Лекция № 11. Анатомия языков визуального моделирования.

Основная задача данной лекции – собрать во едино и изложить всю теоретическую информацию, необходимую для разработки проблемно-ориентированных визуальных языков (domain-specific languages). Сначала мы обсуждаем общий, семиотический подход к обработке и изучению информации, потом уточняем его для визуальных языков, а после этого рассматриваем пример небольшого визуального языка.

Языки: синтаксис, семантика и прагматика. Прежде чем говорить о визуальных языках, рассмотрим строение произвольного языка. Определим язык как произвольную систему знаков, с помощью которой некоторый пользователь может составлять определенные описания (тексты, модели и т.д.) какого-то фрагмента реальности – см. рис. 11.1. Этот фрагмент реальности будем называть предметной областью языка.

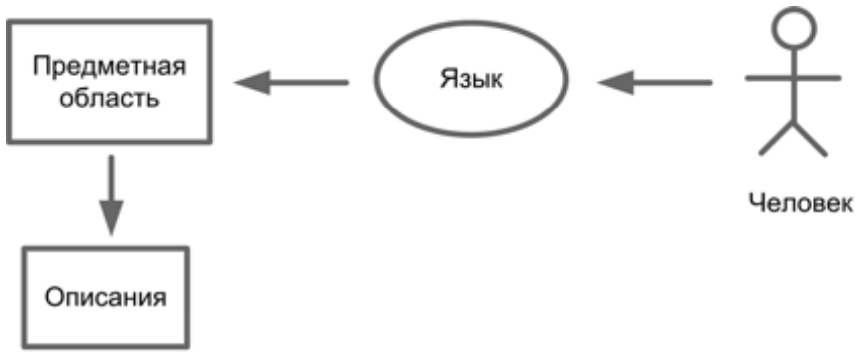


Рис. 11.1.

Созданные с помощью языка описания (зафиксированные на каких-либо материальных носителях – бумаге, магнитофонной ленте и т.д., либо же передаваемые сразу в момент создания, как, происходит с человеческой речью) служат для общения людей, например:

- в рамках какой-либо совместной деятельности, например, юридический язык – множество специальных терминов, определенная трактовка слов обычного язык, специальный способ конструировать предложения, высказывания, аргументы и тексты и, естественно, определенный круг событий, поступков и отношений людей (то есть та самая предметная область), где все это применяется;
- для передачи и накопления знаний, например, научные тексты, научный язык – перед глазами сразу встают полки, уходящие за горизонт с многочисленными фолиантами, созданными за столетия развития науки, а также всевозможные научные статьи, выступления ученых на симпозиумах и конференциях, просто беседы; во всех этих случаях используются весьма специфичные обороты речи, представляющие определенное мировоззрение и мировосприятие, научным текстам свойственна определенная структура и логика;
- для запечатлевания и передачи человеческих чувств, переживаний, впечатлений – например, язык живописи, музыкальный язык, язык классических художественных произведений (поэзия, романы, пьесы).

Описания, созданные с помощью некоторого языка, могут быть использованы также для автоматической обработки различными, созданными человеком, устройствами и машинами, например, тексты на языках программирования предназначены для исполнения вычислительными машинами.

Языки и соответствующие им описания обозначаю целые миры человеческой жизни. Каждый полноценный мир имеет свой язык, свои описания. Существует специальная наука, которая изучает миры человека на основе изучения соответствующих языков и текстов – семиотика.

Семиотика определяет следующие измерения произвольного языка: синтаксис (syntax), семантика (semantic) и прагматика (pragmatic) языка.

Синтаксис – это знаки языка, а также правила, по которым с их помощью составляются тексты на этом языке. Например, музыкальные звуки и правила построения музыкальных произведений. Или правила создания текстов программ с помощью определенного языка программирования: допустимые лексемы, разбиение их на идентификаторы, ключевые слова языка и служебные символы, правила компоновки лексем в корректные предложения языка и правила оформления корректных программ (наличие нужных заголовков, скобок и пр.). Синтаксис явдляется наиболее просто формализуемой частью языка – синтаксические правила естественных языков, правила нотной записи, синтаксис языков программирования и т.д. Однако уровни формализации могут сильно отличаться - например, вряд ли можно придумать синтаксис научного или юридического языка. В этом случае синтаксис является практическим навыком, а не точной спецификацией набора правил.

Семантика – это то, как знаки языка соотносятся с объектами предметной области, которую язык описывает. Например, музыкальные фразы и произведения в целом описывают человеческие чувства, а конструкции языка программирования – определенные атомарные действия вычислительной машины (описание переменных, присваивание им определенных значений, вызов процедуры и т.д.), то есть имеют *исполняемую семантику*.

Наконец, *прагматика* – это правила и способы использования языка человеком, которые выражают назначение языка, то, как и зачем языком пользоваться. Для музыкального языка к прагматике относится все, что связано с воспроизведением музыкальных произведений – голосом или с помощью музыкальных инструментов, по слуху или с помощью нотных записей, а также определенная развитость души слушателей. Для языков программирования прагматикой являются правила и приемы программирования, а также все инструментальные средства (то есть среды разработки, которые обозначаются термином integrated development environment (IDE)), помогающие составлять программы и переводить их в исполняемый машиной код.

Связь синтаксиса, семантики и прагматики с языком, его пользователем и предметной области представлена на рис 11.2.

Рис. 11.2.

Синтаксис, семантика и прагматика визуальных языков. Теперь обратимся к языкам визуального моделирования.

Предметной областью для них является программное обеспечение или контекст системы. В целом, задача визуальных языков – улучшить общение в программных проектах между всеми так или иначе задействованными в этом процессе людьми: программистами, инженерами, заказчиком, будущими пользователями системы, менеджерами различных рангов и т.д. Соответственно, все эти люди являются *пользователями* этих языков. Кроме того, в разряд пользователей необходимо причислить также вычислительные средства – для многих визуальных спецификаций автоматически генерируется исполняемый код.

Теперь поговорим о синтаксисе, семантике и прагматике визуальных языков. *Син таксис* визуальных языков разбивается на:

- *конкретный* (concrete) – правила изображения символов языка, из которых строятся визуальные модели;
- *абстрактный* (abstract) – структура визуальных спецификаций, в рамках которой тщательно классифицированы все графические символы, определены все их атрибуты и связи друг с другом, построена дополнительная система абстрактных понятий для связного и строгого описания конструкций языка;
- *служебный* (serialization) – способ хранения визуальных моделей; в случае с текстовыми языками служебный и конкретный синтаксис совпадают – как пишется так и хранится; в настоящее время, в связи с развитием компьютерных форм представления информации уровень представления часто отделяется от уровня хранения – это различные гипертекстовые форматы (например, html).

Семантика визуальных языков определяется двояко: с одной стороны, это абстракции разработки и функционирования ПО, например, случаи использования, состояния объектов, сообщения. С другой стороны, для многих таких конструкций существуют проекции в исполняемый программный код, тол етсь они имеют исполняемую семантику.

Прагматика визуальных языков – это средства их использования: методы анализа и проектирования ПО, основанные на визуальном моделировании, различные способы визуализации ПО и пр. Кроме того, в прагматику попадают также инструментальные средства, помогающие использовать визуальные языки в промышленных проектах.

Отметим, что мы здесь представили семиотический подход к визуальном моделированию, то есть поставили в центр язык, остальное же (методы и программные средства) сделали его атрибутами. Этот подход удобен с методической точки зрения. Однако на практике он менее удобен по следующим причинам:

- разработчики пользуются, скорее, не языками, а средствами визуального моделирования, включающими, кроме языков еще также соответствующие методы и программные средства; в частности, в рамках DSM-подхода требуются именно целевые DSM-средства, а не просто новые языки; другое дело, что, как, например, в Microsoft DSL Tools, DSM-средства автоматически генерируются по описанию языка, однако так бывает далеко не всегда;
- визуальное моделирование часто оказывается составной частью более общего метода или инструмента – например, анализа и проектирования ПО – и в этом и подобных случаях не фигурирует как отдельная практика; существенно важнее оказывается выстроить сам процесс моделирования, общение с заказчиком и пр., а визуальные модели лишь помогают в этом.

Пример визуального языка. Мы уже изучали визуальные языки, например, UML, BPMN. Однако эти языки слишком большие и сложные, чтобы на их примере учиься создавать собственные визуальные языки. Поэтому мы рассмотрим небоьшой, игрушечный визуальный язык, который позволяет описать множество классов с атрибутами и методами, набор сообщений, которым могут обмениваться экземпляры классов, а также диаграммы состояний и переходов, описывающие поведение каждого класса. На рис. 11.3 – 11.4 представлена небольшая модель, созданная с помощью этого языка.

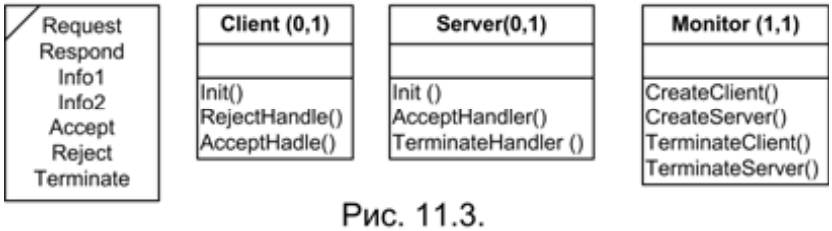


Рис. 11.3.

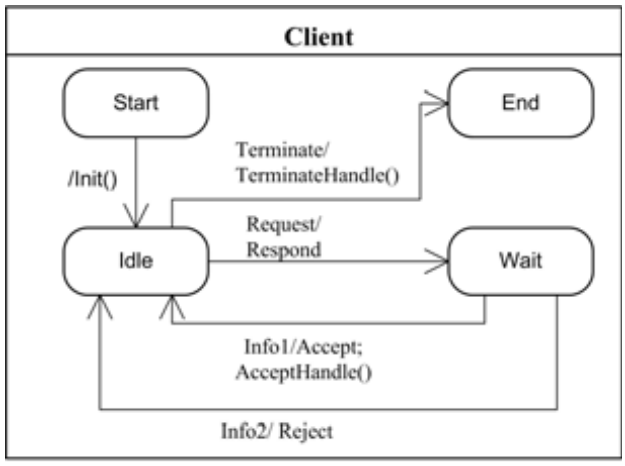


Рис. 11.4.

Мы видим список сигналов, а также три класса – Client, Server и Monitor. Экземпляры этих классов могут взаимодействовать друг с другом через посылку/прием сигналов из списка, схемы их поведения определяется в классах, с помощью конечных автоматов.

В целом, поведение системы, представленной в этом примере, выглядит так. Монитор (экземпляр класса Monitor) создается автоматически, при запуске всей системы, о чем свидетельствует первый параметр, равный 1, после имени класса. В этой нашей системе может существовать только один такой объект, на что указывает значение второго параметра после имени класса, равное 1. Монитор создает экземпляр класса Server (далее – сервер) и экземпляр класса Client (далее – клиент). В этой нашей небольшой игрушечной системе может существовать не более одного клиента и сервера, на что указывают соответствующие параметры после имен этих классов. Далее рассмотрим поведение сервера, представленное на рис. 11.4.

После своего создания мониторов сервер оказывается в состоянии Start, вызывает свою процедуру Init(), которая выполняет все служебные действия по инициализации сервера. Далее он переходит в состояние Idle, в котором готов обслужить запрос клиента. При получении такого запроса (сигнал Request), сервер оповещает клиента с помощью сигнала Respond о том, что запрос до него дошел и требуется дополнительная информация о клиенте. Клиент посылает либо сигнал Info1 либо сигнал Info2. В зависимости от этого сервер или принимает запрос на обработку, или нет. В первом случае он посылает клиенту сигнал Accept и вызывает свою процедуру-обработчик запроса AcceptHandle(), во втором случае он посылает клиенту сигнал Reject. В обоих случая сервер переходит в состояние Idle и готов обрабатывать следующие запросы. В этом же состоянии сервер может обработать сигнал монитора Terminate, означающий необходимость прекращения функционирования и корректного завершения работы (процедура TerminateHandle). После этого сервер переходит в состояние End и окончательно завершается. Нетрудно продолжит этот пример, дописав конечные автоматы для клиента и монитора.

Наверное, про этот язык понятно все или почти все прямо из приведенного выше примера. Однако, так происходит потому, что этот язык очень прост. Если мы добавим связи между классами, параметры сигналов, ветвления в переходах и т.д., то язык бы существенно усложнился и возникла бы реальная потребность в том, чтобы создать его точное описание – определить синтаксис, семантику и что не мало важно – прагматику, которая обеспечила бы практическое использование языка.

Опишем наш язык формально. Начнем с синтаксиса, точнее, с абстрактного синтаксиса.

Абстрактный синтаксис. Существует два широко распространенных способа для описания абстракного синтаксиса – грамматика в форме Бэкуса-Науэра и метамодели. Первый способ пришел из практики описания синтаксиса языков программирования (теперь и уже давно почти все языки программирования – Java, C3, C++ и пр. – описываются таким способом), с помощью второго способа описывается синтаксис языка UML, а также некоторых других языков моделирования.

Рассмотрим сначала первый способ. Не вдаваясь в подробности теории формальных грамматик, не будем здесь давать точные определения. Представим лишь объяснения, достаточные для того, чтобы понимать спецификации языков, выполненные с помощью грамматик в форме Бэкуса-Науэра, а также составлять такие спецификации.

Такие грамматики определяют структуру формальных текстов (программ, визуальных моделей и т.д.). Эта структура должна быть строго определенной для того, чтобы позволить дальнейшую формальную обработку текстов, в частности, генерацию исполняемого кода. То есть любой текст, созданный с помощью этого языка, должен иметь структуру, определяемую грамматикой. Структура текста определяется иерархически. В случае с языком из нашего примера, любая спецификация (в нашем случае мы называем ее моделью) должна состоять из списка сигналов и набора классов:

<model> :: = <signal list> <class>+<model_name>

Список сигналов обозначается как <signal_list > и должен быть строго один во всей модели, а классов может быть сколько угодно, но обязательно должен быть хоть один, что изображается значком + рядом с элементом грамматики <class>.

Далее рассмотрим, как устроен список сигналов. Он состоит из списка строк, обозначающих сигналы, которые могут быть посланы и получены объектами классов данной модели:

<signal_list> :: = <signal>+
<signal> :: = <signal name>

В случае с <signal name> и <model_name> мы добрались до конца в процессе нисходящей детализации элементов языка, то есть эти элементы являются *терминалами* и в рамках нашей грамматики не имеет составных частей. Те же элементы грамматики, которые подвергаются дальнейшей декомпозиции, называются *нетерминалами*. Сказанное выше продемонстрировано на рис. 11.5.

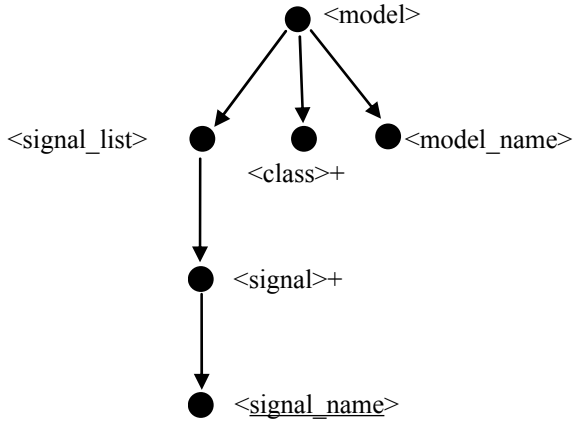


Рис. 11.5.

В нашей грамматике все терминалы, являющиеся произвольными строками-именами (их принято называть *идентификаторами*), обозначаются как обычные нетерминалы, но с добавлением подчеркивания имени.

Далее рассмотрим конструкцию <class>:

<class> :: = <class name> <parametrs> <method>* [<statechart>] <attribute>*

Мы видим, что класс состоит из имени (<class name>), параметров (<parametrs>) набора методов (<method>) и атрибутов (<attribute>), а также диаграммы состояний и переходов (<statechart>). Сразу за нетерминалами <method> и <attribute> следует символ *, который указывает на то, что как методов, так и атрибутов в классе может быть произвольное количество, в том числе и не быть вовсе. Последняя оговорка отличает символ * от +. Нетерминал <statechart> взят в квадратные скобки, что означает, что у класса может не быть диаграммы состояний и переходов.

У класса может есть два параметра, следующих в круглых скобках за его именем. Первый параметр указывает на то, сколько экземпляров класса создается при запуске системы, второй – сколько экземпляров класса одновременно может существовать в системе:

<parametrs>::= (<init>, <num>)

Отметим, что круглые скобки и запятая являются новым видом терминальных элементов, в отличие от угловых и квадратных скобок, которые являются синтаксисом самой грамматики. Круглые скобки и другие подобные элементы появляются в связи тем, что графические символы в нашем языке могут быть «нагружены» текстом, который имеет более сложную структуру, чем просто имя.

Конечный автомат включает в себя имя класса, к которому он относится (<class_ref>), а также набор состояний (<state>+):

<statechart> ::= <class_ref><state>+

Прокомментируем элемент <class_ref>. В нашей грамматике он также является терминалом, но отличающимся от идентификаторов тем, что вместо него подставляется не произвольная трока символов, но одна из тех, которая именует один из классов данной модели. Ведь диаграмма состояний и переходов изображается отдельно от класса, к которому она относится. Их связь происходит через ссылку[1], обозначаемую конструкцией <class_ref>. Будем называть такие терминалы ссылками, отличая их от других элементов грамматики тем, что их имена оканчиваются на ref.

Состояние устроено так:

<state> ::= <state name> <transition>*

Видно, что состояние содержит в себе имя, а также все исходящие переходы, которые переводят объекты данного класса из этого состяония в другие.

Переход, в свою очередь, устроен так:

<transition> :: = [<input>]/<action> {; <action>}* <target_state_ref>

Он инициируется определенным сигналом, который принимается и обрабатывается данным объектом в этом состоянии (конструкция <input>), содержит в себе ряд действий (<action>) и завершается новым состоянием, в которое объект переходит, обработав данный входной сигнал (конструкция <target_state>). В конструкции <transition> содержатся два терминала нового типа, которые нам еще не встречались. Это значок ‘/’, который отделяет входной сигнал от действий по его обработке и символ ‘;’ , разделяющий действия в переходе в том случае, если их более одного. Эти терминалы относятся к тому же типу, что и круглые скобки. Отметим, что фигурные скобки, также как и квадратные, являются частью грамматики. Они руппируют аргумент для опервайии *: ведь если у нас появляется следующее действие, то оно должно быть отделено символом ‘;’ от предыдущего действия, в то время как последнее действие не должно иметь после себя этот разделитель.

Действие в переходе может быть либо посылкой сигнала другому объекту, либо вызовом метода данного объекта:

<action> :: = <metod_invocation>| <send>

Значок | нашей грамматики обозначает эту альтернативу. Ниже представлены описания остальных элементов, которые не должны вызывать затруднение в понимании, и поэтому они оставляются без комментариев:

[1] Пользуясь терминологией языка C++ можно сказать, что это ссылка по имени.

<method> :: = < method name>()
<attribute> ::= <attribute_name> <attribute_type>
<attribute_type> :: = <attribute_type_name>

<target_state> :: = <state_ref>
<send>::= <signal_ref>
<input>::= <signal_ref>
< method_invocation > :: = <method_ref>()

Целиком грамматика нашего языка выглядит так:

<model> :: = <model_name><signal list> <class>+
<class> :: = <class name> <method>* [<statechart >] <attribute>*
<parametrs>::= (<init>, <num>)
< statechart > ::= <class_ref><state>+
<state> ::= <state name> <transition>*
<transition> :: = [<input>]/<action> (; <action>)* <target_state_ref>
<action> :: = <method_invocation>| <send>
<method> :: = < method name>()
<attribute> ::= <attribute_name> <attribute_type>
<attribute_type> :: = <attribute_type_name>
<target_state> :: = <state_ref>
<send>::= <signal_ref>
<input>::= <signal_ref>
< method_invocation > :: = <method_ref>()

На рис. 11.6 представлена метамодель, описывающая этот же язык.

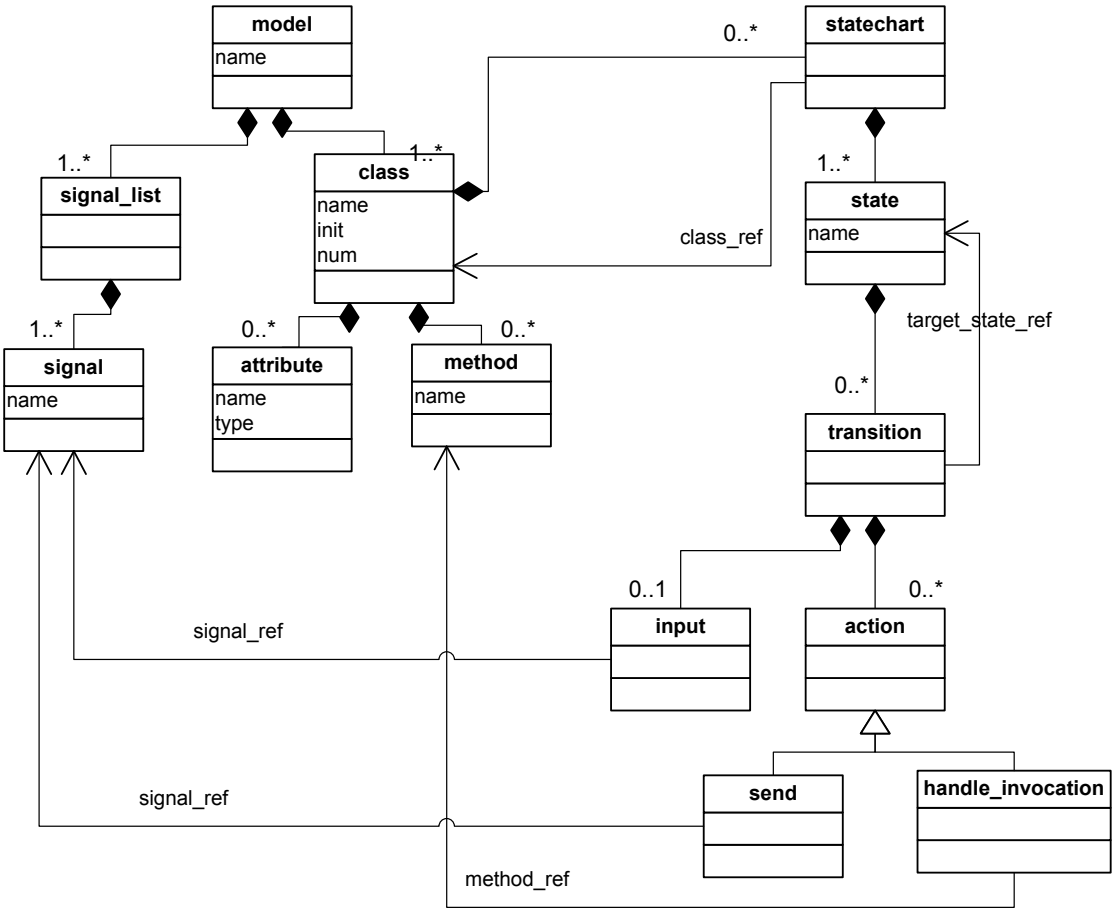


Рис. 11.6

Все нетерминалы грамматики превратились в классы, отношение включения одних нетерминалов в другие показано через отношение агрегирования, операторы *, +, [ ] выражены через множественность агрегирования – 0.., 1.., 0..1 соответственно. Ссылки выражены направленными ассоциациями, а идентификаторы – атрибутами классов с именем name. Наконец, оператор альтернативы ‘|’ выражен абстрактным классом, который имеет несколько альтернативных наследников.

Теперь поговорим о том, что не вошло ни в грамматику, ни в метамодель, хотя по смыслу относится именно к абстрактному синтаксису:

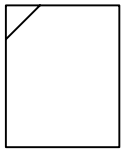
1. Методы, вызываеваемые из переходов автомата, должны обязательно быть методами этого класса.
2. Не может быть пустых классов, не имеющих ни методов, ни атрибутов, ни конечного автомата.
3. Параметры класса являются целыми неотрицательными числами, причем первый параметр меньше либо равен второму.

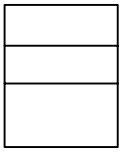
Попытка выразить эту информацию существенно усложнила бы формальное описание абстрактного синтаксиса. Но эта информация нужна, поэтому она присутствует в виде дополнительных ограничений. Эти ограничения можно выразить с помощью декларативного языка OCL, котрый является частью стандарта UML.

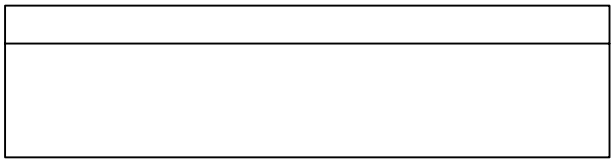
Конкретный синтаксис. Теперь приступим к описанию конкретного синтаксиса. Нам нужно:

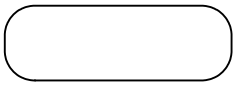
- определить внешний вид всех графических символов;
- задать правила расположения текста относительно графических символов;
- задать поведение этих графических символов: как ломаются линии, как они цепляются к узлам, как растягиваются графические узлы, а также поведение текста внутри них;
- задать «диаграмность» спецификаций: сколько видов диаграмм может быть, какие элементы модели могут владеть диаграммами, поддерживается ли многостраничность диаграмм и как при этом и какие элементы могут «загружаться» на разные диаграммы.

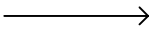
В нашем языке существуют следующие графические символы:

<signal_list_symbol> :: = 

<class_symbol > :: = 

<statechart_symbol > :: = 

<state_symbol> :: = 

<transition_symbol> :: = 

Можно усложнит грамматику абстрактного синтаксиса и включить в нее описание графических терминалов. Пример такой грамматики представлен в [2]. Там обычная грамматика в форме Бэкуса-Науэра дополнена следующими конструкциями:

- **contains –** левый графический символ содержит правый графический символ;
- **is associated with** правый графический символ должен быть ближе к левому, чем любой другой символ на диаграмме; используется для привязки надписей к линиям, к границам прямоугольников, а также для связи одной графической фигуры с другой (та, другая должна быть соединена с линией, один конец которой свободен);
- **is followed by –** за первой фигурой (левый аргумент) следует вторая (правый аргумент), связаны они направленной линией;
- **is connected to –** первый (левый) аргумент соединяется с правым, причем один из них - линия;
- **set –** постфиксный оператор обозначающий что, графические символы, являющиеся его аргументами, могут располагаться на диаграмме в произвольном порядке.

Так, определение диаграммы классов может выглядеть следующим образом:

<class_diagram> ::= {<signal_list> <class>+} **set**

Это означает, что на диаграмме должен быть строго один список сигналов и произвольное количество (в том числен и нулевое) классов.

Определение класса выглядело бы так:

<class> ::= <class_symbol> **contains** { <class_name><parametrs> <attribute_area><method_area>} [<statechart>] * }

Следующая последовательность утверждений грамматики определяет состояние.

<state> ::= <state_symbol> **contains** <state name> **is associated with** <input area>

Эта строка означает, что символ состояния с надписью внутри (обозначает имя состояния) относительно графического агрегата <input area>.находится ближе всех к последнему.

<input area>::= {[<input>]/<action> {; <action>}*}} **is associated with** <transition area>

Эта строка определяет графический агрегат <input area> как текстовая надпись, «прилепленная» к графическому агрегату <transition area>.

<transition area>::= <transition_symbol> **is connected to** <next_state_area>

Эта строка определяет графический агрегат <transition area> как линию, конец которой присоединен к графическому агрегату <next_state_area>.

<next_state_area> ::= <state_symbol> **contains** <state name>

Тут определяется графический агрегат <next_state_area>.как символ состояния с текстовым именем внутри.

Мы не стали пользоваться этим способом определить конкретный синтаксис языка, поскольку, во-первых, он сильно усложняет грамматику, во-вторых, ничего не говорит о том, как именно располагаются надписи на линиях, текст в фигурах, опускается также разбиение на страницы (в [2] указывается, что разбинение на тсраницы не вошдло в SDL/GR) и т.д.

Определим положение текста относительно графических символов нашего языка неформально. Как видно из рис. 11.3, класс разделен на три секции. Его имя вместе с параметрами располагается по центру первой, жирным шрифтом (Arial 12). В следующей секции, с выравниванием слева, располагаются в столбик атрибуты, а в следующей секции – также методы (и методы и атрибуты изображаются шрифтом Arial 10). Класс может отображаться без секции атрибутов и/или без секции методов, несмотря на то, что обе они могут быть не пусты.

Все сигналы располагаются в секции сигналов в столбик, таким же шрифтом, как и методы класса, с выравниваем посередине.

Имена состояний изображаются таким же шрифтом, что и имена классов, и находятся по центру символа состояния.

Линия перехода может быть прямой или ломаной вертикально-горизонтальной. Линия перехода изображается от состояния-источника стрелкой к тому состоянию, в которое объект переходит. Текст, относящийся к переходу, по умолчанию располагается по центру линии-перехода, как показано на рис. 11.7. Предполагается, что этот текст можно «отрывать» от его стандартного положения и перемещать в любое место диаграммы. Однако при перемещении «центра» линии перехода, надпись меняет свое положение в соответствии, сохраняя свои координаты относительно этой точки.

Рис. 11.7.

Теперь определим поведение графических символов. Про то, как ломаются линии мы уже сказали. К фигурам они должны цепляться в любой точке. Сами фигуры могут растягиваться по вертикале и горизонтали, как симметрично, так и по каждому измерению в отдельности. При этом текст внутри не меняет шрифта и центрирования.

Про диаграммность. Одной модели может соответствовать более одной диаграммы классов. Для всех них должны подразумеваются операции добавить/загрузить, удалить/выгрузить, применяемые к классам и списку сигналов. Каждый автомат может располагаться более чем на одной диаграмме. Те же операции добавить/загрузить, удалить/выгрузить применимы к состояниям и отдельным переходам. Для последних – только в случае наличия на диаграмме состояния-источника. Если целевое состояние для загружаемого перехода отсутствует на диаграмме, то оно загружается. При выгрузке перехода целевое состояние остается на диаграмме. Состояние загружается на диаграмму без своих переходов. Последние загружаются на диаграмму последними.

В Microsoft Visio большая часть конкретного синтаксиса определяется с помощью специального декларативного языка, в Microsoft DSL Tools для этого используются многочисленные средства – выбор соответствующих графических классов, задание их свойств, задание форматирования текста в графических символах с помощью вставок на #C (используя механизм частичных классов). Нужно отметить, что в Microsoft Visio имеется большая свобода в задании конкретного синтаксиса и почти все предложенные выше требования могут быть реализованы. В Microsoft DSL Tools эти возможности очень скромные. Выше мы описали очень сильные требования к конкретному синтаксису – не то что DSM-платформы, но и готовые целевые DSM-пакеты и промышленные средства визуального моделирования редко поддерживают их все. Однако, по нашему опыту, предоставление этих возможностей конечному пользователю позволяет ему создавать красивые диаграммы, что важно.

Описание семантики. Перейдем теперь к описанию семантики нашего языка. Ядро исполняемой семантики составляет механизм приема/посылки сигналов. Существуют глобальные часы и глобальна очередь. Через равные промежутки времени глобальных часов происходит отправление следующего сигнала из очереди на обработку готовому к его приему объекту. Таким является объект, который находится в состоянии, в котором предусмотрена обработка данного сигнала. Если таких объектов несколько, то выбирается единственный произвольным способом (то есть алгоритм выбора не специфицируется семантикой языка). Перед отправлением сигнала на обработку список текущих состояний, в которых пребывают объекты, обновляется. Объект может находиться не только в состоянии, но и в переходе. Тогда он не готов к приему сигналов. Переход может обрабатываться произвольное количество времени по абсолютным часам. у имеющихся в системе объектов.

Все объекты «живут» параллельно. Детали этого параллелизма семантикой не определяются. Это может быть псевдопараллельная модель (все отрабатывают по одному переходу по некоторой очереди, или объекты действительно параллельны).

Предполагается, что код методов классов описыватся на языке C# и не входит в наш язык. Предполагается также, что типы атрибутов описываются в соответствии с тем,

как это принято в языке C#. Это же язык является целевым для генерации кода по модели, выполненной с использованием нашего языка.

Прагматика. Назначение нашего языка – учебное. Предполагается, что его пользователи – это слушатели данного курса лекций, а также студенты-исследователи, изучающие визуальное моделирование. В следующей лекции мы покажем, как этот язык будет реализован в рамках Microsoft DSL Tools.

Основная литература

1. Jack Greenfield, Keith Short et al. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. – John Wiley & Sons. 2004. – 666 p.

Дополнительная литература

2. ITU Recommendation Z.100: Specification and Description Language. 08/2002. – 206 p.
3. В.Л.Авербух. К теории компьютерной визуализации. http://cv.imm.uran.ru/articles/cvtheory_w23print.pdf