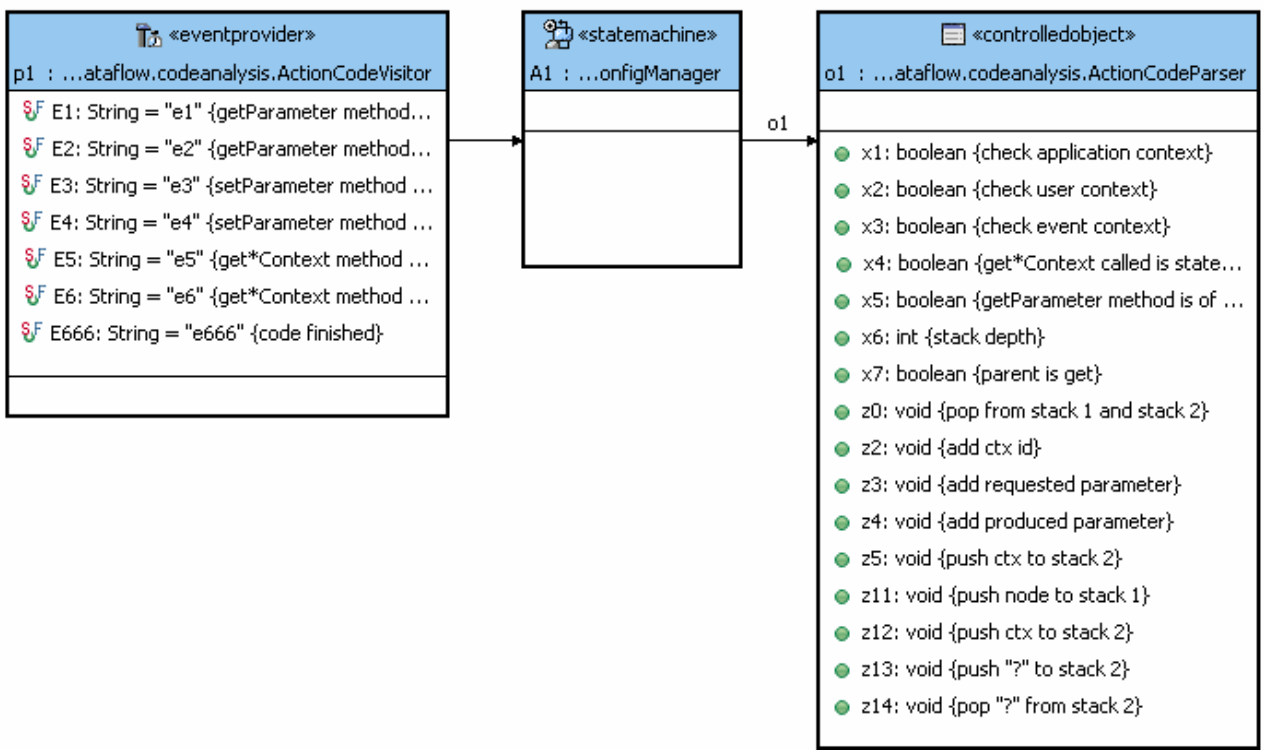

Рис. 6. Схема связей автомата, реализующего анализ кода

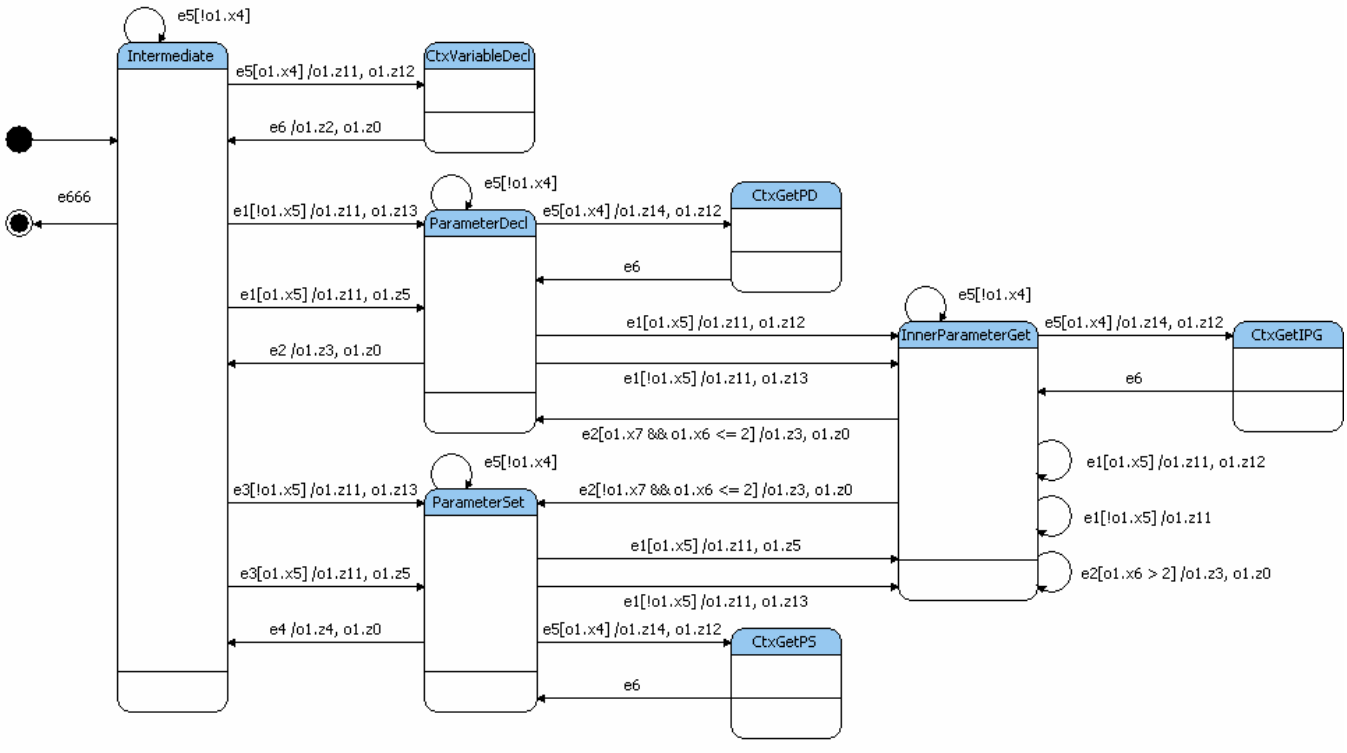

Рис. 7. Диаграмма состояний автомата, реализующего анализ кода

События в системе анализа кода вызываются подклассом класса **ASTVisitor** в начале и конце обработки вершин **MethodInvocation** трех описанных выше типов. Подробное описание работы автомата здесь не приводится. Заметим для примера, что на этапе исследования строки (2) предыдущего примера автомат последовательно пройдет следующий ряд состояний:

- **Intermediate** (промежуточное состояние);
- **ParameterDecl** (исследование вызова метода **getParameter()**);
- **CtxGetPD** (вложенное получение контекста данных от контекста автомата вызовом **getEventContext()**);
- **ParameterDecl**;
- **Intermediate**.

Пояснения требует разве что правая часть диаграммы состояний, а именно, состояние **InnerParameterGet**. Добавление этого состояния обусловлено возможностью рекурсивно вложенного вызова метода **getParameter**, например, в таком виде:

```
c.setParameter(“CONTACT”, smctx.getEventContext().getParameter(“CONTACT));
```

Возможным недостатком такого алгоритма является пропуск всех остальных элементов кода (например, условных переходов), за исключением вызова методов. Прямая обработка условий на условных переходах до момента компиляции программы слабо обоснована, в силу чего такая обработка и не была реализована. Однако выделение изначально конкурирующих групп входных и выходных данных может повысить полноту проверки потоков данных. Например, для следующего кода воздействия:

```
if (proceededOk) {
  c.setParameter(“CONTACT”, contact);
} else {
```

```
    c.setParameter("ERROR", ErrorCodes.ERROR);
  }
```

можно определить два конкурирующих элемента выходных данных **CONTACT** и **ERROR**. Очевидно, что алгоритм автоматической выборки элементов входных и выходных данных, а также дальнейшая проверка (анализ диаграммы состояний и поиск «плохих» путей) при выделении конкурирующих групп данных существенно усложнится. В силу этого конкурирующие группы не были реализованы в рамках данной работы.

2.3.3. Получение идентификатора элемента данных

Когда алгоритм анализа кода обрабатывает вызовы методов **getParameter()** или **setParameter()**, возникает необходимость определения имени, под которым параметр запрашивается или кладется в контекст. Для описанного выше примера задача решается достаточно просто, так как все имена в нем были заданы строковыми литералами. На практике же чаще всего для сохранения параметров, например, в сессии, используются статические константы, определенные иногда в отдельных (специально для этого предназначенных) файлах.

Алгоритм получения имени запрошенного элемента данных был реализован по следующей схеме.

```
функция получитьЗначениеКонстанты (параметрМетода) {
  если параметрМетода простой строковой литерал, вернуть его;
  если параметрМетода есть сумма выражений пi,
    вернуть сумму результатов функции получитьЗначениеКонстанты(пi);
  иначе искать в пределах текущего проекта статическую константу с именем
  параметрМетода {
    для этого, если парамертМетода не содержит точек,
      проверить текущий класс,
    если содержит,
      проверить классы и пакеты, подключенные (include) к текущему
      файлу, а также текущий пакет;
  }
```

```
}
```

При поиске выделяются только поля классов, определенные с модификаторами **`static`** и **`final`** и имеющие примитивный или строковый тип. Такие ограничения обоснованы тем, что анализ кода работает до момента компиляции, а следовательно, изначально не рассчитан на обработку непостоянных элементов. В то же время, такая реализация соответствует общепринятому стилю программирования, согласно которому элементы хранятся в контекстах данных под именами, обычно определяемыми как статические константы.

2.4. Верификация потоков данных

Входными данными для алгоритма верификации потоков является хэш-карта (**`HashMap`**), в которой ключами служат идентификаторы воздействий (например, **`o2.Z36`**), а значениями – объекты **`DFAction`**, содержащие по четыре списка входных и выходных данных. Три списка соответствуют трем контекстам данных, четвертый список соответствует входным и выходным данным, определенным спецификацией (тэгами *Javadoc*).

В целях облегчения объяснений, приведем описание процесса верификации только для данных контекста приложения (далее – просто *контекста*). Процесс верификации данных контекста пользователя проводится точно по такой же схеме. Что касается контекста события, то его верификация является подзадачей задачи верификации контекста приложения, так как интервал жизни приложения включает в себя интервал жизни контекста события. В свою очередь, списки данных, определенные спецификацией, не отличаются от списков данных, полученных с помощью анализа кода, поэтому отдельное их рассмотрение также не имеет смысла.

Для реализации проверки определенного в первой части работы верификационного правила требуется выделить все возможные последовательности вызовов воздействий и появления событий. Будем использовать диаграмму специального вида для определения всех таких последовательностей. Назовем ее *диаграммой деятельности*. Каждой вершине такой диаграммы соответствует один вызов воздействия или появление события на диаграмме состояний. И пусть вершины диаграммы деятельности соединяются таким образом, чтобы пути в ней соответствовали последовательности вызова воздействий и появления событий на диаграмме состояний. Пример диаграммы деятельности и соответствующей ей диаграммы состояний изображен на рис. 8.

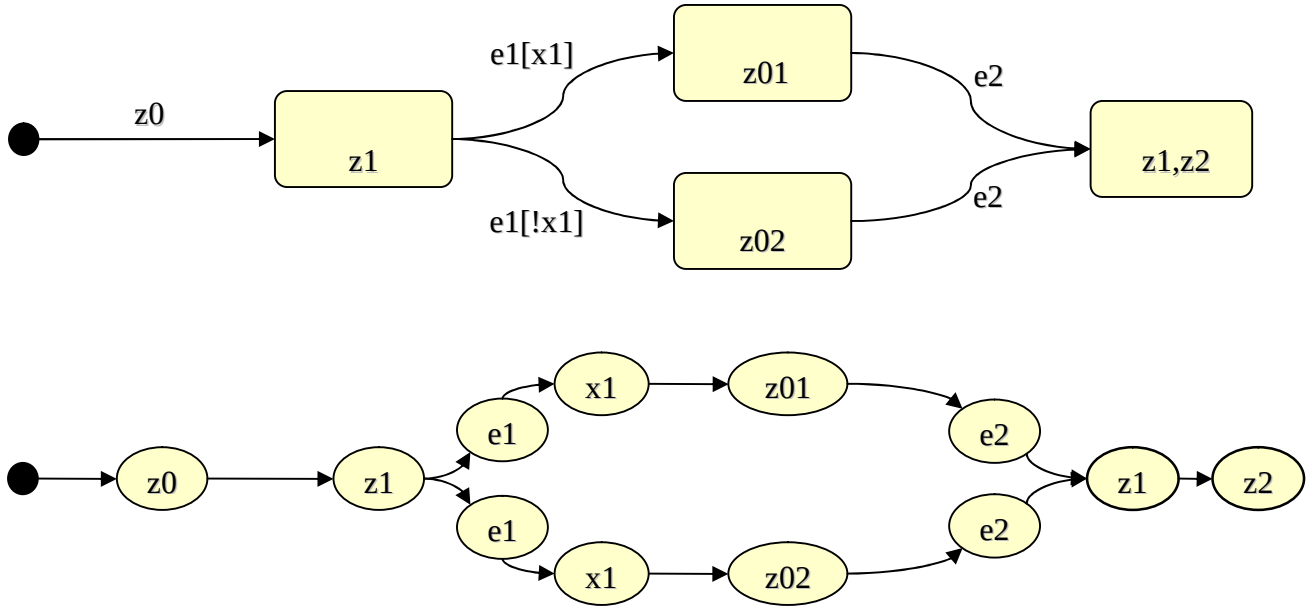

Рис. 8. Пример диаграммы состояний (вверху) и соответствующей ей диаграммы деятельности (внизу)

Верификация потоков данных с помощью такой диаграммы производится в два этапа:

- построение диаграммы деятельности по диаграмме состояний;

– анализ диаграммы деятельности и поиск «плохих» путей.

Опишем эти этапы.

2.5. Построение диаграммы деятельности

Диаграмма классов, представленная на рис. 9, описывает основные использованные для представления диаграммы деятельности структуры и взаимосвязь между ними. Как видно из рисунка, вершине (**`Activity`**) диаграммы деятельности (**`ActivityChart`**) соответствует описание входных и выходных значений соответствующего действия или события (**`DFAction`**) и элемент автоматной модели (**`ModelElement`**) – переход или состояние, на котором вызывается соответствующее действие или обрабатывается событие.

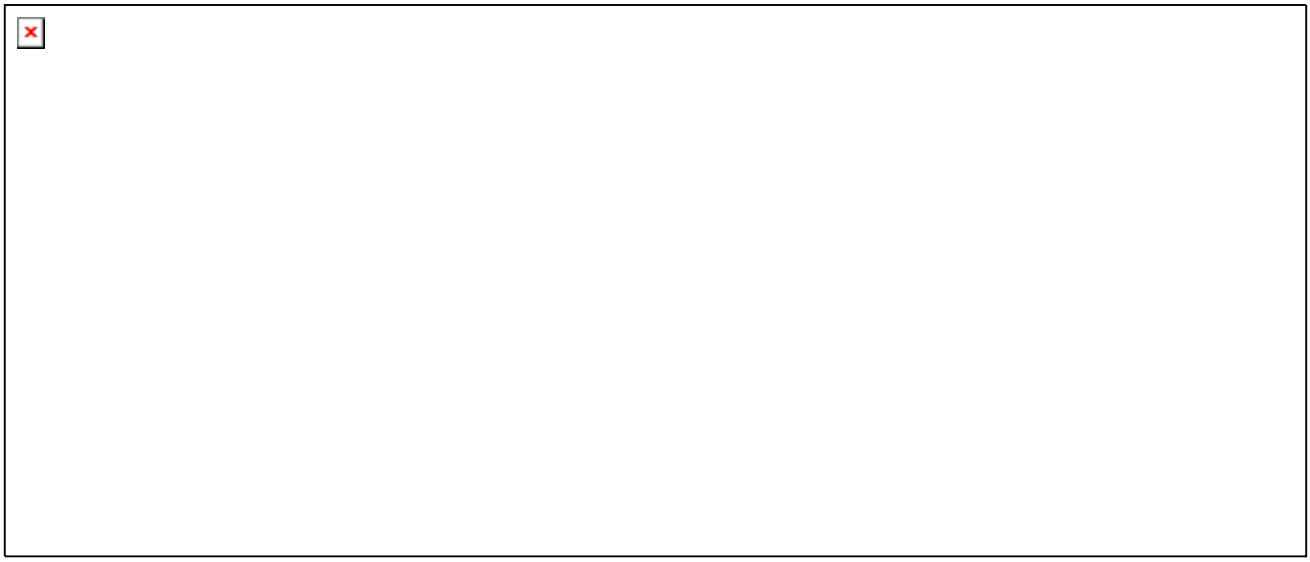

Рис. 9. Диаграмма классов реализации диаграммы деятельности

Рассмотрим работу алгоритма построения диаграммы деятельности на примере элементарного Интернет-приложения *WebChat*, описывающегося одним автоматом, диаграмма переходов которого изображена на рис. 10.

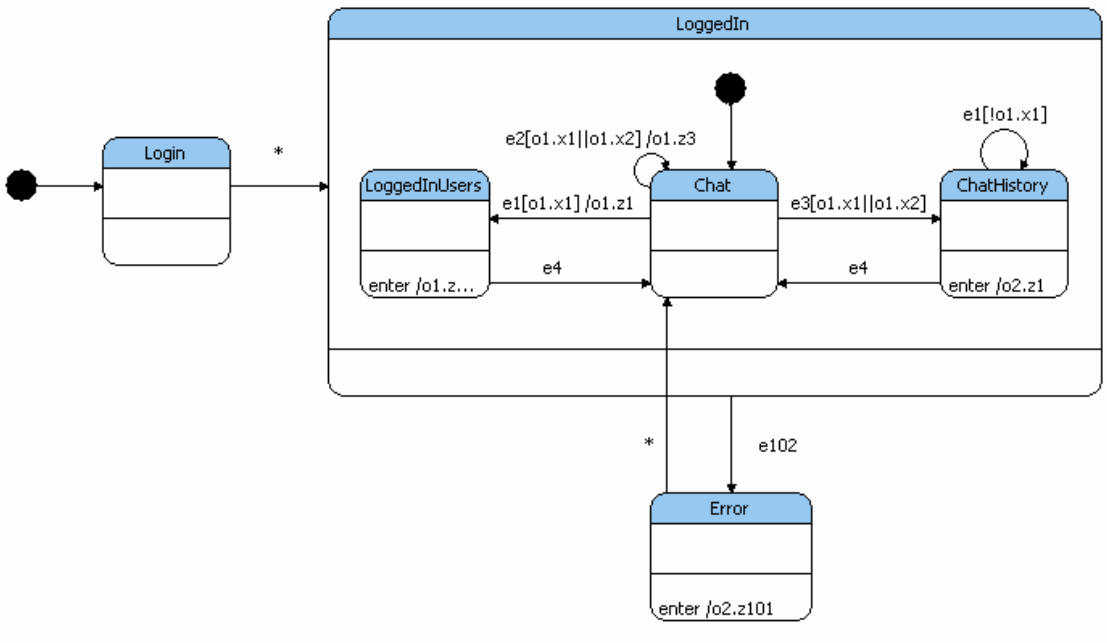

Рис. 10. Диаграмма состояний автомата *WebChat*

2.5.1. Шаг первый

Первым шагом построения диаграммы деятельности является выделение промежуточной диаграммы, являющейся почти изоморфной копией диаграммы состояний. Назовем ее *псевдо диаграммой деятельности*. Для этого каждому элементу автоматной модели – переходу или состоянию – сопоставляется вершина в диаграмме деятельности. Результат этого шага для рассматриваемого примера *WebChat* изображен рис. 11, где голубым цветом отмечены вершины, соответствующие состояниям автоматной модели, а белым – вершины, соответствующие переходам.

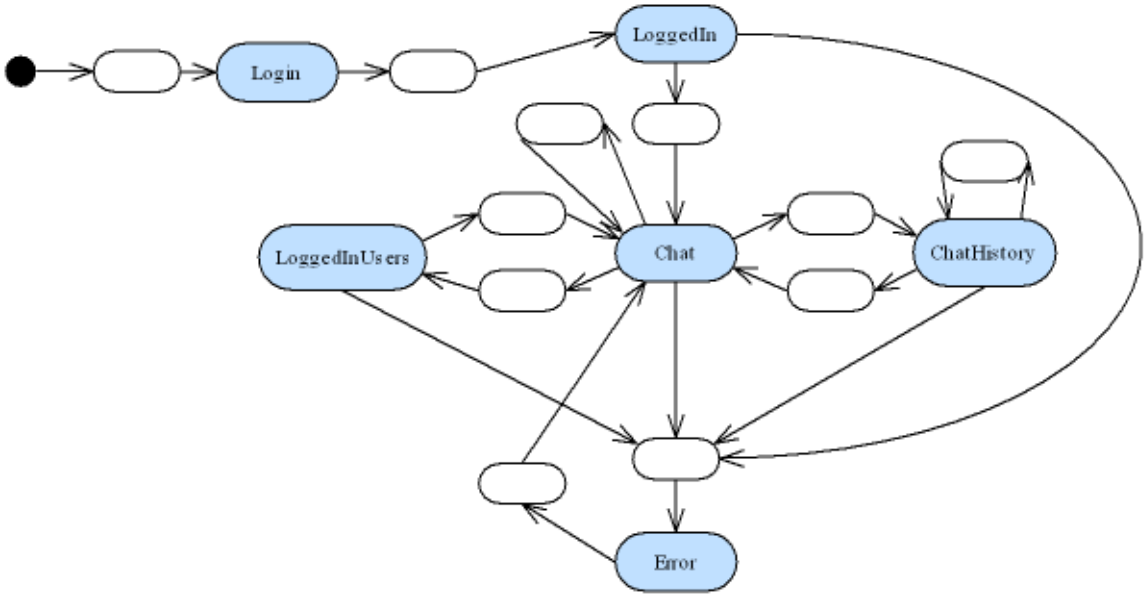

Рис. 11. Результат первого шага алгоритма построения диаграммы деятельности

Построение диаграммы деятельности осуществляется обходом диаграммы состояний [30, 31]. Отличием данной его реализации является определение переходов как отдельного вида вершин, являющихся объектами обхода наравне с состояниями. Метод выделения элементов, следующих за текущим элементом, использует такие правила:

− если текущий элемент является состоянием, добавить в формируемый список следующих элементов все исходящие из него переходы;

− если текущий элемент является составным состоянием, добавить в формируемый список все переходы, исходящие из начального состояния, вложенного в текущее;

− если текущий элемент имеет суперсостояние, добавить все исходящие из него переходы в формируемый список;

− если текущий элемент является переходом, а его целевое состояние является конечным и имеет суперсостояние, то вернуть все элементы, следующие за суперсостоянием (рекурсивный вызов этого же метода);

− если текущий элемент является переходом, а его целевое состояние является конечным, но не вложенным, то ничего не возвращать и добавить целевое состояние в список конечных состояний;

− если текущий элемент является переходом и его целевое состояние не является конечным состоянием, добавить в формируемый список целевое состояние.

Обратим внимание, что предложенный алгоритм не обрабатывает вызовы вложенных автоматов. Их обработка существенно усложнит алгоритм и уменьшит скорость его работы в силу возможности рекурсивного вызова автоматов с возможностью выхода из вложенного автомата в любой момент (из любого состояния). При этом ясно, что использование общих переменных в разных автоматах редко бывает оправдано. Таким образом, в рамках данного исследования каждый автомат в системе рассматривается отдельно, а определения и верификация общих для нескольких автоматов данных остается на совести и в зоне ответственности программиста.

Также заметим, что в процессе обработки диаграммы состояний был в отдельный список выделен набор вершин диаграммы деятельности, соответствующих конечным (не вложенным) состояниям автоматной модели. В завершение построения диаграммы деятельности, к ней добавляется переходы от всех этих выделенных вершин в общую конечную вершину. Таким образом, полученная диаграмма деятельности обладает только одной начальной (в силу свойств диаграммы состояний) и одной конечной вершиной, что упрощает работу с такой диаграммой.

2.5.2. Шаг второй

Следующим шагом является удаление из полученной псевдо диаграммы деятельности *тривиальных* вершин – таких, с которыми ассоциирован элемент

модели, не содержащий вызовов входных или выходных воздействий и событий (например, переход по произвольному событию с охранным условием **`true`** и без выходных воздействий или состояние без действий на входе).

Этот шаг призван облегчить дальнейшее построение диаграммы деятельности.

Далее при описании алгоритмов модификации диаграммы деятельности будем для краткости называть вершину **`a1`** *предшествующей* вершине **`a2`** если в диаграмму входит ребро исходящее из **`a1`** и входящее в **`a2`**. Аналогичным образом определим понятие вершины, *следующей* за другой вершиной. Благодаря этим определениям алгоритм удаления тривиальной вершины легко описывается в два предложения.

Алгоритм линейно перебирает все вершины диаграммы. Каждая тривиальная вершина удаляется, а все предшествующие ей вершины попарно связываются со всеми следующими за ней вершинами.

2.5.3. Шаг третий

Третьим и последним шагом при построении диаграммы деятельности является разбиение вершин полученной псевдо диаграммы деятельности на группы, в которых вершинам соответствуют описатели воздействий **`DFAction`** (полученные из хеш-карты, являющейся результатом обработки кода). Результатом этой операции будет законченная диаграмма, отображающая последовательность вызова воздействий и появления событий.

Разбиение вершины, ассоциированной с состоянием автоматной модели, изображено на рис. 12. Каждому воздействию, вызываемому при входе в состояние, сопоставляется вершина диаграммы деятельности. Эти вершины соединяются в порядке вызова воздействий, все входящие в разбиваемую

вершину ребра перенаправляются на первую вершину в полученной цепочке, начала всех исходящих из разбиваемой вершины ребер устанавливаются на последнюю в цепочке.

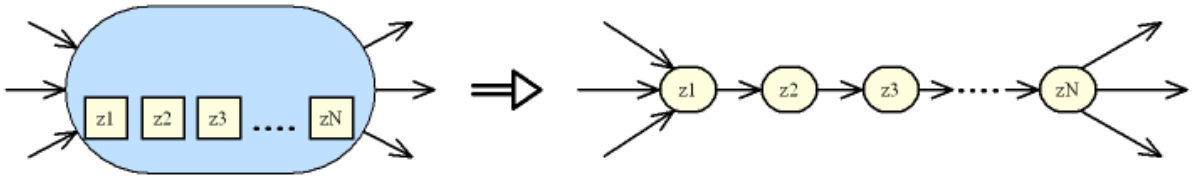

Рис. 12. Схема разбиения состояния

Для переходов схема немного усложняется за счет охранных условий (примерная схема изображена на рис. 13). Для получения нескольких возможных путей прохода требуется преобразовать охранное условие в форму дизъюнкции конъюнкций (дизъюнктивную нормальную форму). Подробно о способе обработки охранных условий в *UniMod* и, в частности, выделении дизъюнктивных форм, рассказано в работе [32].

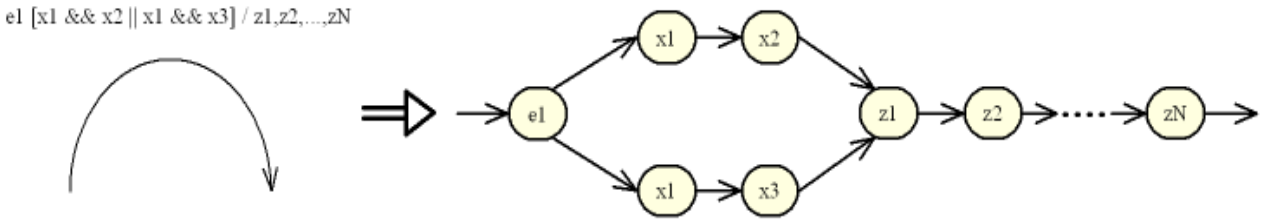

Рис. 13. Схема разбиения состояния

Как следует из рисунка, конъюнкты каждого дизъюнкта составляются в альтернативные цепочки, начало их присоединяется к событию, а конец – к цепочке выходных воздействий.

Следует обратить внимание, что некоторые элементы этой схемы могут быть не определены. Например, для перехода по произвольному событию, вершина **e1** будет исключена из результирующей схемы, а следовательно, полученное

разбиение вершины будет иметь несколько входов (один вход для каждого дизъюнкта). Аналогично, в случае отсутствия выходных воздействий результирующая схема будет иметь несколько выходов. Отметим здесь, что благодаря проведению предыдущего шага (удалению тривиальных вершин), получение пустой заменяющей схемы невозможно.

Используя две описанные схемы (для состояния и для перехода), получаем метод, который сопоставляет вершине псевдо диаграммы деятельности некоторую схему, имеющую (в общем случае) конечный набор входов и выходов. Заменим каждую вершину диаграммы деятельности такой схемой. При этом все вершины, следующие за заменяемой вершиной, свяжем попарно со всеми выходами схемы, а все вершины, предшествующие заменяемой – со всеми входами схемы.

Отметим, что параллельно с развертыванием вершин (построением заменяющей схемы) производится сопоставление новым вершинам описателей **`DFAction`**, содержащих списки входящих и выходящих данных. Таким образом, сформированная после осуществления этого шага структура (диаграмма деятельности) обладает всеми необходимыми для поиска плохих путей данными. Она показывает последовательность, в которой вызываются воздействия и появляются события при работе автомата, а также содержит информацию о входных и выходных данных для каждого воздействия или события.

2.6. Поиск «плохих» путей

Верификация полученной диаграммы деятельности, также как ее построение, основан на алгоритме обхода в ширину. Использованная для решения этой задачи модификация алгоритма предусматривает проход по обращенным ребрам от всех вершин, в которых запрашивается проверяемый элемент данных. Опишем этот шаг алгоритма подробнее.

В процессе получения диаграммы деятельности дополнительно была сформирована хеш-карта, ключами в которой служат идентификаторы элементов данных, а значениями – списки вершин диаграммы деятельности, в которых запрашивается соответствующий элемент.

Для старта обхода в ширину в качестве списка начальных вершин (вершин, которые опускаются в очередь перед началом цикла обхода) берется один список из этой хеш-карты, соответствующий рассматриваемому элементу данных. Алгоритм обхода считает изначально покрашенной любую вершину, которая производит рассматриваемый элемент. Используется модификация алгоритма обхода, запоминающая пути [31]. В случае достижения таким обходом начальной вершины, путь, приведший к ней, минует все вершины, производящие исследуемый элемент данных, а следовательно, такой путь является «плохим».

2.7. Выводы

В предыдущих разделах был описан алгоритм верификации потоков данных, основанный на решениях трех основных подзадач:

− анализ кода воздействий с автоматическим выделением входных и выходных данных;

− построение диаграммы деятельности по диаграмме состояний, определяющей все возможные последовательности вызовов воздействий и появления событий;

− анализ диаграммы деятельности и поиск «плохих» путей, на которых не производятся какие-либо запрашиваемые данные.

2.7.1. Анализ кода воздействий

Скорость работы алгоритма анализа кода зависит от реализации использованных в нем стандартных предоставляемых платформой *Eclipse*

методов, а также от размеров частей проекта. Линейный обход дерева, описывающего код метода, умножается на поиск значений констант. Верхняя оценка трудоемкости алгоритма, таким образом, может быть определена в виде $O(M*N*T)$, где M – максимальный размер метода, реализовывающего воздействие (количество вершин в дереве), N – число классов, импортируемых в файл, в котором реализован объект управления, T – максимальный размер импортируемого класса.

2.7.2. Построение диаграммы деятельности

Алгоритм построения диаграммы деятельности выполняется в три шага. Выделение плоской копии диаграммы состояний (первый шаг) осуществляется модифицированным обходом в ширину с трудоемкостью $O(V+E)$, где V – количество состояний в диаграмме, а E – количество переходов.

Удаление тривиальных вершин (второй шаг) выполняется за V' операций (количество вершин диаграммы деятельности), приблизительно равной $O(V+E)$. В действительности удаление каждой вершины требует $O(X*Y)$ операций, где X – количество входящих в вершину ребер, а Y – количество исходящих ребер. Отметим здесь, что графы, представляющие диаграммы состояний в рассматриваемой задаче, обычно достаточно разрежены. Иначе ясная визуализация диаграмм будет невозможна. И даже несмотря на то, что соотношение числа ребер к числу вершин в диаграмме деятельности обычно больше чем в диаграмме состояний (в том числе и за счет удаления тривиальных вершин), величиной $O(X*Y)$ можно пренебречь.

Третий шаг построения диаграммы деятельности – разбиение вершин – имеет трудоемкость $O(A)$, где A равно суммарному количеству вызовов воздействий и появления событий на диаграмме состояний.

Таким образом, алгоритм построения диаграммы деятельности имеет линейную трудоемкость *O(V+E+A)*.

Правильность предложенного алгоритма обуславливается тем, что он основан на правилах интерпретации диаграммы состояний. А именно, построение диаграммы деятельности реализовано согласно семантике диаграммы состояний, задающей все возможные пути, по которым может следовать поток управления.

2.7.3. Поиск «плохих» путей

Поиск «плохих» путей организован согласно следующему утверждению. Если удалить из диаграммы деятельности все вершины, производящие некоторый элемент данных, то не должно существовать пути от начальной вершины до любого элемента, запрашивающего исследуемый элемент данных.

Алгоритм поиска путей основан на линейном переборе всех возможных идентификаторов данных в системе (в исследуемом контексте). Обозначим их число как *D*. Далее, для каждого идентификатора, производится обход в ширину от вершин, запрашивающих рассматриваемый элемент данных по обращенным ребрам до начальной вершины. Трудоемкость такого алгоритма, таким образом, равна *O(D*V'')*, где *V''* – количество вершин в окончательном варианте диаграммы деятельности, что приблизительно равно *A*. Окончательный вариант оценки трудоемкости – *O(D*A)*.

Правильность алгоритма является следствием корректности алгоритма обхода в ширину, а также фактом, что такой обход находит все пути от исходной вершины до целевой. В данном случае есть набор исходных вершин – тех, которые запрашивают рассматриваемый элемента данных, и одна целевая вершина – ассоциированная с начальной вершиной диаграммы состояний автомата. Учитывая, что использованная модификация обхода продвигает поиск по обращенным ребрам, а также то, что она не проходит вершины, производящие

исследуемый элемент данных, такой обход действительно возвращает все «плохие» пути для рассматриваемого элемента данных.

2.7.4. Замечание

Заметим, что в алгоритме верификации события рассматриваются как операции, для которых определены только выходные данные. И верификация потоков данных контекста события производится по схеме верификации данных остальных контекстов (отличается на практике только выдача результатов – показываются не «плохие» пути, а «плохие» переходы). Такой подход имеет право на существование, однако обладает очевидной альтернативой. Так как параметры события всегда принадлежат контексту события, и так как время жизни этого контекста совпадает с одним переходом, его рассмотрение можно было бы провести отдельно от верификации данных остальных двух контекстов. Такая задача решалась бы значительно проще, и, кроме того, структура диаграммы деятельности, необходимой для верификации остальных контекстов, также опростилась бы (хотя и незначительно) за счет исключения из нее событий. Причиной, по которой такое решение не было описано, является стремление унифицировать задачу и показать в данной работе общее ее решение.

3. ПРИМЕР

Продемонстрируем применение разработанного в данной работе метода на примере простого приложения *HelloUser*. Диаграмма состояний и схема связей автомата **A1** из этого приложения изображены на рис. 14 и 15.

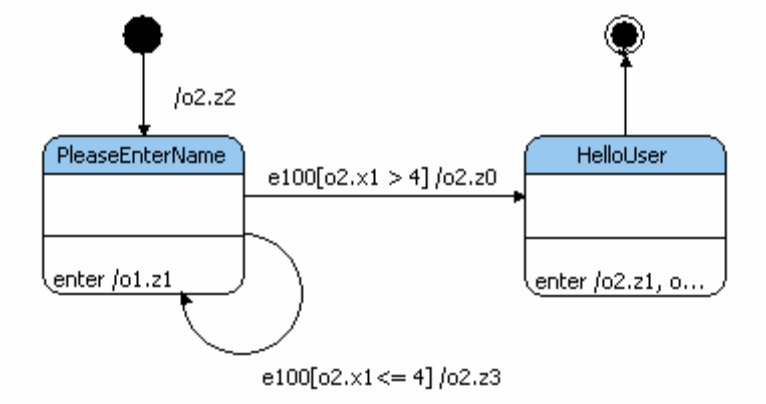

Рис. 14. Диаграмма состояний автомата **A1** из приложения *HelloUser*

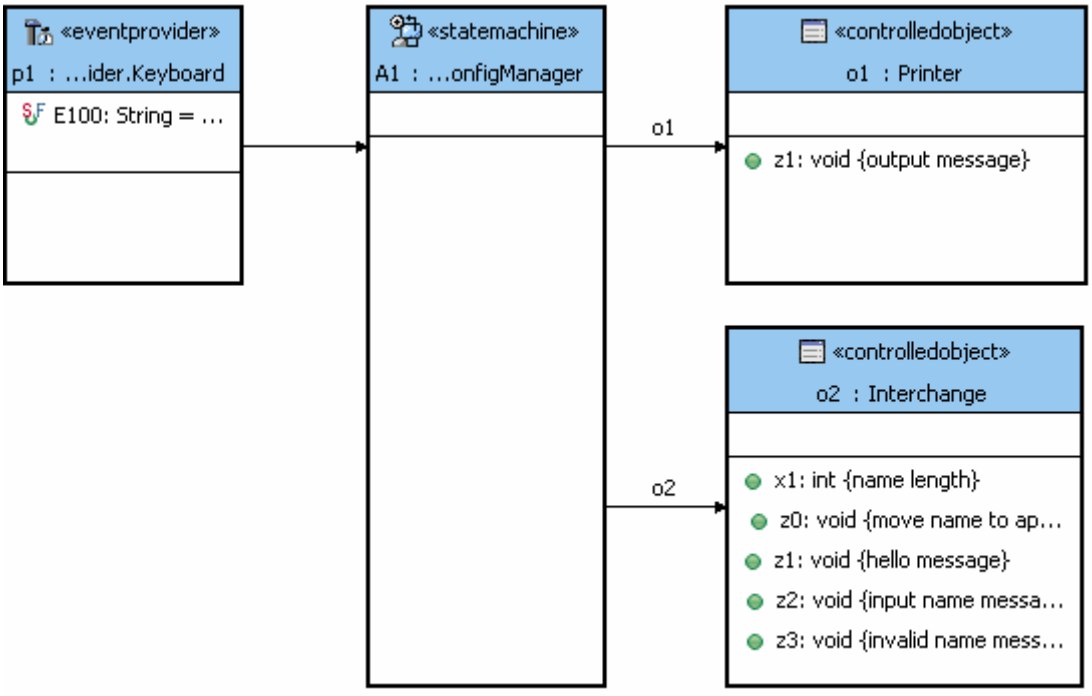

Рис. 15. Схема связей автомата **A1** из приложения *HelloUser*

Приложение *HelloUser* запрашивает имя пользователя до тех пор, пока не будет введено имя, длина которого больше четырех символов. После этого на экран выводится сообщение **Hello, <имя пользователя>!** и работа приложения завершается. Согласно предложенному в работе методу определения информации о потоках данных в составе спецификации событий и воздействий, составим таблицу спецификаций (табл. 3).

Таблица 3. Спецификация событий и воздействий приложения *HelloUser*

Событие или действие	Описание	Входные данные	Выходные данные
e100	Пользователь ввел строку	–	Строка в контексте события
o1.z1	Вывод сообщения на экран	Сообщение в контексте приложения	–
o2.x1	Длина введенного имени	Строка в контексте события	–
o2.z0	Перенести введенное имя из контекста события в контекст приложения	Строка в контексте события	Имя в контексте приложения
o2.z1	Определить приветственное сообщение (**Hello, <имя>!**)	Имя в контексте приложения	Сообщение в контексте приложения
o2.z2	Определить сообщение-запрос имени	–	Сообщение в контексте приложения
o2.z3	Определить сообщение о некорректном имени и повторный запрос ввода	–	Сообщение в контексте приложения

Данная таблица спецификаций описывает входные и выходные данные на естественном языке для удобства понимания примера. На практике обычно спецификация перечисляет имена входных и выходных данных, под которыми они запрашиваются из контекста или добавляются в контекст.

Как описывалось во второй части данной работы, в пакете *UniMod* таблицы спецификаций реализованы с помощью тэгов *Javadoc*. Тэги данного примера включены в следующий листинг кода объектов управления **`Printer`** и **`Interchange`** и поставщика событий **`Keyboard`**:

```
public class Printer implements ControlledObject {
  /**
   * @unimod.action.descr output message
   * @unimod.application.in MESSAGE
   */
  public void z1(StateMachineContext context) {
    String message =
          (String)context.getApplicationContext().getParameter("MESSAGE");
    System.out.println(message);
  }
}

public class Interchange implements ControlledObject {
  /**
   * @unimod.action.descr move name to application context
   * @unimod.event.in INPUT
   * @unimod.application.out USER
   */
  public void z0(StateMachineContext context) {
    String input =
          (String)context.getEventContext().getParameter("INPUT");
    context.getApplicationContext().setParameter("USER", input);

  }

  /**
   * @unimod.action.descr hello message
   * @unimod.application.in USER
   * @unimod.application.out MESSAGE
   */
  public void z1(StateMachineContext context) {
    Context c = context.getApplicationContext();
    c.setParameter("MESSAGE", "Hello, "
         + c.getParameter("USER")+"!");
  }

  /**
   * @unimod.action.descr input name message
```

```
     * @unimod.application.out MESSAGE
     */
    public void z2(StateMachineContext context) {
      context.getApplicationContext().setParameter("MESSAGE",
           "Please input your name..");
    }

    /**
     * @unimod.action.descr invalid name message
     * @unimod.application.out MESSAGE
     */
    public void z3(StateMachineContext context) {
      context.getApplicationContext().setParameter("MESSAGE",
           "Invalid name! Must not be shorter that 4 characters. \n"
           +"Please input correct name..");
    }

    /**
     * @unimod.action.descr name length
     * @unimod.event.in INPUT
     */
    public int x1(StateMachineContext context) {
      String input =
           (String)context.getEventContext().getParameter("INPUT");
      if (input == null) return 0;
      return input.length();
    }
}
public class Keyboard implements EventProvider {
  /**
   * @unimod.event.descr Line input from keybord
   * @unimod.event.parameter INPUT
   */
  public static final String E100 = "e100";

  // . . .
}
```

Итак, спецификация операций приложения *HelloUser* реализована с помощью тэгов *Javadoc*. Исследовав код классов **`Printer`** и **`Interchange`** с помощью алгоритма анализа кода, описанного в разделе 2.3, можно удостовериться, что реализация воздействий совпадает по входным и выходным данным со спецификацией.

Теперь изобразим шаги верификации потоков данных приложения. Напомним, что для верификации необходимо построить диаграмму деятельности, каждая вершина которой соответствует некоторому воздействию или событию на

диаграмме состояний. С помощью этой диаграммы можно легко понять, в какой последовательности вызываются воздействия при работе автомата и, таким образом, определить, не запрашиваются ли какие-нибудь не определенные данные.

Первый шаг построения диаграммы деятельности сопоставляет каждому состоянию и переходу отдельную вершину как показано на рис. 16.

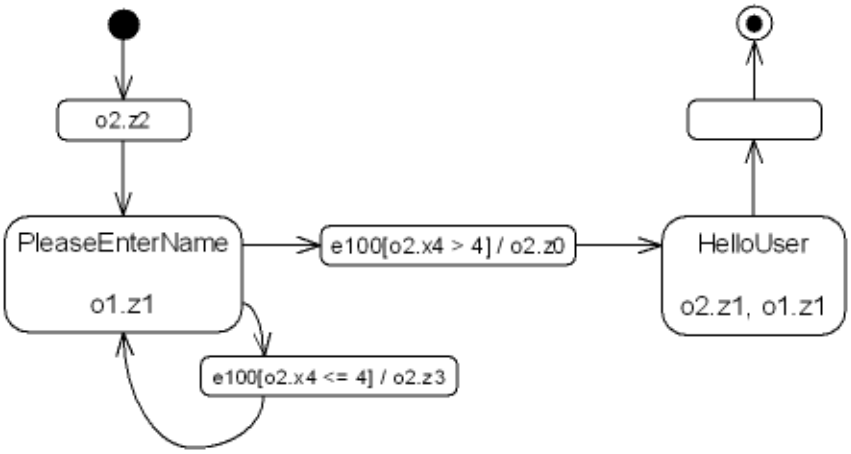

Рис. 16. Построение диаграммы деятельности, первый шаг

После удаления единственной тривиальной вершины (той, в которой не вызывается никаких воздействий и не обрабатываются события), данная диаграмма практически не изменяется (рис. 17).

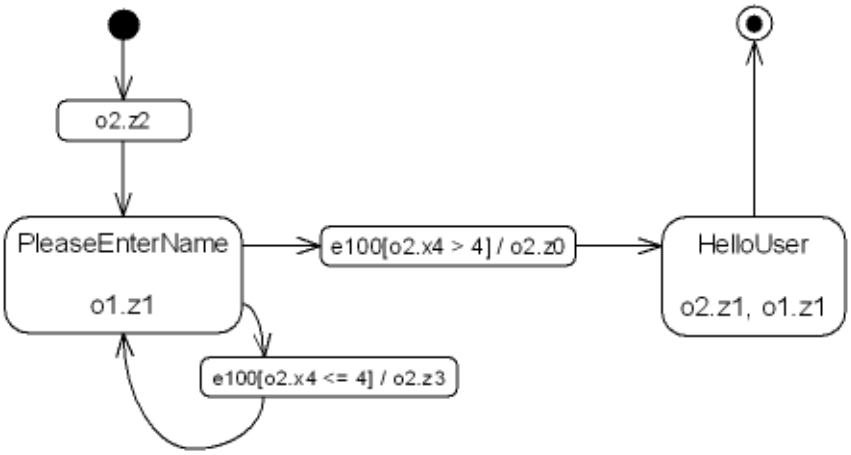

Рис. 17. Построение диаграммы деятельности, удаление тривиальных вершин

Следующим и заключительным шагом при построении диаграммы деятельности будет разбиение вершин. Вершины полученной после этого шага диаграммы (рис. 18) соответствуют вызовам воздействий и появлению событий на диаграмме состояний автомата **A1**.

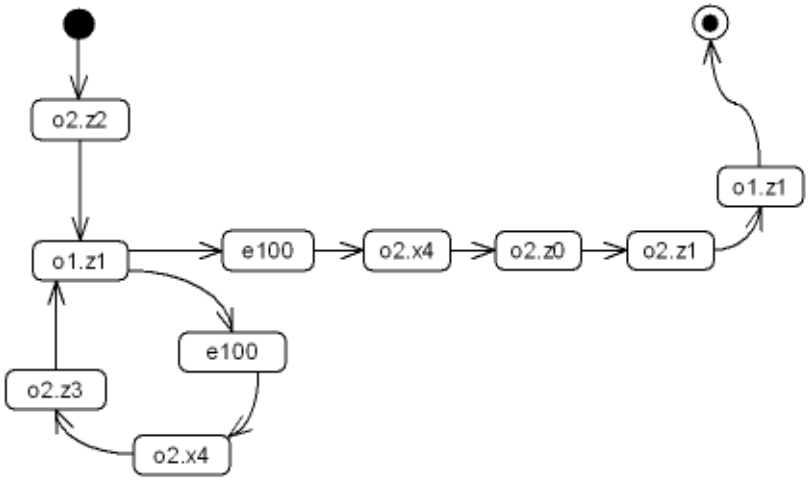

Рис. 18. Диаграмма деятельности для автомата **A1** из приложения *HelloUser*

Верификация потоков данных по этой диаграмме деятельности проводится для следующих элементов: **INPUT**, **MESSAGE**, **USER**. Например, для элемента данных **MESSAGE**, проанализировав код объектов управления или воспользовавшись спецификацией, можно определить следующее:

- элемент данных **MESSAGE** запрашивается из контекста приложения воздействием **o1.z1**;

- элемент данных **MESSAGE** опускается в контекст приложения воздействиями **o2.z1**, **o2.z2** и **o2.z3**.

Как следует из рис. 19, работа с элементом данных **MESSAGE** не должна вызвать ошибки, так как на любом пути от начальной вершины до любого вызова **o1.z1** встречается хотя бы один раз вызов одного из методов **o2.z1**, **o2.z2** и **o2.z3**. А следовательно, к моменту запроса сообщения **MESSAGE** из контекста это сообщение будет там присутствовать.

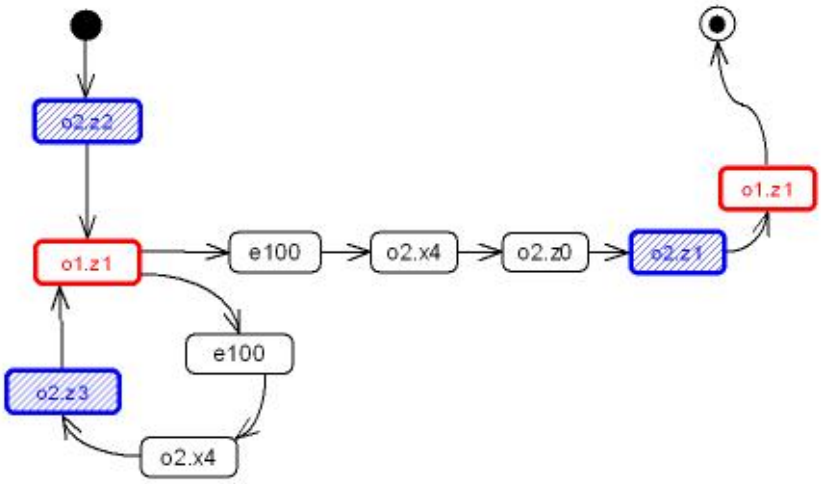

Рис. 19. Диаграмма деятельности для автомата **A1**, вершины запрашивающие и определяющие элемент данных **MESSAGE**

Аналогичная проверка остальных элементов данных показывает, что приложение *HelloUser* не содержит никаких путей, которые могли бы привести к запросу неопределенных данных.

Заметим, что за счет отсутствия в исходной диаграмме вложенных состояний и сложных охранных условий, полученная диаграмма деятельности очень проста по сравнению с обычно встречающимися на практике задачами. Из-за своей простоты приведенный пример не отображает в полной мере необходимость верификации потоков данных. Однако в реальных задачах поведение системы почти всегда является существенно более сложным. Верификация потоков данных автоматически выделяет слабые места таких систем, в том числе неочевидные с первого и даже второго взгляда, что позволяет избежать некорректной реализации и тем самым уменьшить затраты времени на отладку и тестирование программы.

ЗАКЛЮЧЕНИЕ

Предложенный в первой части настоящей работы подход для моделирования потоков данных в рамках автоматной методологии позволяет:

− повысить точность (детальность) задания модели;

− организовать автоматическую проверку соответствия реализации воздействий спецификациям;

− формализовать правило верификации потоков данных таким образом, чтобы стала возможна его автоматическая проверка.

Автоматическая верификация потоков данных в модели, как показывает опыт (и, в частности, некоторые примеры, приведенные в работе), обладает существенной практической ценностью. Она позволяет избежать некорректной реализации и тем самым сократить длительность процесса отладки и тестирования программ.

СПИСОК ЛИТЕРАТУРЫ

1. *Гайсарян С.С.* Объектно-ориентированные технологии проектирования прикладных программных систем. http://www.citforum.ru/programming/oop_rsis/index.shtml.
2. *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя. М.: ДМК, 2000.
3. *Фаулер М.* UML. Основы, 3-е издание. СПб.: Символ-Плюс, 2004.
4. *Гуров В.С., Нарвский А.С., Шалыто А.А.* Автоматизация проектирования событийных объектно-ориентированных программ с явным выделением состояний //Труды X Всероссийской научно-методической конференции "Телематика-2003". СПб.: СПбГИТМО (ТУ). 2003. http://tm.ifmo.ru.
5. *Шалыто А.А.* SWITCH-Технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
6. *Шалыто А.А., Туккель Н.И.* Танки и автоматы //BYTE/Россия. 2003. №2, с. 69-73. http://is.ifmo.ru (раздел "Статьи").
7. *Гуров В.С., Мазин М.А.* Веб-сайт проекта UniMod. http://UniMod.sourceforge.net/.
8. Eclipse Platform. http://eclipse.org.
9. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. СПб.: Невский диалект, 2001.
10. *Ларман К.* Применение UML и шаблонов проектирования. М.: Издательский дом «Вильямс», 2001.
11. *Rumbaugh J., Blaha M., Premerlanietal W.* Object-oriented modeling and design. Prentice-Hall. New Jenersy. 1991.

12. *Коуд П., Норт Д., Мейфилд М.* "Объектные модели. Стратегии, шаблоны и приложения". М.: Лори, 1999.
13. *Jacobson I.* Object-Oriented Software Engineering. ASM press, 1992.
14. Центр ИТ «Проинфотех», веб-сайт проекта «Информационные технологии для вашего предприятия», 2003. http://www.proinfotech.ru/ (раздел «Моделирование процессов данных»).
15. *Ambler S. W.* Data Flow Diagrams Overview. http://www.agilemodeling.com/artifacts/dataFlowDiagram.htm.
16. Structured Analysis and Design. http://www.infosec.jmu.edu/courses/CS555infosec99/StudyUnits/Unit7/SASD/sld001.htm.
17. *Harel D., Politi M.* Modeling Reactive Systems with Statecharts: The STATEMATE Approach. McGraw-Hill, 1998. http://www.wisdom.weizmann.ac.il/~dharel/reactive_systems.html.
18. OMG Inc. UML 2 OCL Final Adopted Specification. http://www.omg.org/cgi-bin/doc?ptc/2003-10-14.
19. *Warmer J.B., Kleppe A.G.* The object constraint language: precise modeling with UML. Addison Wesley Longman, Inc., 1999.
20. OMG Inc. Unified Modeling Language Specification. Version 1.4.2. http://www.omg.org/cgi-bin/doc?formal/04-07-02.
21. OMG Inc. UML 2.0 Superstructure Specification. http://www.omg.org/cgi-bin/doc?ptc/2004-10-02.
22. Object Constraint Language Library, University of Kent, 2003. http://www.cs.kent.ac.uk/projects/ocl/.
23. Dresden OCL Toolkit. Dresden University of Technology. http://dresden-ocl.sourceforge.net/index.html.

24. *Akehurst D.* OCL4Java library, Canterbury University. http://www.cs.ukc.ac.uk/people/staff/dha/xInterests/UMLandOCL/.
25. OCL Environment. Laboratorul de Cercetare in Informatica, Babes-Bolyai University. http://lci.cs.ubbcluj.ro/.
26. Cybernetic Intelligence GmbH OCL Tool. http://www.cybernetic.org/products.htm.
27. Octopus – OCL Tool for Precise UML Specifications. Klasse Objecten. http://www.klasse.nl/ocl.
28. OCL AddIn for Rational Rose. http://www.empowertec.de/products/rational-rose-ocl.htm
29. *Akehurst D., Patrascoiu O.*. OCL 2.0 - Implementing the Standard for Multiple Metamodels.
30. *Кормен Т., Лайзерсон Ч., Ривест Р.* Алгоритмы. Построение и анализ. М.: МЦМНО, 2000.
31. *Касьянов В.Н., Евстигнеев В.А.* Графы в программировании: обработка, визуализация и применение. СПб.: БХВ-Петербург, 2003.
32. *Гуров В.С., Мазин М.А.,Нарвский А.С., Шалыто А.А.* UML. SWITCH-технология. Eclipse //Информационно-управляющие системы. 2004, № 6.
33. *Bichler L., Radermacher A., Schurr A.* Integrating data flow equations with UML\Realtime. 1999.
34. *Harel D., Gery E.*. Executable Object Modeling with Statecharts. 1997.
35. ArgoUML, A UML design tool with cognitive support. http://argouml.tigris.org.
36. *Ahrendt W., Baar T., Beckert B., Bubel R., Giese M., Hoehnle R., Menzel W., Mostowski W., Roth A., Schlager S., Schmitt P.H.* The KeY Tool. // Department of Computing Science, Chalmers University and Goeteborg University, Technical Report in Computing Science No. 2003-5, February 2003. http://download.key-project.org/.
37. Borland Together. http://www.borland.com/together/.

38. Rational Rose. http://www.rational.com/
39. Elixir Technologies, http://www.elixirtech.com.
40. Kent Modeling Framework. KMF-team, Computing Laboratory, University of Kent. http://www.cs.kent.ac.uk/projects/kmf/.
41. Eclipse Modeling Framework. http://www.eclipse.org/emf/.

ПРИЛОЖЕНИЕ 1. Применение языка *OCL* в рамках автоматной методологии

*Я*зык *OCL* [18, 19], являющийся частью *UML*, предназначен для спецификации дополнительных требований, которые невозможно или затруднительно выразить графическими средствами *UML*. *Я*зык *OCL* легко читаем, выразителен и прост в использовании. При этом он позволяет организовывать автоматическую проверку описанных ограничений.

Многие программные пакеты – *Dresden OCL Toolkit* [23], *OCL for Java* [24], *OCL Environment* [25] – предлагают средства для автоматического контроля ограничений во время исполнения программ.

Интерпретаторы *OCL* встроены во многие *CASE*-пакеты: *Dresden OCL Toolkit* [23] был встроен в *CASE*-пакет *ArgoUML* [35], в рамках проекта *KeY* [36] реализована поддержка *OCL* для *CASE*-средства *TogetherCC* [37], [28] обеспечивает поддержку *OCL* в *Rational Rose* [38], для *CASE*-пакета *Elixir* [39] компанией *iContract* реализован встраиваемый модуль, поддерживающий *OCL*. Наиболее актуальным в рамках данной работы является пакет интерпретации *OCL*, написанный Девидом Акехерстом (David Akeherst) [29, 20]. Обработка выражений этим пакетом разделяется на два основных этапа (рис. 20).

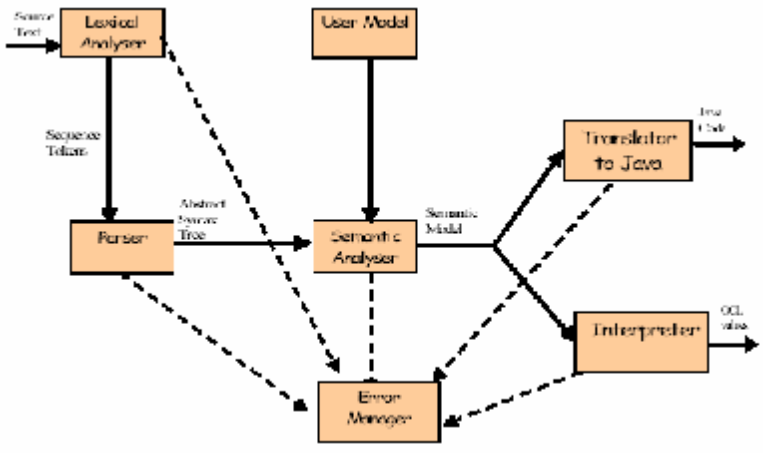

Рис. 20. Схема реализации обработчика *OCL*

Результатом первого этапа является семантическая модель выражения, которая на втором этапе интерпретируется или является базой для генерации кода на языке *Java*. При этом вторая фаза может производиться разными модулями, что обуславливает три варианта этого пакета, реализованные для работы с языком *Java*, для встраивания в среду разработки *KMF* (*Kent Modeling Framework*) [40], и как модуль для среды моделирования *EMF* (*Eclipse Modeling Framework*) [41].

Ограничение указывает на зависимость между соответственными значениями двух объектов, либо между различными значениями одного объекта. Ограничение может быть выражено в виде некоторой функции (количественное ограничение), либо отношения (качественное ограничение). В рамках этой работы наиболее актуальны ограничения на пред- и постусловия функций, а также на состояния и события. Важным видом ограничений являются инварианты: утверждения о том, что значение некоторой функции от атрибутов, состояний и событий остается постоянным при функционировании объекта.

Итак, с помощью *OCL* можно указать:

- • инварианты на классы и типы в модели классов;
- • инварианты на стереотипы;
- • пред- и постусловия операций и методов;
- • охранные условия на диаграммах состояний;
- • ограничения на операции.

Опишем способы включения *OCL* в автоматную модель.

Для поставщика событий может быть дано описание, определяющее последовательность появления событий. Примером ограничения такого рода

может служить заявление, что пост события в чат не может произойти до того, как в систему зайдет хоть один посетитель.

Для каждого типа контекста возможно описание инвариантов.

```
-- Контекст пользователя должен содержать идентификатор пользователя:
context UserContext  inv: not getParameter(“userId”).isEmpty
                          and getParameter(“userId”).length > 5
-- Контекст приложения должен содержать имя своего домена:
context ApplicationContext inv: not getParameter(“docroot”).isEmpty
```

Для каждого действия возможно описание пред- и постусловий на языке *OCL*.

```
-- Метод z09 сохраняет зарегистрировавшегося покупателя Customer в базу,
-- присваивая ему некоторый Id. Необходимо получить правильно
-- оформленный заказ:
z09 pre: (not UserContext.getParameter(“Customer”).isEmpty)
         and (not UserContext.getParameter(“Customer”).LastName.isEmpty)
         and (UserContext.getParameter(“Customer”).password.length > 5)
         and  (UserContext.getParameter(“Customer”).email.contains(“@”))
z09 post: not UserContext.getParameter(“Customer”).Id.isEmpty
```

Проверка пред- и постусловий проводится во время выполнения программы, соответственно перед и после вызова соответствующего метода.

Верификация инвариантов проводится каждый раз при изменении хранимых контекстами данных.

В случае невыполнения тех или иных ограничений возможны следующие варианты действий:

- ассоциировать ограничение с некоторым методом, вызываемым, когда ограничение не выполняется;
- вызывать прерывание или полностью останавливать программу;
- выводить служебную информацию в лог программы;

- в случае использования транзакций, «откатывать» их;
- выводить сообщение оператору или пользователю.

Более «жесткие» реакции на провал проверки ограничений обычно используются при отладке программы. Например, в процессе отладки часто оказывается удобным вывод ошибок в лог программы с последующим ее остановом. После того как программа была качественно протестирована, проверка ограничений может стать не нужной. В этом случае ее можно отключить и код проверки не только не будет выполняться, но и не будет включен в финальную версию программы, предоставляемую заказчику.