УДК 004.434

ИСПОЛЬЗОВАНИЕ ЯЗЫКА ОПИСАНИЯ ДИАГРАММ

К. Б. Степанян*,*
начальник управления операционного обслуживания
ОАО «Управляющая компания «Арсагера»

Обсуждается практическое применение языка DiaDeL для описания графо-подобных диаграмм на примере диаграмм состояний. Приводится пример известной диаграммы состояний телефона, автоматически построенной по описанию на предложенном языке. Язык DiaDeL позволяет формально определить графический синтаксис (нотацию) диаграмм заданного типа и связать нотацию с семантикой, заданной в форме набора классов.

Ключевые слова *— графический язык, абстрактный синтаксис, метамодель, визуализация диаграмм, диаграмма состояний.*

Введение

Актуальность и эффективность практического применения двумерных графо-подобных диаграмм в настоящее время не ставятся под сомнение. Более того, большинство типов диаграмм имеет не только строго формализованную нотацию, но и семантику, что позволяет тем или иным образом интерпретировать их. Подобные диаграммы называются визуальными языками. Наиболее яркое применение они нашли в областях моделирования, проектирования и реализации программного обеспечения.

Ранее нами был предложен язык описания диаграмм DiaDeL (Diagram Definition Language) [1–3], основанный на предположении, что семантика каждой отдельной диаграммы задана в виде набора конкретных программных объектов определенных классов. Набор всех возможных классов объектов, которые могут появляться на диаграммах данного типа, называется семантической моделью, а набор объектов, соответствующих конкретной диаграмме, называется экземпляром ее семантической модели.

Настоящая статья демонстрирует применение языка DiaDeL на примере построения диаграмм состояний с помощью системы, описанной в работах [1, 2]. Существует множество различных диалектов диаграмм состояний. Из всех возможных выбраны диаграммы состояний языка UML [4], как наиболее показательные по ряду критериев:

- диаграммы состояний — очень важный и практически широко используемый тип диаграмм [5, 6];
- нотация позволяет продемонстрировать большинство возможностей языка DiaDeL;
- существует полная и непротиворечивая спецификация семантики [7];
- существует реализованная семантическая модель [8].

Следует оговориться сразу, что в статье не приводится синтаксис языка DiaDeL, он рассмотрен в других статьях [1, 2]. Основное назначение работы — продемонстрировать «выразительные» способности и возможности языка.

Анализ семантической модели диаграммы состояний

Первым шагом построения отображения диаграммы является анализ ее семантической модели. Цель анализа — определить, удовлетворяет ли модель требованиям, предъявляемым языком DiaDeL. Весь процесс можно разбить на следующие шаги.

1. Определение элементов, которые будут представлены на диаграмме, как фигуры или декорации. Для простоты будем называть их *элементы-вершины*.

2. Определение элементов, которые будут представлены на диаграмме, как линии (*элементы-связи*).

3. Определение «корневого» элемента, соответствующего самой диаграмме (*элемент-диаграмма*).

---

* Научный руководитель — канд. физ.-мат. наук, заведующий лабораторией астрономического программирования Института прикладной астрономии РАН *Ф. А. Новиков.*

4. Проверка, что эти элементы предоставляют необходимую информацию в соответствии с требованиями DiaDeL:

а) элемент-диаграмма и элементы-вершины, которые должны быть отображены, как контейнеры, предоставляют содержащиеся в них элементы-вершины в виде одного списка;

б) элементы-вершины, которые должны на диаграмме иметь инцидентные элементы-связи, предоставляют их в виде одного списка;

в) элементы-связи предоставляют элемент-вершину, инцидентную началу, и элемент-вершину, инцидентную концу.

5. Проверка, что всю остальную информацию, необходимую для отображения (если таковая имеется), модель предоставляет не в виде списков или массивов, поскольку DiaDeL не поддерживает циклы.

Семантическая модель диаграммы состояний отображена на рис. 1. Модель взята из официальной спецификации языка UML [7]. Для построения диаграмм состояний использовалась реализация модели EclipseUML v2.1 [8]. В ней все сущности модели представлены интерфейсами, а все свойства имеют традиционный для Java вид *getProperty()*, *setProperty()*.

Проанализируем представленную модель и пройдем по шагам описанного выше процесса.

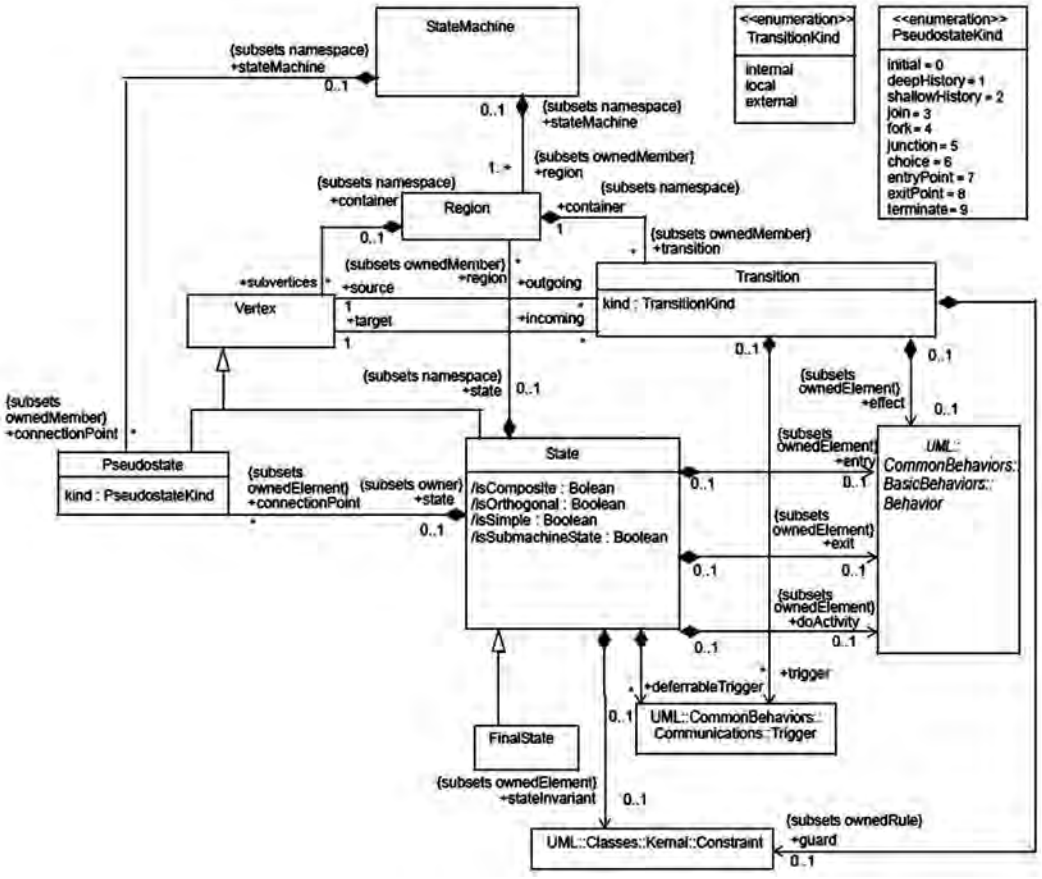
1. Элементы-вершины — это интерфейсы *State* (представляет состояние как простое, так и составное), *Pseudostate* (представляет специальные вершины) и *FinalState* (представляет конечное состояние).

2. Элемент-связь — это интерфейс *Transition* (представляет переход между состояниями).

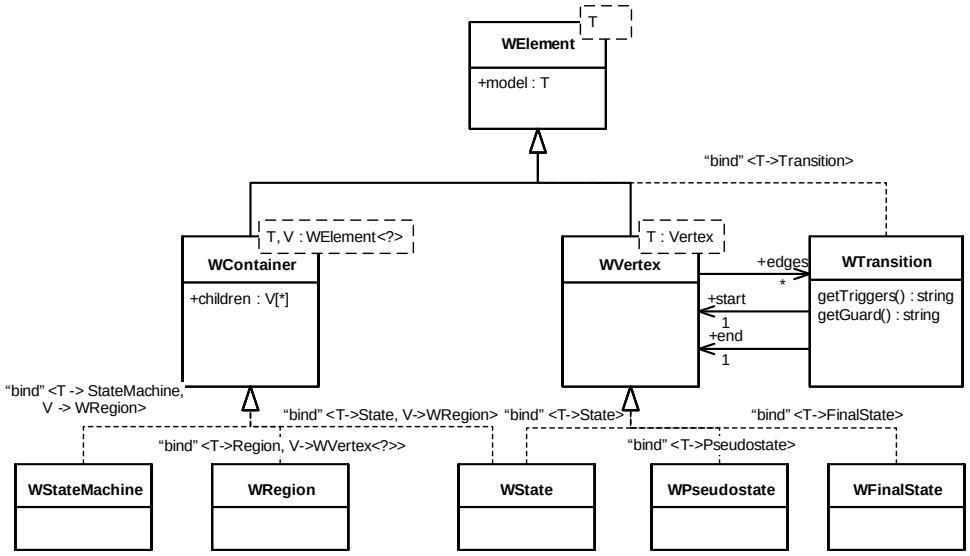
3. Элемент-диаграмма — это интерфейс *StateMachine* (представляет сам конечный автомат).

4. Проанализируем, предоставляет ли модель информацию в соответствии с требованиями DiaDeL. Заметим, что информация должна предоставляться с помощью вызова одного метода/свойства (или цепочки вызовов, которую можно записать через точки, например, *m.getParent().select("*", this)*).

а. Как видно из диаграммы, модель поддерживает регионы внутри состояний и конечного автомата. Конечный автомат (или состояние) содержит список регионов, каждый из которых содержит вложенные состояния. В силу сжатости изложения мы не будем ставить цель отобразить регионы и примем допущение, что конечный автомат и составные состояния имеют ровно один



■ ***Рис. 1.*** *Семантическая модель диаграммы состояний UML*



■ ***Рис. 2.*** *Структура классов-оболочек*

регион, содержащий вложенные состояния. Следовательно, получить коллекцию вложенных состояний можно следующим вызовом: *m.getRegions().get(0).getSubvertices()*, где *m* — это экземпляр, реализующий интерфейс *StateMachine* или *State*.

б. Интерфейс *Vertex* — базовый для всех элементов-вершин — предоставляет отдельные списки исходящих и входящих элементов-связей. Следовательно, этому требованию модель не удовлетворяет.

в. Интерфейс *Transition* позволяет определить начало и конец перехода, используя свойства *getSource()* и *getTarget()*.

5. **Мы не будем описывать анализ всех данных**, которые необходимо отобразить на диаграмме, ограничившись лишь тем, что не может быть обработано в секции **refresh()** для отображения согласно нотации. Заметим, что триггеры, которые инициируют переход, представлены списком *getTriggers()* (ассоциация между *Transition* и *Trigger*). Отобразить их необходимо в одну строку через запятую. Условие перехода *getGuard()* (ассоциация между *Transition* и *Constraint*) представляется набором объектов с корневым объектом класса *Constraint* (эта часть модели на рисунке не представлена). Для конструирования из них строки условия требуется циклический обход. В силу ограничений подобную обработку невозможно написать в секции **refresh()**.

Проведенный анализ показал, что экземпляры модели не могут быть использованы напрямую для отображения диаграммы состояний, и мы не можем изменить саму семантическую модель, поскольку она является внешним модулем для нашей системы. Решением в подобной ситуации может быть создание классов-оболочек, которые «обертывают» исходные объекты, предоставляют к ним доступ, а также предоставляют данные в удобном для обработки формате. Таким подходом является применение шаблона проектирования «Адаптер», описанного в книге [9]. Поскольку задача создания классов-оболочек является типовой и не представляет большого интереса, мы опустим описание ее построения, приведя лишь их структуру (рис. 2) и краткие комментарии.

Приведенная модель классов-оболочек построена на основании параметризованных классов. Базовый класс *WElement<T>* реализует доступ к объекту оборачиваемого класса (оборачиваемый класс представляется везде параметром *T*). Класс-оболочка для перехода *WTransition* позволяет получить триггеры и условие перехода сразу в виде строки. *WTransition* предоставляет доступ к инцидентным вершинам, как к объектам *WVertex<T : Vertex>*. Те, в свою очередь, предоставляют доступ ко всем инцидентным переходам через один список. Таким образом, решаются проблемы, обозначенные выше.

Описание графических конструкций

Описание графических конструкций начнем с описания представления конечного состояния, которое изображается значком ◉.

Конечное состояние должно иметь фиксированные размеры, поэтому в качестве основной конструкции необходимо использовать декорацию. Она же будет отображать внешнюю окружность. Для отображения внутреннего круга прикрепим к основной декорации вспомогательную.

Получится следующий программный код на языке DiaDeL:

```
decoration circle { // внутренний круг
    shape = ellipse;
    minsize = (30,30);
    brush.color = colors.black;
}
decoration finalstate { // основная конструкция
    shape = ellipse;
    minsize = (38, 38);
    brush.style = styles.no;
    attachments[50%, 50%] = (:circle)[50%, 50%];
}
```

По умолчанию все графические примитивы рисуются сплошным пером толщиной в 1 пиксель и закрашиваются белым цветом, поэтому здесь и далее значения по умолчанию в описании опускаются.

Для правильного отображения конструкции ее необходимо связать с сущностью семантической модели.

```
bridge <finalstate> {
    model := org.diadel.visualizer.wrapper.statemachine.WFinalState;
    edges := model.getEdges();
}
```

Вышеприведенный программный код связывает конструкцию *finalstate* с сущностью семантической модели *WFinalState* и декларирует, что элементы-связи, которые должны быть отображены как ребра, инцидентные конечному состоянию, доступны через метод *getEdges()*.

Специальные вершины, так же как и конечное состояние, должны отображаться на диаграмме в виде значков фиксированного размера. Следовательно, для их представления необходимо использовать декорацию. Сложность заключается в том, что в семантической модели все специальные вершины представлены одной сущностью — *Pseudostate*. В зависимости от типа (определяется свойством *kind*), которым обладает экземпляр *Pseudostate*, его необходимо отображать, используя различные графические конструкции. В рамках данной статьи мы ограничимся отображением только следующих специальных вершин:

●	начальная вершина	Ⓗ	вершина истории
◇	вершина выбора	⊗	вершина точки выхода

Поскольку отображаться они должны с помощью разных графических примитивов, то мы не можем взять за основу ни один из последних. Для решения этой проблемы используем следующий подход. Возьмем за основу декорацию без графического примитива и в центре прикрепим к ней декорации с различными графическими примитивами. В секции **refresh()** в зависимости от типа специальной вершины будем управлять видимостью прикрепленных декораций таким образом, чтобы на экране получались требуемые визуальные конструкции. Нам потребуется три вспомогательные декорации:

— декорация в виде креста:

```
decoration cross {
    shape = polyline((0%,0%), (100%,100%), (50%,50%), (100%,0%), (0%,100%));
    minsize = (21, 21);
}
```

— декорация в виде ромба:

```
decoration diamond {
    shape = polygon((0%,50%), (50%,0%), (100%,50%), (50%, 100%));
    minsize = (30,30);
}
```

— декорация в виде круга (или окружности) приведена уже выше при описании представления конечного состояния. Ее можно использовать еще раз, расширив описанием шрифта для отображения текста. Таким образом, в описание, приведенное выше, необходимо добавить следующие строки:

```
decoration circle {
    ...
    text.font.name = «Arial»;
    text.font.size = 10;
}
```

Основная декорация, к середине которой крепятся все перечисленные выше конструкции:

```
decoration pstate {
    minsize = (30,30);
    attachments[50%, 50%] = (c:circle)[50%, 50%];
    attachments[50%, 50%] = (d:diamond)[50%, 50%];
    attachments[50%, 50%] = (x:cross)[50%, 50%];
}
```

Связь декорации *pstate* с сущностью семантической модели:

```
bridge <pstate> {
    model := org.diadel.visualizer.wrapper.statemachine.WPseudostate;
    edges := model.getEdges();

    refresh() {
        // вершина выбора
        d.visible = 6 == model.getModel().getKind().getValue();
        c.visible = not d.visible;
        // начальная вершина
        if (0 == model.getModel().getKind().getValue()) {
            c.brush.color = colors.black;
        }
        // вершина истории
        if (2 == model.getModel().getKind().getValue()) {
            c.brush.style = styles.no;
            c.text.value = «H»;
        }
        // вершина точка выхода
        x.visible = 8 == model.getModel().getKind().getValue();
        if (x.visible) {
            c.brush.color = colors.white;
        }
    }
}
```

Данный программный код написан в расчете только на отображение заявленных четырех типов, но его несложно расширить и на остальные типы специальных вершин.

Самой сложной конструкцией у описываемой диаграммы является состояние. Состояние может быть простым (рис. 3, *а*) и составным (рис. 3, *б*) (содержащим вложенные состояния), но должно отображаться одной конструкцией, поскольку в семантической модели это одна сущность. Более того, нотация **UML-диаграмм состояний** допускает перечисление внутри состояния действий, связанных с состоянием.

Опишем сначала конструкцию, которая будет отображать на диаграмме состояние простым способом. А затем расширим ее так, чтобы она умела отображать состояние как в сложном, так и в простом варианте. Для представления состояния будем использовать фигуру, как конструкцию, позволяющую изменять свои размеры:

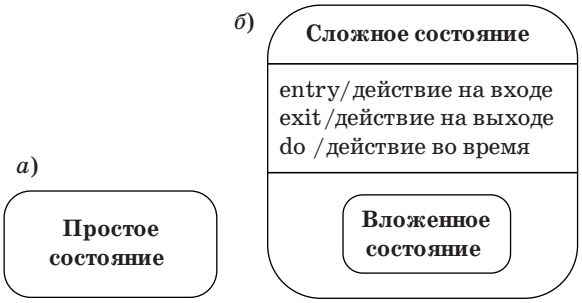
```
figure state {
    shape = roundedrectangle;
    minsize = (100, 50);
    text.font.name = «Arial»;
    text.font.size = 10;
}
```

Для правильного отображения свяжем конструкцию с сущностью *WState* из обертки семантической модели и инициализируем в секции *refresh()* текст внутри конструкции названием состояния:

```
bridge <state> {
    model := org.diadel.visualizer.wrapper.statemachine.WState;
    edges := model.getEdges();
    refresh() {
        text.value = model.getModel().getName();
    }
}
```

Этим небольшим программным кодом мы добились представления состояния в простом варианте. Проанализируем сложный вариант отображения (рис. 4).

Вся конструкция состоит из трех частей: первая отображает имя состояния, вторая — действия состояния, третья — вложенные состояния. Следовательно, основную конструкцию можно описать как фигуру, состоящую из трех других фигур (частей), расположенных вертикально. Первая часть будет отображать имя. Вторая часть, как и основная фигура, тоже состоит из трех конструкций, только они являются текстом. А третья часть является фигурой-контейнером, которая может содержать в себе любые конструкции-вершины диаграммы (потому что составное состояние может содержать в себе не только другие состояния, но и специальные вершины, и конечное состояние). Итак, нам потребуется описать пять конструкций: основную фигуру, три части и текст, из которого состоит вторая часть. Начнем с самого простого — текста и первой части:



■ ***Рис. 3.*** *Варианты представления состояния: а — простой; б — сложный*

■ ***Рис. 4.*** *Схематичное изображение сложного варианта представления состояния*

```
text label {
    font.size = 8;
    font.name = «Arial»;
    hor_align = align.left; // выравнивание по левому краю
}

figure name_section {
    shape = none;
    minsize = (70, 30);
    // используем стандартный шрифт
    text.font.size = 10;
    text.font.bold = true;
}
```

Вторая часть состоит из трех конструкций *label*, расположенных вертикально. Для того чтобы отделить ее от первой части, укажем горизонтальную линию, рисуемую в самом верху фигуры в качестве ее графического примитива:

```
figure behavior_section {
    shape = polyline((0%,0%), (100%,0%));
    minsize = (70, 13);
    layout = vertical;
    parts = {entry : label, exit : label, do : label};
}
```

Третья часть содержит в себе все допустимые конструкции-вершины диаграммы. Содержащи-

еся конструкции могут быть расположены свободно внутри фигуры:

```
figure substates_section {
    shape = polyline((0%,0%), (100%,0%));
    minsize = (70, 30);
    layout = free;
    contain = {state, pstate, finalstate};
}
```

Модифицируем описанную ранее фигуру для простого варианта отображения состояния так, чтобы она состояла из трех представленных выше частей:

```
figure state {
    shape = roundedrectangle;
    minsize = (100, 50);
    layout = vertical;
    parts = {ns : name_section, bs : behavior_section,
            ss : substates_section};
}
```

Определения для текста были убраны, поскольку мы больше не будем отображать внутри конструкции текст. Описание частей фигуры и их расположение добавлено.

Значительные изменения претерпит и семантический мост, связанный с состоянием. Однако перед описанием изменений поставим еще одну задачу. Для того чтобы построенные диаграммы были компактней, не будем отображать части состояния в том случае, если они пусты. Таким образом, если у состояния нет вложенных состояний (или других сущностей) и нет связанных с ним действий, оно должно выглядеть, как в случае простого варианта отображения:

```
bridge <state> {
    model := org.diadel.visualizer.wrapper.statemachine.WState;
    edges := model.getEdges();

    refresh() {
        // инициализируем текст внутри первой части
        ns.text.value = model.getModel().getName();
        // не отображать текст, если нет соответствующего действия
        bs.entry.visible = model.getModel().getEntry() != null;
        // если отображаем, то инициализируем типом и названием
        if (bs.entry.visible) {
            bs.entry.value = « entry / « +
                    model.getModel().getEntry().getName();
        }
        bs.exit.visible = model.getModel().getExit() != null;
        if (bs.exit.visible) {
            bs.exit.value = « exit / « +
                    model.getModel().getExit().getName();
        }
        bs.do.visible = model.getModel().getDoActivity() != null;
        if (bs.do.visible) {
            bs.do.value = « do / « +
                    model.getModel().getDoActivity().getName();
        }
        // если ни один текст не отображается, скрываем и вторую часть
        bs.visible = bs.entry.visible or bs.exit.visible or bs.do.visible;

        // отображаем третью часть, только если есть вложенные состояния
        ss.visible = model.getChildren().size() > 0;
        if (ss.visible) {
            // третья часть по вертикали занимает все оставшееся место
            ss.size.height = size.height - ns.size.height;
            if (bs.visible) {
                ss.size.height = ss.size.height - bs.size.height;
            }
        }
    }
}
```

Теперь, когда состояние отображается составной фигурой и все вложенные состояния должны отображаться в третьей части, необходимо также описать семантический мост для нее, чтобы указать, как извлекать информацию из семантической модели о вложенных сущностях:

```
bridge <substates_section> {
    model := parent.model;
    children := model.getChildren().get(0).getChildren();
}
```

Обратите внимание, что семантический мост не связывает часть с конкретной сущностью, а ссылается на сущность родительской фигуры. Подобным образом указывается, что этот семантический мост должен быть использован в том случае, если фигура является частью другой фигуры.

Пришло время описать графическое представление перехода. Переход должен отображаться следующим образом (рис. 5).

Конструкцию выше можно описать, как сплошную линию, к концу которой прикреплена открытая стрелка, а к середине — текст. Объявлять новую конструкцию для текста необходимости нет, так как можно использовать уже объявленную конструкцию *label*. Открытую стрелку опишем, как декорацию с графическим примитивом — полилинией:

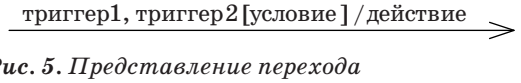
```
decoration open_arr {
    shape = polyline((0%,0%), (100%,50%), (0%,100%));
    minsize = (20,15);
}
```

Описание конструкции для перехода будет основываться на линии с двумя прикреплениями:

```
line transition {
    links = {(state,state), (state,pstate), (pstate,state),
        (pstate,pstate), (state,finalstate), (pstate,finalstate)};
    attachments[100%] = (:open_arr)[100%,50%];
    attachments[50%] = (lbl:label)[50%,110%];
}
```

Пара смежных вершин *state* и *pstate* входит в объявление дважды, поскольку графические связи в DiaDeL направленные.

Свяжем линию *transition* соответствующей оболочкой из нашей модели-обертки:



■ ***Рис. 5.*** *Представление перехода*

```
bridge <transition> {
    model := org.diadel.visualizer.wrapper.statemachine.WTransition;
    start := model.getStart();
    end   := model.getEnd();

    refresh() {
        lbl.value = «»;
        if («» != model.getTriggers()) {
            lbl.value = model.getTriggers();
        }
        if («» != model.getGuard()) {
            lbl.value = lbl.value + « [«;
            lbl.value = lbl.value + model.getGuard();
            lbl.value = lbl.value + «]»;
        }
        if (null != model.getModel().getEffect()) {
            lbl.value = lbl.value + « / «;
            lbl.value = lbl.value +
                    model.getModel().getEffect().getName();
        }
    }
}
```

Семантический мост для линии содержит объявления для определения инцидентных ее концам сущностей из модели и инициализацию метки на переходе. Для получения всех возможных триггеров, перечисленных через запятую, используется метод *getTriggers()* из обертки.

Последним штрихом описания будет декларация самой диаграммы состояний и семантического моста для нее:

```
diagram state_machine_diagram {
    contain = {state, transition, pstate, finalstate};
}
bridge <state_machine_diagram> {
    model := org.diadel.visualizer.wrapper.statemachine.WStateMachine;
    children := model.getChildren().get(0).getChildren();
}
```

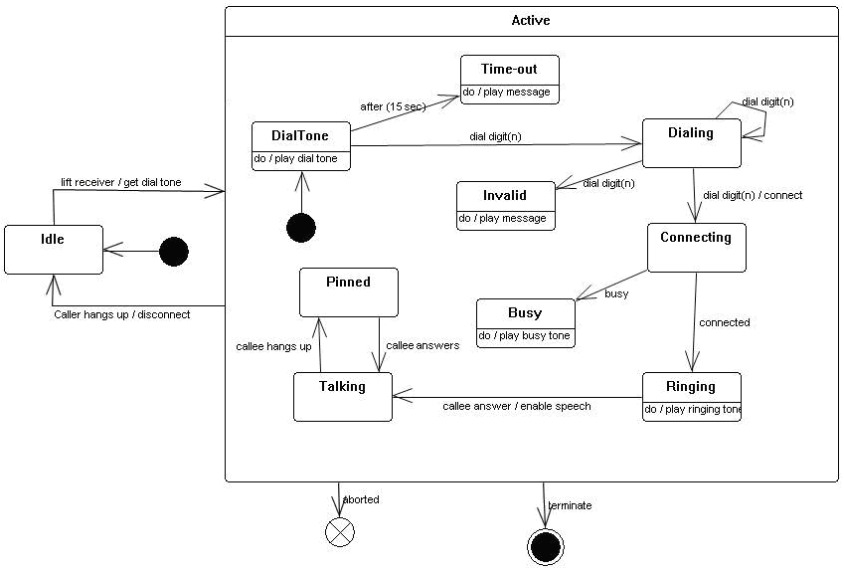
Получение коллекции сущностей, которые должны быть отображены на диаграмме, имеет несколько странный вид — *model.getChildren().get(0).getChildren()*. Это обусловлено тем, что состояния и прочие сущности-вершины содержатся в диаграмме опосредованно, через регионы. Ранее, в разделе «Анализ семантической модели», мы сделали предположение, что диаграмма и все составные состояния будут иметь ровно один регион, который и содержит все вложенные состояния и другие сущности-вершины.

Пример построенной диаграммы состояний

Используем приведенное выше DiaDeL-описание для построения экземпляра диаграммы состояний. Для этого нам понадобится экземпляр семантической модели, показанной на рис. 1, который будет отображен согласно описанию. В качестве экземпляра семантической модели возьмем машину состояний, моделирующую работу телефона [7]. Передав описание и экземпляр модели на вход системе автоматического представления диаграмм, мы получим результат, представленный на рис. 6.

Построенная диаграмма соответствует графической нотации диаграмм состояний UML. Мы добились компактного представления состояний, т. е. отображаются только те секции состояния, которые содержат информацию для отображения. Диаграмма не лишена недостатков, но все они носят графический характер и являются недостатками системы визуализации.

Следует также отметить, что вопрос укладывания построенных диаграмм лежит вне рамок языка **DiaDeL, т. е. расположение вершин и линий**, которое читатель видит на диаграмме, было выполнено вручную.



■ ***Рис. 6.*** *Диаграмма состояний, отображающая модель работы телефона, построенная по DiaDeL-описанию*

Заключение

В статье представлено практическое использование языка описания диаграмм DiaDeL на примере построения диаграммы состояний UML. Проанализирована входная семантическая модель, описаны на языке DiaDeL графические конструкции и их связь с элементами семантической модели. Приведена построенная диаграмма состояний.

Пример использования языка DiaDeL показывает, что поставленные перед языком цели успешно достигнуты. Язык позволяет гибко определять различные графические конструкции и свойства их отображения, связывать конструкции с внешними сущностями и определять их конечное представление в зависимости от состояния представляемых сущностей.

Платой за гибкость являются достаточно жесткие требования, предъявляемые к семантической модели. Однако их можно удовлетворить, разработав обертку для семантической модели, что в большинстве случаев не является сложной задачей для опытного программиста.

Язык DiaDeL дает возможность определять конечное отображение конструкции в зависимости от состояния представляемого объекта. Это позволяет, с одной стороны, решить базовые задачи, такие как отображение имени или других данных, с другой стороны, создавать более сложные конструкции и подстраивать их представление под отображаемый объект, что вносит еще одну «степень свободы» в проектирование конструкций и связей между ними и сущностями семантической модели.

Текущая версия реализации языка DiaDeL еще не может претендовать на промышленный уровень. Однако положенные в основу языка идеи обеспечивают его дальнейшее развитие. Концепция внешней семантической модели соответствует современному компонентному построению систем. Текстовая форма языка дает возможность расширять его в дальнейшем новыми конструкциями в целях повышения удобства использования и предоставления новых возможностей. Эти и другие факты позволяют считать разработку языка DiaDeL востребованной и перспективной.

Литература

1. **Степанян К. Б.** Язык описания диаграмм // Научно-технические ведомости СПбГПУ. 2006. № 6-1. С. 36–41.
2. **Новиков Ф. А., Степанян К. Б.** Язык описания диаграмм // Информационно-управляющие системы. 2007. № 4. С. 28–36.
3. **Новиков Ф. А., Степанян К. Б.** Использование порождающего программирования при реализации языка описания диаграмм // Информационно-управляющие системы. 2008. № 6. С. 33–36.
4. **Буч Г., Якобсон А., Рамбо Д.** UML. 2-е изд. СПб.: Питер, 2006.
5. **Канжелев С., Шалыто А.** Автоматическая генерация кода программ с явным выделением состояний / Software Engineering Conference (Russia) «Paths to Competitive Advantage» (SECR 2006). М., 2006. С. 60–63.
6. **Канжелев С., Шалыто А.** Преобразование графов переходов, представленных в формате MS Visio, в исходные коды программ для различных языков программирования (инструментальное средство MetaAuto). 102 с. http://is.ifmo.ru/projects/metaauto
7. **OMG Unified Modeling Language** (OMG UML), Superstructure, v2.1.2. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF
8. **EclipseUML.** http://www.eclipseplugincentral.com/displayarticle572.html
9. **Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2004. 366 с.