Статья опубликована в материалах конференции *Software Engineering Conference (Russia) – SEC(R)2006*, с. 43–46.

ФОРМАЛЬНАЯ СЕМАНТИКА ДИАГРАММ СОСТОЯНИЙ, УДОБНАЯ ДЛЯ ПРАКТИЧЕСКОГО ПРИМЕНЕНИЯ

Маврин Павел Юрьевич
магистрант СПбГУ ИТМО
email: mavrin@rain.ifmo.ru

Корнеев Георгий Александрович
ассистент СПбГУ ИТМО
email: kgeorgiy@rain.ifmo.ru

Шалыто Анатолий Абрамович
доктор технических наук
профессор СПбГУ ИТМО
email: shalyto@mail.ifmo.ru

Аннотация

В статье предлагается новая семантика для диаграмм состояний (statechart), используемых в языке моделирования UML. От известных семантик предлагаемую отличает формальное описание и удобство при практическом использовании.

Ключевые слова: *Диаграммы состояний, семантика.*

1. Введение

Одна из часто встречающихся в программировании задач — программирование систем, поведение которых определяется некоторым текущим состоянием. Например, электронные часы могут находиться в состояниях: «время», «дата», «секундомер» и т. д. Как правило, подобные задачи естественным образом появляются при программировании реактивных систем — событийно-управляемых систем, реагирующих на воздействия внешней среды.

Диаграммы состояний [1] являются одной из наиболее мощных и гибких форм описания поведения реактивных систем.

Для того, чтобы поведение системы соответствовало диаграмме состояний, описывающей его, прежде всего, необходимо определиться с используемой семантикой. Здесь могут возникнуть некоторые проблемы, так как известные семантики либо недостаточно формализованы [2, 3], либо практически неприменимы в реальных системах [4]. Отметим также работу [5], в которой была выполнена попытка формализации семантики диаграмм состояний, в которых отсутствуют AND-состояния [2].

В настоящей работе предлагается семантика диаграмм состояний, которая лишена указанных недостатков.

2. Структура модели

Для начала выделим элементы диаграмм состояний, которые будут использоваться в предлагаемой модели. По возможности в этом разделе будет применяться терминология из работы [2].

2.1. События

Из всех типов событий, используемых в диаграммах состояний *UML*, в предлагаемой модели сохранится только один тип — сигналы. Все остальные типы могут быть промоделированы с использованием только этого типа событий. Будем разделять события на внешние и внутренние. Внутренние события — это сигналы, посланные системой самой себе в процессе обработки другого события. Все остальные сигналы будем считать внешними.

2.2. Состояния

В предлагаемой модели сохранены только основные типы состояний, используемые в диаграммах состояний: простые состояния, OR-состояния и AND-состояния.

2.3. Дерево состояний

Структура состояний может быть представлена в виде дерева, в корне которого находится основное состояние, а в листьях — простые состояния. Это дерево будем называть деревом состояний.

2.4. Активные состояния. Дерево активных состояний

В каждый момент времени некоторые состояния являются активными. При этом соблюдаются следующие условия:

- основное состояние активно;
- если OR-состояние активно, то активно ровно одно из его подсостояний, иначе все его подсостояния неактивны;
- если AND-состояние активно, то активны все OR-состояния, объединенные им, иначе все они неактивны.

Таким образом, активные состояния также могут быть представлены в виде дерева, в корне которого находится основное состояние, а в листьях — простые состояния. Это дерево будем называть деревом активных состояний.

2.5. Переходы

Переходы определяют правила изменения дерева активных состояний при появлении событий. Формально эти правила будут изложены в следующих частях работы. Переход задается исходным состоянием, целевым состоянием, активирующим событием и защитным условием.

2.6. Действия

Действие — это набор простых операций, выполнение которых занимает время, значительно меньшее по сравнению со временем, характерным для системы. (В качестве такого времени можно использовать, например, среднюю величину интервала между событиями). Ограничение на время выполнения действия связано с тем, что рассматриваемая модель предполагает, что обработку события не могут производить несколько потоков одновременно. Если события будут приходить быстрее, чем они будут успевать обрабатываться, то возможен бесконечный рост очереди сообщений.

2.7. Действия при входе и при выходе

Действие при входе, приписанное некоторому состоянию — это действие, совершаемое системой в тот момент, когда это состояние становится активным. Аналогично, действие при выходе, приписанное некоторому состоянию — это действие, совершаемое системой в тот момент, когда это состояние перестает быть активным.

2.8. Действие при переходе

Действие при переходе — это действие, которое выполняется при выполнении перехода. Оно выполняется после действий при выходе и до действий при входе. Подробнее процесс выполнения перехода описан в следующих разделах работы.

3. Семантика

Описание семантики в рассматриваемом случае — это описание поведения системы при обработке события. В предлагаемой семантике при обработке события возможно выполнение одного или нескольких переходов. Поэтому рассмотрим сначала, как происходит выполнение одного перехода.

3.1. Выполнение перехода

Разделим выполнение перехода на четыре этапа. Пусть выполняется переход в состояние *A*.

Этап 1. Пойдем от состояния *A* вверх по дереву состояний до первого активного состояния. Пусть это состояние *B*.

Этап 2. Заметим, что состояние *B* обязано быть составным. Поэтому у него есть ровно один ребенок в дереве активных состояний. Отрежем всю ветку, соответствующую этому ребенку, выполнив предварительно действие при выходе для каждого состояния этой ветки, от детей к корню.

Этап 3. Выполним действие при переходе.

Этап 4. Подвесим новую ветку к состоянию *B*, дополним ее состояниями так, чтобы выполнялись условия для активных состояний. Выполним для всех новых состояний действия при входе, от корня к детям.

На основе изложенного строится алгоритм 1.

Алгоритм 1. Выполнение перехода

Дано: `Transition t` — переход, который нужно выполнить.

Надо: Выполнить переход.

```
void makeTransition(Transition t) {
  State b = t.target;
  List<State> l = new ArrayList<State>();
  while (!b.isActive()) {
    l.add(b);
    b = b.parent;
  }
  removeBranch(b.children[0]);
  t.action.run();
  addChainToTree(l, b);
  fillBranch(b.children[0]);
  startNewStates(b.children[0]);
}
```

3.2. Обработка события

Будем считать, что система обрабатывает внешние события по одному, причем за время обработки одного события новые внешние события не поступают. Будем действовать следующим образом: делегируем событие всем активным состояниям, по возможности выполним активированные при этом переходы, далее проделаем то же самое со всеми полученными в процессе обработки внутренними событиями. Действуем так, до тех пор, пока система не придет в стабильное состояние. Эта идея реализована в алгоритме 2.

Алгоритм 2. Обработка события

Дано: `Event e` — пришедшее событие.

Надо: Обработать событие.

```
Queue<Event> =
  new LinkedList<Event>()

Queue<Transition> tq =
  new LinkedList<Transition>()

void processEvent(Event e) {
```

```
  offerEvent(e);
  while (!eq.isEmpty()) {
    delegateEventToStates(eq.remove());
    while (!tq.isEmpty()) {
      Transition t = tq.remove();
      if (t.source.isActive()) {
        performTransition(t);
      }
    }
  }
}
void offerEvent(Event e) {
  eq.offer(e);
}
void offerTransition(Transition t) {
  tq.offer(t);
}
```

Алгоритм работает следующим образом. Формируются очереди событий и переходов. До тех пор, пока очередь событий не опустеет, события делегируются активным состояниям. Если в процессе обработки события состоянием активируется переход, то он добавляется в очередь вызовом метода `offerTransition`. Если было активировано внутреннее событие, то оно добавляется в очередь вызовом метода `offerEvent`.

Делегирование события состояниям в разных системах может происходить различным образом. Например, в некоторых системах может быть удобно обрабатывать события от корня к детям, в других — от детей к корню, в третьих — состояние обрабатывает только события, которые не были обработаны ни одним из его подсостояний.

4. Детерминированность систем

Если в системе никогда не активируется одновременно более одного перехода, то ее поведение легко предсказуемо. Однако, если активируется несколько переходов, встает вопрос о том, какие из них выполнить, и в каком порядке это делать. Эту проблему семантики решают по-разному. Поэтому некоторые системы могут вести себя неодинаково с точки зрения разных семантик [6]. По нашему мнению, эти проблемы связаны с тем, что существующие семантики при определении детерминированных и недетерминированных систем не учитывают характер действий, выполняющихся при переходах. Например, систему, изображенную на рис. 2, большинство семантик определит как детерминированную вне зависимости от того, какие действия выполняются на переходах. При этом порядок, в котором они выполняются, либо не определен совсем [2, 3], либо явно задается функцией приоритетов [4]. На наш взгляд, правильней было бы опираться на тип действий при определении детерминированности.

Будем называть детерминированными системы, при работе которых независимо от того, какой недетерминированный выбор сделает алгоритм, общий эффект от обработки события окажется одним и тем же. Под общим эффектом здесь понимается изменение дерева активных состояний и изменение переменных окружения.

Например, система на рис. 1 является детерминированной. Действительно, если система находится в состоянии (*D*, *F*) и пришло событие *e*, то активируются переходы из *D* в *E* и из *F* в *G*. Они могут быть выполнены в любом порядке. Однако независимо от порядка выполнения переходов, система переходит в состояние (*E*, *G*). Поэтому эта система детерминирована.

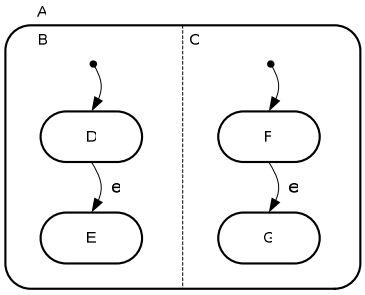

Рис. 1. Детерминированная система

Если при переходах выполняются действия z_1 и z_2 (рис. 2), то ее детерминированность существенно зависит от типа действий. Например, если действие z_1: {a = 0}, а действие z_2: {a = 1}, то такая система не является детерминированной, поскольку общий эффект действий зависит от порядка их выполнения.

Если же действие z_1: {a = 0}, а действие z_2: {b = 0}, то такая система является детерминированной, поскольку при любом порядке выполнения действий общий эффект остается одним и тем же: {a = 0, b = 0}.

Как показывает практика, большинство асинхронных реактивных систем могут быть описаны в предложенной модели таким образом, чтобы быть детерминированными в указанном смысле.

Недетерминированные системы часто возникают в синхронных системах — в системах, которые опрашивают внешние переменные с некоторым интервалом времени, и, в зависимости от их значений, делают переходы и выполняют действия. В рассматриваемой модели такие системы описываются с помощью введения одного события (назовем его `t`), которое будет посылаться системе на обработку через некоторый интервал времени.

Например, если у часовой бомбы, диаграмма состояний которой изображена на рис. 3, находящейся в состоянии «Детонация», кнопка «Отмена» будет нажата в тот момент, когда закончится время, то, в зависимости от того, какой переход выполнится первым, эффект может быть различным.

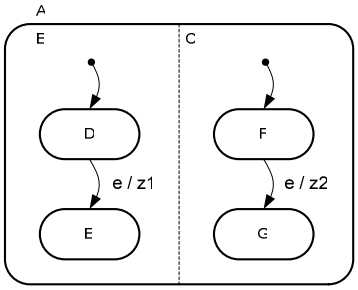

Рис. 2. Детерминированность системы зависит от типа действий

На практике, как правило, такие системы либо приводят к детерминированному виду (например, так, как это сделано на рис. 4), либо вообще оставляют это без внимания, считая, что при нескольких возможных вариантах, все они являются удовлетворительными. При этом поведение будет зависеть от конкретной реализации.

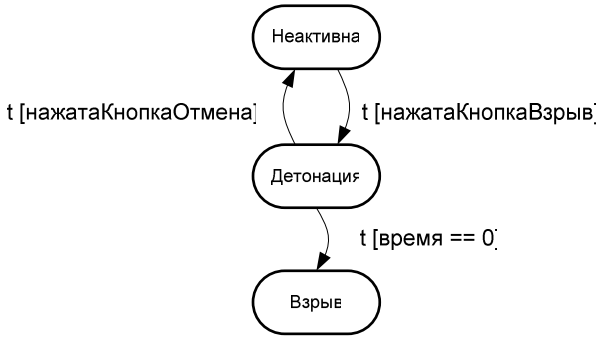

Рис. 3. Недетерминированная система

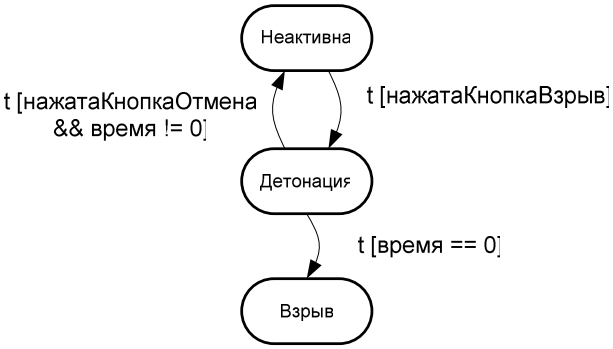

Рис. 4. Детерминированная версия системы

5. Заключение

Отметим основные достоинства предложенной семантики.

1. Семантика изложена формально, что исключает неоднозначность ее трактовки.
2. Она позволяет определить поведение системы в ситуациях, когда событие активирует сразу несколько переходов.
3. При определении детерминированности системы учитывается характер действий, выполняющихся при переходах. Это позволяет разделить системы на детерминированные и недетерминированные по-новому.
4. Семантика изложена в виде достаточно простых алгоритмов, что облегчает ее использование на практике.

В заключение отметим, что благодаря описанной семантике стало возможным использование диаграмм состояний при моделировании жизненного цикла компоненты сложного программного комплекса [7]. Применение диаграмм состояний вместо классических диаграмм переходов позволило уменьшить число состояний с 25 до 15, а число переходов — с 50 до 20. Это позволяет надеяться, что предложенная семантика может быть эффективно использована для решения подобных задач и при разработке других программ.

Источники

[1] *Harel D.* “Statecharts: A Visual Formalism for Complex Systems”, *Science of Computer Programming,* vol. 8, pp. 231–274, June, 1987.

[2] *Harel D., Naamad A.* “The STATEMATE semantics of statecharts” *ACM Transactions on Software Engineering and Methodology,* vol. 5, pp. 293–333, 1996.

[3] *Harel D., Kugler H.* “The RHAPSODY Semantics of Statecharts (or, On the Executable Core of the UML)” *Integration of Software Specification Techniques for Application in Engineering,* pp. 325–354, 2004.

[4] *Wasowski A., Sestoft P.* “On the Formal Semantics of VisualSTATE Statecharts”

http://citeseer.ist.psu.edu/wasowski02formal.html

[5] *Гуров В. С., Мазин М. А., Шалыто А. А.* “Операционная семантика UML-диаграмм состояний в программном пакете UniMod”, *Материалы XII Всероссийской научно-методической конференции "Телематика-2005".* СПбГУ ИТМО, 2005.

http://tm.ifmo.ru/tm2005/src/224as.pdf

[6] *Crane M. L., Dingel J.* “UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal.” *MoDELS,* pp. 97–112, 2005.

[7] *Маврин П. Ю., Корнеев Г. А., Станкевич А. С., Шалыто А. А.* “Моделирование жизненного цикла компоненты программного комплекса с использованием диаграмм состояний”, опубликовано на сайте http://is.ifmo.ru/, в разделе "Статьи".