В.В. Артюхин

CASE-технологии создания имитационных моделей в среде *Pilgrim 5*

Имитационному моделированию поддаются практически все процессы независимо от их природы, будь она экономической, физической, социальной или любой другой. Система Pilgrim 5 давно завоевала популярность как мощная и гибкая среда имитационного моделирования. В нынешнее время требуются новые удобные инструменты, позволяющие создавать компьютерные модели не только профессионалам в области информационных технологий, но и специалистам в других областях.

Концепция

Английская аббревиатура «CASE» расшифровывается как Computer Aided Software Engineering. На русский язык это дословно переводится как «разработка программного обеспечения с помощью компьютера», а термином «CASE-средства» обычно называют разнообразные программы или модули, которые облегчают пользователю жизнь при решении его насущных задач на персональном компьютере и делают его работу более удобной и комфортной. В частности, именно к этому классу программного обеспечения относятся:

- мастера, т.е. модули более крупных программ или отдельные программы, позволяющие пользователю произвести любое сложное действие посредством выбора ряда опций за несколько шагов, и обычно реализуемые в виде последовательности диалоговых окон, часто содержащих кнопки «Вперед», «Назад», «Завершить». Мастера обычно используются в операционных системах семейства *Microsoft Windows*;
- графические конструкторы, т.е. программные средства, помогающие пользователю создать некий сложный документ с помощью автоматизированной обработки схемы, рисунка, таблицы или чего-либо подобного. Как правило, конструкторы реализуются в виде отдельных программных продуктов. В качестве примера можно привести пакет *Kinetix 3D Studio MAX*, позволяющую создавать трехмерные изображения и анимацию без программирования и сохранять результаты в разных форматах. Вообще называть *3D Studio MAX* конструктором довольно опрометчиво — это очень мощная система трехмерной визуализации, но, тем не менее, по сути — это графический конструктор.

Понятие «графический конструктор» довольно широкое (как и все, рассматриваемые в данной статье). Очевидно, что термин «конструктор», означает нечто, способное преобразовывать и компоновать некоторые составляющие части для получения осмысленного целого. В нашем случае в роли этого «нечто» выступает программное обеспечение, проверяющее корректность и преобразующее информацию, полученную от пользователя (путем удобным для него) в иной формат, пригодный для дальнейшей обработки (но уже не настолько понятный пользователю и пригодный для визуального анализа). Таким образом, конструктор является промежуточным звеном между пользователем и программой обработки некоторой информации или целевой программой (рис. 1). Термин «графический» в данном случае означает, что информация, которую вводит пользователь, имеет графический характер (полностью или, как правило, частично).

Существуют и другие разновидности CASE-средств, но когда мы говорим о CASE-технологии в данной статье, то имеем в виду

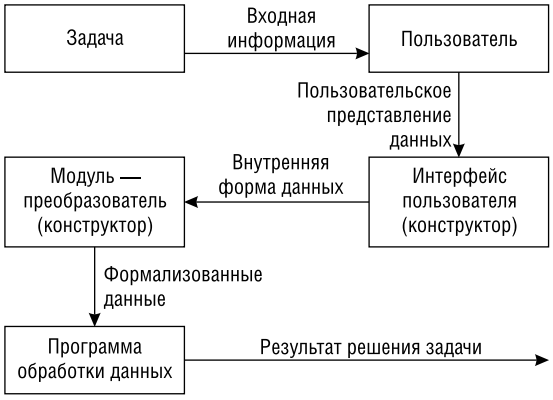

Рис. 1. Процесс решения задачи с помощью графического конструктора

разработку имитационных моделей с помощью совершенно конкретного продукта — графического конструктора для среды имитационного моделирования *Pilgrim 5*.

Чтобы понять степень необходимости графического конструктора для среды *Pilgrim 5*, придется обратиться к технической стороне его реализации.

Сама среда *Pilgrim 5* для операционной системы *Windows* программно реализована в виде библиотеки статической компоновки pilgrim.lib, файла ресурсов pilgrim.res и двух заголовочных файлов на языке программирования C++ — pilgrim.h и simulate.h. Файл модели в этой среде представлен исходным файлом программы на языке C++. Файл одной из простейших моделей выглядит следующим образом:

```
//Стол бухгалтера
#include "Pilgrim.h"
forward
{
   //Глобальные параметры модели
   modbeg("Стол бухгалтера",10,1000,
          (long)time(NULL),none,
          4,none,6,2);
   //Генератор
   ag("Документы",1,1,unif,5,2,0,3);
   network(dummy,dummy)
   {
     top(2):
          queue("Очередь документов",none,3);
          place;
     top(3):
          serv("Бухгалтер",1,none,norm,3,3,0,4);
          place;
     top(4):
          term("Корзина");
          place;
     fault(123);
   }
   modend("Model.rep",1,8,none);
   return 0;
}
```

Без применения конструктора пользователь (в текстовом редакторе) вводит текст модели вручную с использованием конструкций *Pilgrim 5* и функций, таких как serv(...), attach(...) и других, представляющих узлы модели. Конструкции определены в заголовочных файлах, функции описаны в тех же заголовочных файлах и определены в файле статической библиотеки.

Далее модель транслируется с помощью обычного компилятора C++ и на выходе получается исполняемый программный файл с расширением .exe, выполнив который в операционной системе семейства *Microsoft Windows*, пользователь может запускать модель, осуществлять анализ задержек в очередях, динамики потока, загрузки узлов, результатов в файле отчета (*.rep) и трассировку модели. После компиляции модель не требует дополнительных файлов (таких как динамические библиотеки *.dll) для своей работы.

Таким образом, *Pilgrim 5* представляет собой не просто систему имитационного моделирования, а систему, реализованную в виде того, что часто называют фреймворк (набор функций, классов или иных компонент одинаковой архитектуры, предназначенных для упрощения создания программ определенной узкой направленности). Такая организация системы весьма удобна для программистов — составителей моделей, знакомых с языком C++. Хотя и здесь часто возникают сложности, поскольку благодаря множеству макроопределений в за-

головочных файлах исходный код для программы-модели совсем не похож на обычный текст *Windows*-программы, кроме того, *Pilgrim* предъявляет к тексту модели дополнительные требования, не проверяемые компилятором.

Как же быть экономистам, финансистам или иным специалистам, которые могут извлечь ощутимую выгоду из имитационного моделирования, но которые не встречались с этим языком или не изучали программирование вообще?

Ситуация в значительной степени усложняется, если усложняется сама модель. Например, в ситуациях, когда один узел имеет несколько выходов, требуется написание достаточно сложных конструкций ветвления if языка C++ и составитель модели должен точно знать, как их составлять и где помещать, т. е. автор модели должен не только изучить синтаксис языка C++, но и знать ряд фундаментальных техник программирования, что среди специалистов в области экономики случается не слишком часто. Как правило, проблемы начинаются еще на стадии обучения работе с системой.

Кроме того, поскольку модель в своем отредактированном и готовом к компиляции виде представляет собой не что иное, как текстовый файл и не несет в себе никакой понятной графической информации (например, отражающей топологию сети модели), пользователь еще до начала редактирования модели должен совершенно ясно представлять себе ее структуру: количество и типы узлов, их параметры, переходы между ними и то, в каких случаях происходят эти переходы. На практике это означает предварительное применение таких «технических» средств, как карандаш и бумага или, в лучшем случае, какого-либо постороннего графического редактора. И если вдруг пользователь хочет изменить что-либо в сети модели, то часто он должен в значительной степени переписать файл программы, что кажется неестественным, если на бумаге достаточно просто стереть одну стрелку и прочертить другую.

Следует также отметить, что хотя модели *Pilgrim 5* могут иметь разную степень сложности, исходные файлы, отражающие их, имеют строго определенную структуру (рис. 2).

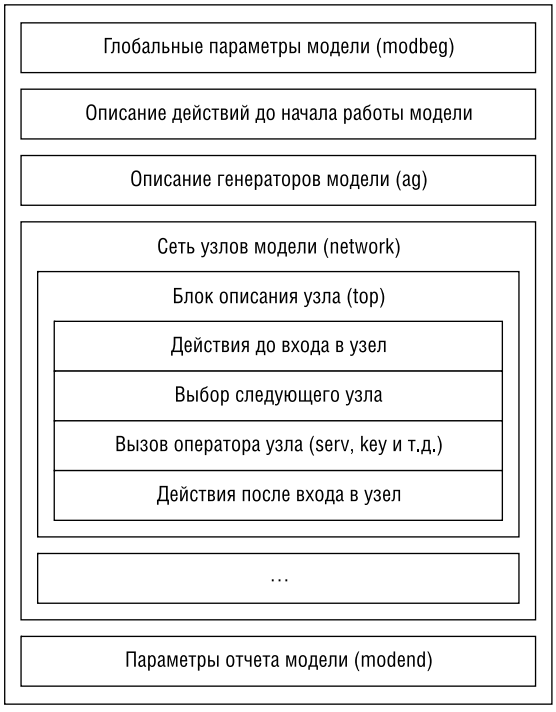

Рис. 2. Структура исходного файла модели в среде *Pilgrim 5*

Далее приведен список причин, по которым создание графического конструктора для среды имитационного моделирования *Pilgrim 5* с самого начала являлось уместным и, как было отмечено выше, практически необходимым.

1. Большинство пользователей системы *Pilgrim 5* не являются программистами и потому испытывают сложности при ручном составлении моделей.

2. При работе вручную, как правило, пользователь вынужден составлять модель дважды — в графическом и текстовом виде, хотя второе обычно в точности отражает первое, как минимум по количеству и типам используемых узлов и имеющимся между ними связям.

3. Изменение направления перехода транзакта из какого-либо узла может привести к необходимости значительной переработки, т. е. перенабора текста модели, что не является рациональным.

4. Наличие четкой структуры исходного файла модели, безусловно, свидетельствует о возможности и уместности частичной автоматизации процесса составления моделей.

Конструктор *Gem*

В свое время появление конструктора имитационных моделей *Gem* для среды *Pilgrim 5* в значительной степени упростило и ускорило процесс создания моделей. Пользователь обрел инструмент, позволяющий получить исходный файл модели из ее чертежа в конструкторе. Иными словами, конструктор *Gem* способен генерировать формальное, готовое к компиляции, представление модели из ее графического представления, избавляя пользователя от необходимости делать двойную работу, предварительно составляя модель на бумаге. На рис. 3 приведен внешний вид основного экрана конструктора *Gem*. Из рисунка видно, что в конструкторе редактируется модель, исходный текст которой был приведен ранее.

Gem ликвидировал множество проблем, в частности, пользователю были предоставлены следующие возможности:

- построение структуры модели с помощью мыши и метода drag and drop («перетащить и оставить») применительно к узлам и связям между ними;
- удобные диалоги для редактирования переменных модели, ее глобальных параметров и параметров отчета;
- единое диалоговое окно для ввода названия узла, его номера, параметров, дей-

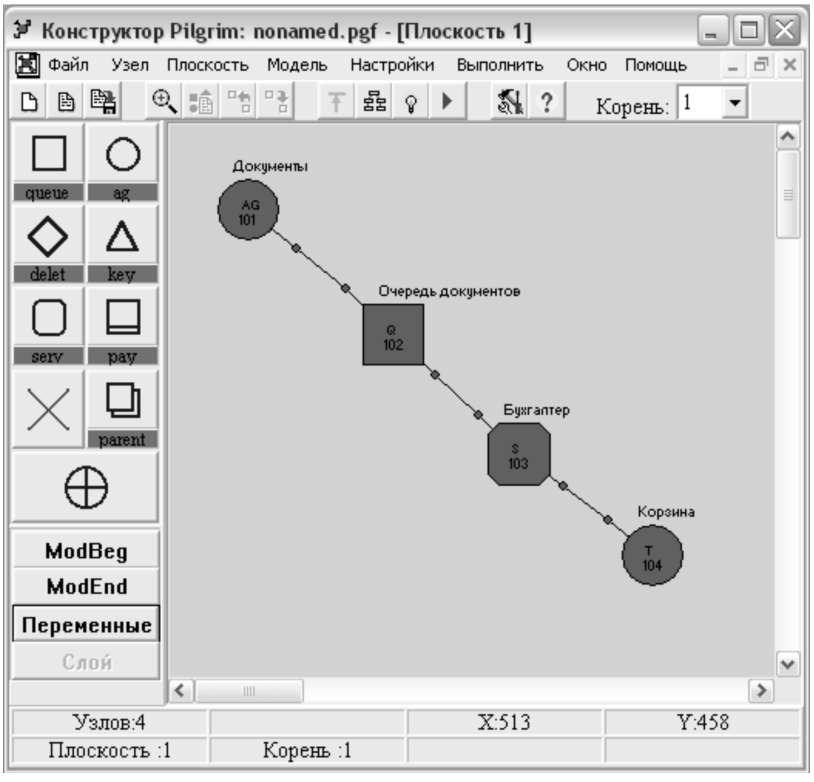

Рис. 3. Основное окно конструктора *Gem*

ствий необходимых к произведению до и после входа транзакта в узел, условий, с помощью которых для транзакта выбирается дальнейший маршрут;

- при изменении нескольких выходов из узлов нет необходимости пересматривать всю модель — нужно только удалить ненужные выходы, добавить нужные и снова запустить процесс генерирования файла модели;
- иерархический подход к построению моделей с помощью узла Parent, не являющего узлом *Pilgrim 5*. С помощью этого узла возможен перенос частей редактируемой модели (например, каких-либо полусамостоятельных частей некоторого процесса) на так называемые подслои. В основном слое модели эта ее часть заменяется узлом типа Parent, что в некоторых случаях позволяет добиться значительно лучшей читабельности графа. При генерации исходного файла модели, конструктор автоматически распознает узлы типа Parent и транслирует их в реальные цепочки узлов, отображенные на том подслое, на который ссылается этот конкретный Parent;
- генерация исходного файла модели по ее чертежу — делается это буквально одним щелчком мыши.

К сожалению, не была решена главная проблема пользователей-непрограммистов. Дело в том, что при всех своих достоинствах *Gem* не позволяет пользователю полностью абстрагироваться от синтаксиса C++.

За исключением вызовов функций узлов, весь остальной текст действий (до начала работы модели, до входа транзакта в узел, после входа транзакта в узел и условия переходов между узлами) по-прежнему должен вводиться вручную. Конструктор с помощью диалоговых окон позволяет пользователю не беспокоиться о том, куда будет вставлен текст, но проверка синтаксической корректности этого текста, как и раньше, должна осуществляться самим пользователем. Это же касается и диалогового окна для определения переменных параметров: пользователь должен знать, что целое в языке C++ имеет тип int, а дробное — float (или double). Говоря коротко, приложение не осуществляет синтаксического контроля. Во все указанные диалоговые окна пользователь может ввести любые «знакосочетания», поля для ввода параметров узлов также не проверяются ни на тип, ни на допустимость введенного в них значения (например, типичными ошибками, не контролируемыми конструктором *Gem* являются ввод отрицательного времени обслуживания, дробного числа каналов у сервера или переменного параметра, который не был определен или чей тип записан с ошибкой).

Разумеется, грубые ошибки будут замечены компилятором C++, но это случается не всегда. Если пользователь при вводе параметров сервера перепутал поля и вместо числа каналов записал дробное значение математического ожидания, то такая программа скомпилируется и даже запустится, но результаты будет выдавать неверные. При этом следует помнить, что мы говорим о пользователе, который не является программистом, и при всем своем желании может оказаться неспособным исправить ошибки в текстовом файле модели, даже если они будут обнаружены компилятором.

Справедливости ради стоит отметить, что конструктор *Gem* проверяет некоторые смысловые ошибки модели. Однако проверке подлежат исключительно элементарные оплошности пользователей, такие как наличие входов в генератор или выходов из терминатора. Например, модель, которая состоит из одного генератора и следующего за ним сервера с выходом в самого себя, по мнению конструктора *Gem* вполне «проходная».

Подведем итог. Конструктор *Gem* облегчает процесс создания имитационных моделей для пользователя, выстраивая для него узлы с их параметрами и связями, а также текст, введенный пользователем, в соответствии с необходимой структурой исходного файла модели для среды *Pilgrim 5*,

но не берет на себя ответственности ни за синтаксическую корректность модели, ни за ее смысловую нагрузку. Эволюция и рост популярности пакета *Pilgrim 5* способствовали созданию для него нового графического конструктора.

Конструктор *Architect*

При разработке графического конструктора имитационных моделей *Architect* концепция «удобства для пользователя» понималась, как необходимость предоставить ему возможность работать с составляемой моделью в терминах его предметной области или, в крайнем случае, в терминах имитационного моделирования. Требовалось избавить пользователя от взаимодействия с исходным файлом модели, т.е. с текстом на языке программирования C++ (иными словами, позволить ему работать с моделью на некотором уровне абстракции, не заботясь о деталях ее реализации в самой системе имитационного моделирования). Таким образом, была предпринята попытка снять «образовательный ценз» для пользователей *Pilgrim 5*, не являющихся программистами или другими IT-специалистами.

На рис. 4 изображен основной экран конструктора *Architect*. Здесь также редак-

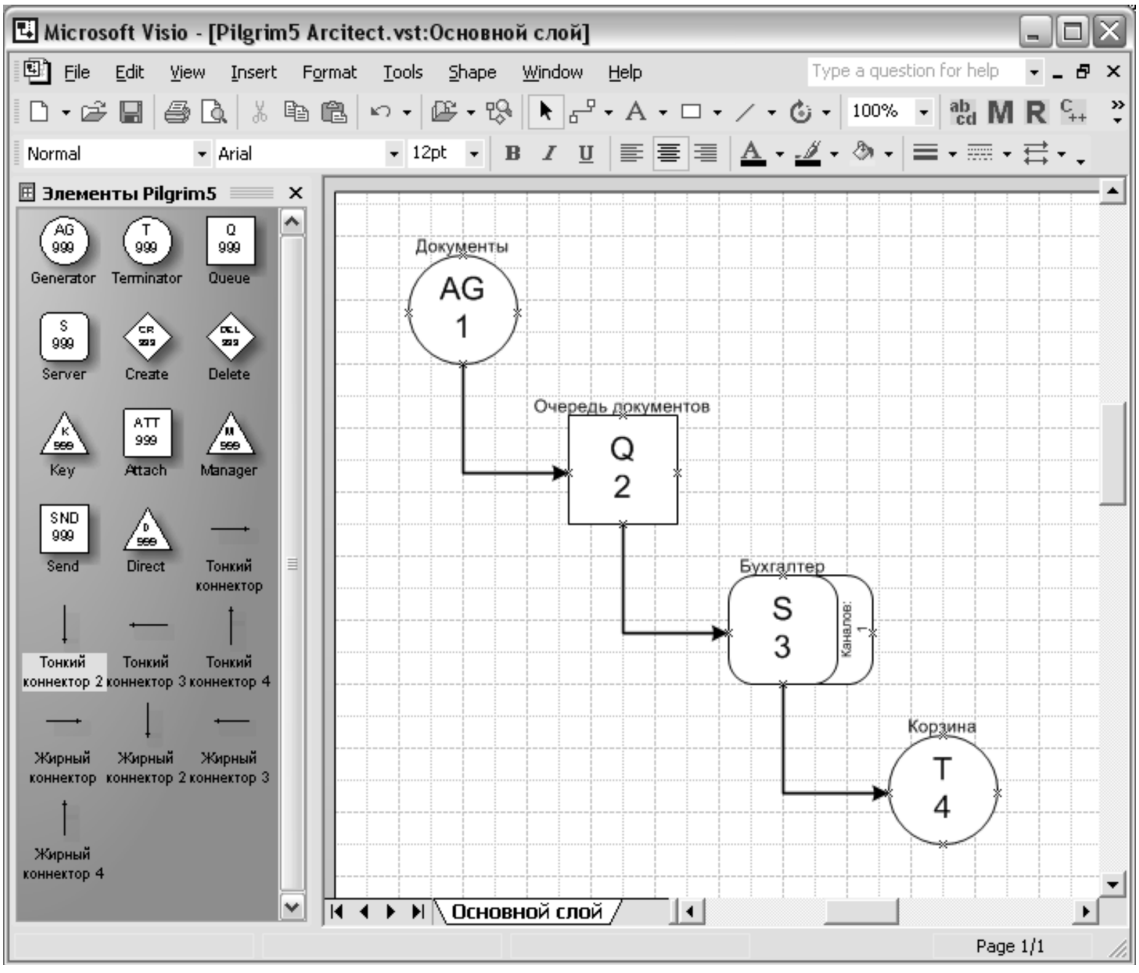

Рис. 4. Основное окно конструктора *Architect*

тируется модель, текст которой был приведен в начале статьи. Вообще, все приведенные тексты моделей, были получены с помощью *Architect* и оставлены в том виде, в каком он их выдает.

Architect реализован в виде шаблона (или решения) программы *Microsoft Visio*, что облегчает создание моделей для пользователей уже знакомых с этим редактором схем за счет известного интерфейса и методик работы. Одновременно такой подход к реализации конструктора не осложнит работу для новых пользователей. Построение структуры модели, как и в конструкторе *Gem*, происходит при помощи технологии drag and drop.

Кроме того, при визуальном редактировании схемы можно пользоваться практически всеми возможностями, предоставляемыми *Visio*, например:

- выделять ключевые переходы между узлами (стрелки) различной толщиной, видом или цветом линий;
- выполнять заливку узлов различным цветом;
- помещать текстовые сообщения на страницу со схемой и т. д.

Указанные возможности редактирования не сказываются на содержании исходного файла модели, но помогают в подготовке сопутствующей документации и делают саму схему более читабельной.

Последовательность шагов при построении модели в конструкторе *Architect* практически не отличается от очередности шагов того же процесса при использовании карандаша и бумаги или при использовании *Gem*. Эта последовательность включает в себя:

- составление сети узлов;
- выбор их параметров;
- определение действий до и после входа транзакта в каждый узел;
- определение условий переходов между узлами;
- установка глобальных параметров модели и параметров отчета;
- генерация исходного файла модели (хотя при «бумажной» технологии последний шаг нужно производить вручную, как и все остальные).

В следующем разделе будет подробно описана технология работы с конструктором *Architect*, а пока рассмотрим ряд ключевых отличий в отношении того, как выполняются приведенные шаги реализации имитационной модели.

1. В редакторе *Architect* имеется возможность сохранять в одном файле документа несколько моделей. Это могут быть совершено разные модели, или одна и та же модель с разными параметрами узлов. Единственное, что нужно сделать для переключения на трансляцию другой модели — это изменить один параметр («Основной слой») в диалоговом окне редактирования глобальных параметров модели.

2. Конструктор допускает модернизацию с целью приближения к предметной области пользователя. Возможна замена палитры с узлами, например, если большинство моделей имеет кадровую направленность, можно создать палитру, где узел Server заменен на узлы «Юрист», «Бухгалтер», «Грузчик» и т. д., причем узлы могут иметь уникальные параметры по умолчанию. Конечно, такая возможность будет полезна только опытным пользователям и экспертам.

3. Предусмотрена синтаксическая проверка параметров узлов на тип и характер вводимых значений. Например, редактор не позволит указать отрицательное число в качестве количества каналов сервера или узел Create в качестве направления проводки для узла Send. Причем проверка производится дважды: во время редактирования параметров узла и во время трансляции схемы в исходный файл модели (сделано это на тот случай, если с момента последнего редактирования параметров узла изме-

нилась схема модели, количество или тип переменных параметров и т. д.).

4. Пользователю не надо писать исходный код на языке C++ при составлении необходимых к выполнению действий. Для этого предусмотрен редактор, позволяющий добавлять в нужные места модели код для изменения переменных, параметров транзактов и вызовов операторов.

5. Есть возможность добавлять в выражения условные блоки и опять-таки без написания кода вручную. Как и при программировании действий допускается использование переменных параметров, некоторых параметров узлов, параметров транзактов и глобальных переменных *Pilgrim 5* (например, timer). Единственные знания, которыми должен обладать пользователь — это знания том, что он хочет получить, плюс элементарные знания о правилах построения и вычисления выражений. И составление действий и вставка условных блоков стали значительно удобнее благодаря специально разработанной технологии Step Chain Logic (или, просто, SCL), позволившей отделить мнемоническое представление выражений (намного более понятное пользователю) от сухого представления в синтаксисе языка C++. Подробнее технология SCL будет рассмотрена ниже.

6. Трансляция модели в исходный код происходит на глазах у пользователя. Он видит, какие синтаксические или семантические ошибки присутствуют и где именно они встретились транслятору.

В целом конструктор *Architect* берет на себя большую долю ответственности за правильность синтаксиса и семантики разрабатываемой модели.

Все сказанное выше относится к программе *Architect 1.0 и 2.0*. В настоящее время разрабатывается версия 3.0, реализуемая в виде отдельного приложения (*не* шаблона *Visio*). В ней предусмотрена интеграция с *Microsoft Visual C++* для автоматической сборки проекта, компиляции и линковки исполняемого файла модели, что еще более упростит работу пользователей *Pilgrim 5*.

Установка

Концепция конструктора *Architect* постоянно эволюционирует, а сама программа модернизируется. Как следствие, процедуры по ее установке и использованию также не остаются неизменными. Все последние инструкции и актуальные замечания по работе с конструктором находятся в файлах Install.txt и ReadMe.txt, поставляемых с конкретной версией и экземпляром *Architect*, который, в свою очередь поставляется вместе с *Pilgrim 5*. Конструктор поступает к пользователю в папке *Architect* в запакованном виде и устанавливается с помощью стандартного механизма *Windows Installer* (запуском программы setup.exe).

В распакованном виде конструктор состоит из следующих файлов (рис. 5):

Рис. 5. Комплект поставки конструктора *Architect* (файловая структура)

• Pilgrim 5 elements.vss — палитра *Microsoft Visio* с узлами *Pilgrim 5*. Данный файл открывается автоматически при запуске шаблона Pilgrim 5 Architect.vst;

• Pilgrim 5 Architect.vst — главный файл программы в виде файла-шаблона приложения *Microsoft Visio*. Именно с помощью этого файла выполняется запуск конструктора. Чтобы начать построение модели можно открыть этот файл непосредственно из Проводника *Windows* (*Windows Explorer*) с помощью команды Выполнить… (Run…) или из *Microsoft Visio*;

• четыре файла с растровыми значками для дополнения основной панели инструментов *Microsoft Visio* четырьмя кнопками с функциями, специфическими для конструктора *Architect* и *Pilgrim 5* (modbeg_small.ico, vars_small.ico, compile_small.ico, report_small.ico);

• в папке Samples находятся примеры реализации различных моделей *Pilgrim 5* с помощью конструктора *Architect* в виде файлов документов *Microsoft Visio* (с расширением vsd);

• Install.txt — последние замечания по установке (файл отображается в окне в процессе установки);

• ReadMe.txt — актуальные замечания по работе с поставляемой версией *Architect*, наиболее часто задаваемые вопросы (Frequently Asked Questions — FAQ).

Графический конструктор *Architect* избавляет пользователя от многих забот, связанных с созданием имитационных моделей. Однако для реализации действующих моделей его одного недостаточно (рис. 6).

Для того чтобы иметь возможность создавать и запускать готовые имитационные модели *Pilgrim 5*, на компьютере должно быть корректно установлено следующее программное обеспечение:

• *Microsoft Windows 95* или выше в качестве операционной системы;

• *Microsoft Visio 2002* или выше из состава пакета *Microsoft Office* (рекомендуется

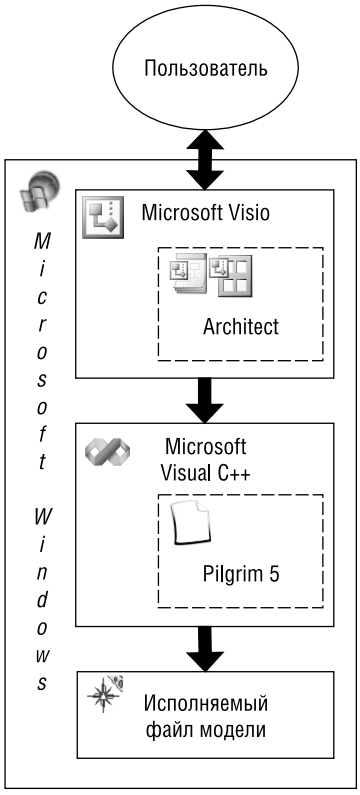

Рис. 6. Приложения и компоненты, необходимые для создания и запуска имитационной модели

Microsoft Visio 2003) — графический конструктор реализованный в виде шаблона документов данного приложения;

• *Microsoft Visual C++ 4.0* или выше из состава пакета *Microsoft Visual Studio* — для компиляции и сборки готовых исполняемых программных файлов моделей (с расширением exe);

• непосредственно пакет *Pilgrim 5*, состоящий из статических библиотек (lib), заголовочных файлов языка C++ (h) и файлов ресурсов (res), — необходим в процессе компоновки программных модулей.

Кроме того, следует обратить внимание на то, что в процессе своей установки графический конструктор *Architect* производит запись в определенные разделы системного реестра *Windows*, поэтому пользователь,

который производит установку, должен обладать соответствующими правами доступа.

Запуск

В этом и последующих разделах инструкции по запуску и иллюстрации экранных форм приводятся применительно к программе *Microsoft Visio 2003*.

Для того чтобы начать работу с конструктором *Architect*, необходимо запустить программу *Microsoft Visio* (рис. 7). Для этого с помощью мышки нажмите на кнопку «Пуск» («Start») и выберите последовательно пункты меню «Программы» («Programs»), «Microsoft Office», «Microsoft Visio 2003». Альтернативно, можно запустить *Microsoft Visio* с помощью ярлыка на рабочем столе (если таковой имеется), комбинации клавиш (если она закреплена за ярлыком), команды «Выполнить…» («Run…») или непосредственно из программы Проводник (Explorer).

На экране должно появиться главное окно программы *Visio*, изображенное на рис. 8.

Как уже было отмечено выше, конструктор *Architect* реализован в виде шаблона *Visio*. Перед началом работы будет не лишним узнать, как работает связка «шаблон–документ».

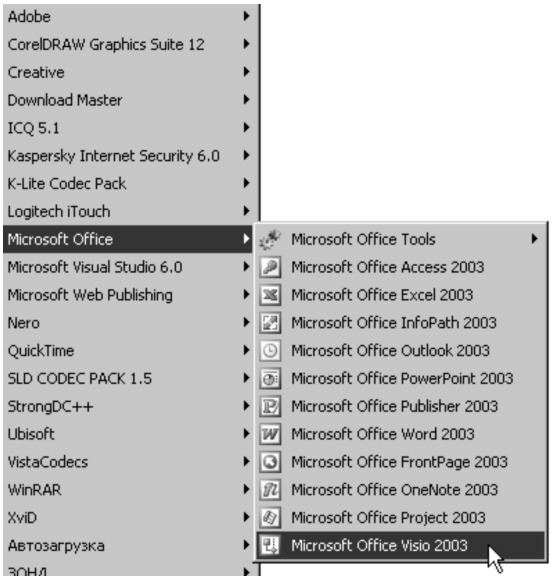

Рис. 7. Запуск приложения *Microsoft Visio 2003*

Приложение *Microsoft Visio* предназначено для создания разнообразных схем и чертежей. Оно позволяет строить их с помощью множества фигур, сгруппированных в палитры (stencils). (Английский термин «stencil» может переводиться по-разному, для наших целей вполне подойдет слово «палитра»). Каждый документ-схема может использовать

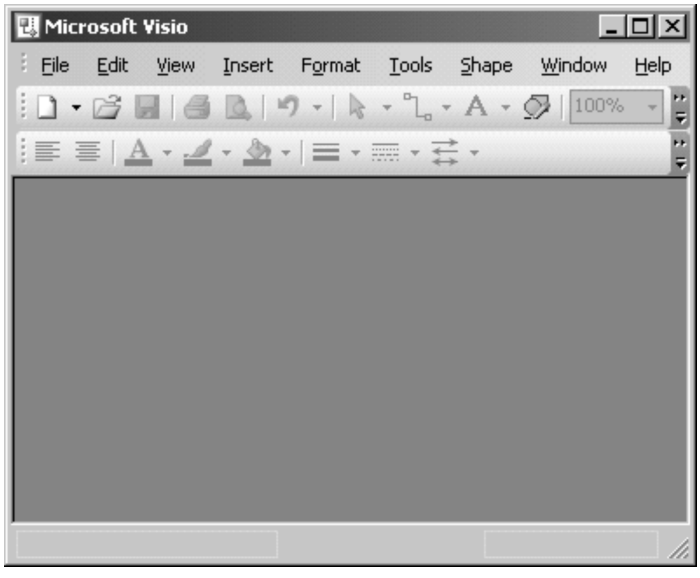

Рис. 8. Главное окно приложения *Microsoft Visio 2003*

произвольное число палитр, поставляемых с *Visio* или созданных самим пользователем. Схемы создаются посредством перетаскивания фигур с палитры в рабочую область документа методом drag and drop, хорошо знакомому каждому пользователю *Windows*. На рис. 9 изображена панель «Фигуры» («Shapes»), в которой представлены три палитры, наиболее часто используемые для создания блочных диаграмм. Палитры хранятся в отдельных файлах с расширением vss.

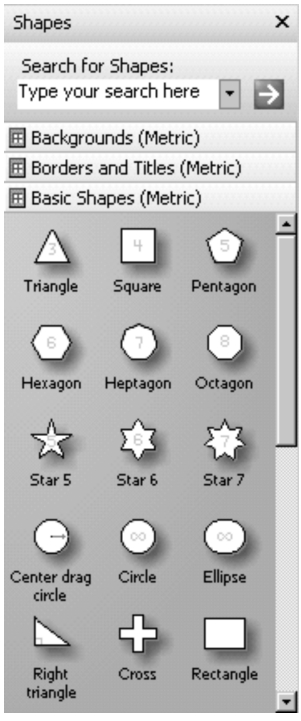

Рис. 9. Примеры палитр с фигурами *Microsoft Visio*

Microsoft Visio позволяет аннотировать схемы текстом, выделять любые элементы цветом, стилями линий и т. д.

Помимо этого документы и палитры могут содержать дополнительный программный код. Например, некоторые пользовательские фигуры могут видоизменяться в зависимости от специфических числовых параметров. Для того чтобы пользователю было проще вводить такие параметры, автор фигуры или всей палитры может связать с фигурами дополнительные диалоговые окна и код, который их обслуживает и видоизменяет фигуры. Кроме того, возможно, весь документ требует или допускает некую специфическую настройку. В этом случае дополнительный программный код связывается с целым документом. Разумеется, для создания пользовательского кода необходимы знания и навыки в области программирования, а точнее знания и навыки по созданию приложений на языке Visual Basic for Applications, среда разработки для которого встроена во все приложения *Microsoft Office*.

Когда пользователь сохраняет документ на жестком диске или где-либо еще, вместе с документом сохраняются ссылки на все палитры, элементы которых были в нем использованы, а также все дополнительные формы/диалоги и весь программный код из шаблона.

В сущности, после создания документа в файле шаблона нет необходимости, поскольку все, что было в шаблоне, при сохранении документа было скопировано в него. Необходимость в палитре узлов *Pilgrim 5*, тем не менее, остается. Если на момент открытия документа на том же компьютере, где он был создан, или на другом не будет присутствовать файл Pilgrim 5 elements.vss, документ все равно откроется (вы даже не получите никаких сообщений об ошибках). Вы по-прежнему сможете транслировать модель в код на C++ и изменять параметры и положение существующих в модели узлов, а также связи между ними. Однако в отсутствие палитры с узлами вы не сможете добавлять новые узлы, если в них появится необходимость.

Шаблоны (templates) — это, по сути, те же документы, но они сохранены с расширением vst. На основе таких документов-шаблонов можно создавать новые документы. При этом документ на основе шаблона создается пустым, однако в него изначально копируются все палитры и код из шаблона.

Поскольку модели *Pilgrim 5* нагляднее всего представляются именно в виде схем, становится очевидным, почему приложение *Microsoft Visio* было выбрано в качестве платформы для реализации конструктора. Аннотирование и другое дополнительное оформление моделей также бывают весьма полезны при составлении отчетов по моделированию и других документов. Примеры такого оформления мы рассмотрим несколько ниже, а сейчас вернемся к описанию процесса запуска конструктора *Architect*.

После запуска в главном окне *Visio* появится диалог выбора типа рисунка. Слева в списке групп необходимо выбрать «Другое» («Other»), а затем справа — единственный вид схемы «Pilgrim 5 Architect» (рис. 10).

Если по каким-то причинам после запуска главное окно программы *Visio* осталось пустым (как на рис. 8), последовательно выберите в главном меню пункты «Файл» («File»), «Новый» («New»), «Выбор типа рисунка…» («Choose drawing type…»), после чего описанное окно появится (рис. 11).

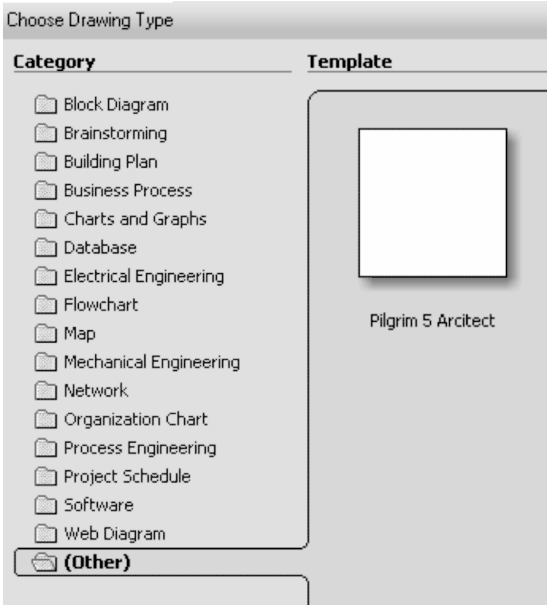

Рис. 10. Диалоговое окно «Выбор типа рисунка…»

Впрочем, если вы используете меню, то без этого окна можно вообще обойтись, выбрав команды «Файл» («File»), «Новый»

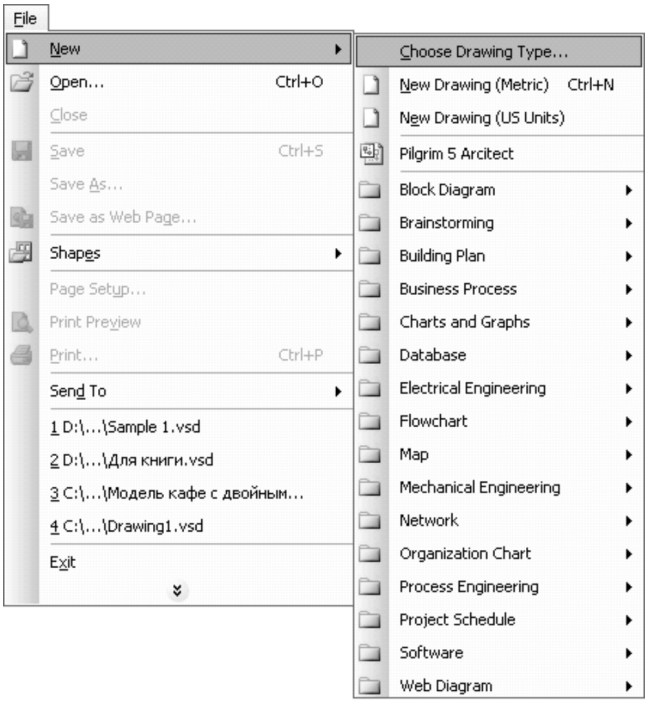

Рис. 11. Выбор типа рисунка с помощью главного меню *Visio*

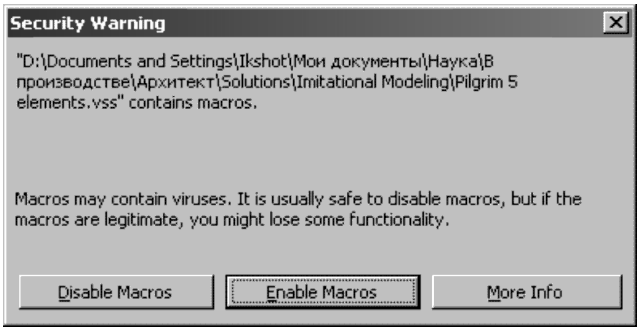

Рис. 12. Диалоговое окно блокировки/разрешения макросов

(«New»), «Pilgrim 5 Architect». С помощью этих действий вы сразу укажите *Visio* на необходимость создания документа на основе шаблона для имитационных моделей.

В процессе создания нового файла и открытия шаблона и палитры на экране последовательно появятся два окна вида, изображенного на рис. 12. Дело в том, что шаблон и палитра содержат программный код. Недобросовестный программист может при помощи встроенного в документы кода нанести вред вашему компьютеру (например, уничтожить данные на жестком диске).

Указанные диалоговые окна позволяют заблокировать или разрешить выполнение программного кода (Visual Basic for Applications является макроязыком, поэтому встроенный в документы код также называется макрокодом, набором «макросов», «скриптами», «сценариями»). Подобное окно появится дважды и, поскольку программный код является «сердцем» и «мотором» конструктора *Architect*, без которых ни одна из его полезных функций не будет работать, вам необходимо дважды нажать на кнопку «Разрешить макросы» («Enable Macros»), по одному разу для программного кода в шаблоне и в палитре.

В зависимости от настроек безопасности *Visio* окна блокировки/разрешения макросов могут и не появиться, причем это может означать как то, что программный код запускается без вашего ведома, так и то, что он блокируется опять-таки без вашего участия.

Самым первым признаком того, что программный код не выполняется, является отсутствие автоматической нумерации узлов модели. Попробуйте перетащить любой узел из палитры в рабочую область. В обычных условиях его номер должен измениться на 1. Если же его номер остается 999, значит макрос заблокирован. Это может случиться, если в упомянутых выше диалоговых окнах вы выбрали кнопку «Заблокировать макросы» («Disable Macros») или если уровень макробезопасности *Visio* установлен в «Высокий» («High») или «Очень высокий» («Very High»). В последнем случае вам необходимо изменить настройки макробезопасности, а затем закрыть и снова открыть программу *Visio*. Чтобы изменить уровень безопасности, последовательно выберите пункты меню «Сервис» («Tools»), «Параметры…» («Options…»). В появившемся диалоговом окне (рис. 13) перейдите на вкладку «Безопасность» («Security») и нажмите на единственную кнопку — «Защита от макросов» («Macro security»).

В следующем диалоговом окне (рис. 14) вы увидите список уровней макробезопасности. С помощью мыши установите его в «Средний» («Medium»). При таком уровне безопасности программа не будет блокировать или запрещать все макросы подряд, а будет спрашивать пользователя в каждом конкретном случае.

После настройки уровня безопасности дважды нажмите кнопку «OK» в двух диалоговых окнах, чтобы принять изменения и за-

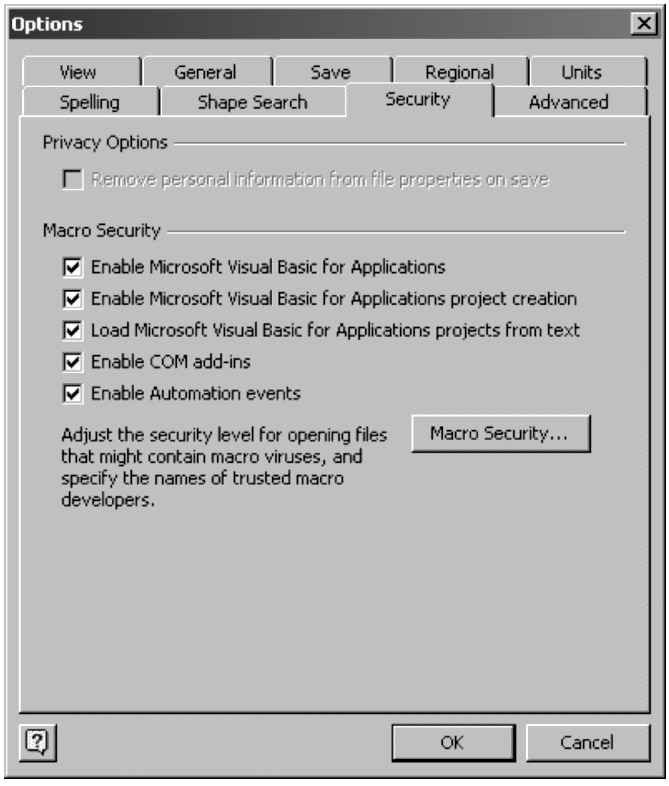

Рис. 13. Диалоговое окно параметров *Visio*

Рис. 14. Диалоговое окно настройки параметров макробезопасности *Visio*

крыть их. Изменения вступят в силу после перезапуска *Visio* (т. е. после закрытия программы и ее повторного запуска).

Сохранение файла с моделью выполняется так же, как и любого другого документа *Visio*. Открытие сохраненного файла, в основном, аналогично созданию нового, за исключением того, что вместо команды «Создать», следует использовать команду «Открыть» («Open») с последующим выбором существующего файла из списка.

Итак, после загрузки *Microsoft Visio*, изменения настроек (при необходимости) и создания нового документа на базе шаблона *Pilgrim 5 Architect* вы можете приступить к составлению своей модели.

Основы работы

Поскольку данный материал не предназначен для обучения читателя основам или тонкостям работы с программой *Microsoft Visio 2003*, в нем описываются только приемы работы, прямо или косвенно связанные с графическим конструктором *Pilgrim 5 Architect*.

Созданный документ, который впоследствии станет схемой имитационной модели, мало чем отличается от любого другого документа *Visio*. В частности, основная рабочая область будущей модели выглядит абсолютно стандартно: по умолчанию она белая, пустая, имеет размер листа бумаги формата A4, и на нее нанесена сетка, помогающая позиционировать фигуры. Однако при использовании *Pilgrim 5 Architect* на экране присутствует несколько специфических элементов (рис. 15).

По умолчанию в правой части инструментальной панели «Стандартная» («Standart») имеются четыре дополнительные кноп-

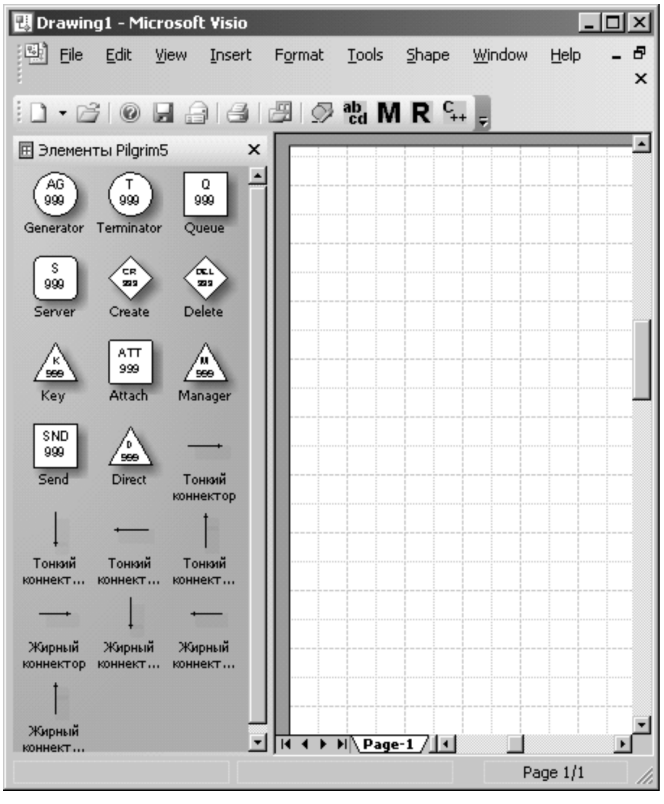

Рис. 15. Главное окно *Microsoft Visio 2003* после создания документа по шаблону *Pilgrim 5 Architect*

ки. Они крайне важны для работы с конструктором, и чтобы пользователь не потерял их в большой количестве других кнопок на той же панели, иконки, нанесенные на них, выполнены в черно-белой гамме. Свое знакомство с конструктором мы начнем именно с этих кнопок (рис. 16).

Рис. 16. Кнопки управления параметрами модели

Для удобства пользователя кнопки снабжены всплывающими подсказками. Они имеют следующее назначение (слева направо):

- вызов диалога редактирования переменных параметров модели;
- вызов диалога редактирования глобальных параметров модели;
- вызов диалога редактирования параметров отчета по модели;
- вызов диалога трансляции модели в программный код.

Рассмотрим функции каждой из них подробно.

Нажатие кнопки редактирования переменных параметров модели (с рисунком, содержащим буквы «$^{ab}_{cd}$») выводит на экран диалоговое окно, представленное на рис. 17.

Переменные параметры модели это любые вычисляемые или считываемые из какого-либо источника величины, которые непосредственно не связаны с архитектурой среды имитационного моделирования *Pilgrim 5*, но могут использоваться:

- в ходе навигации транзактов по модели (например, для определения следующего узла на их пути);
- в качестве параметров узлов модели;
- для хранения некоторых дополнительных результатов, которые не подсчитываются автоматически;
- любым другим способом, который создатель модели сочтет нужным использовать.

С точки зрения программиста, переменные параметры модели соответствуют обыч-

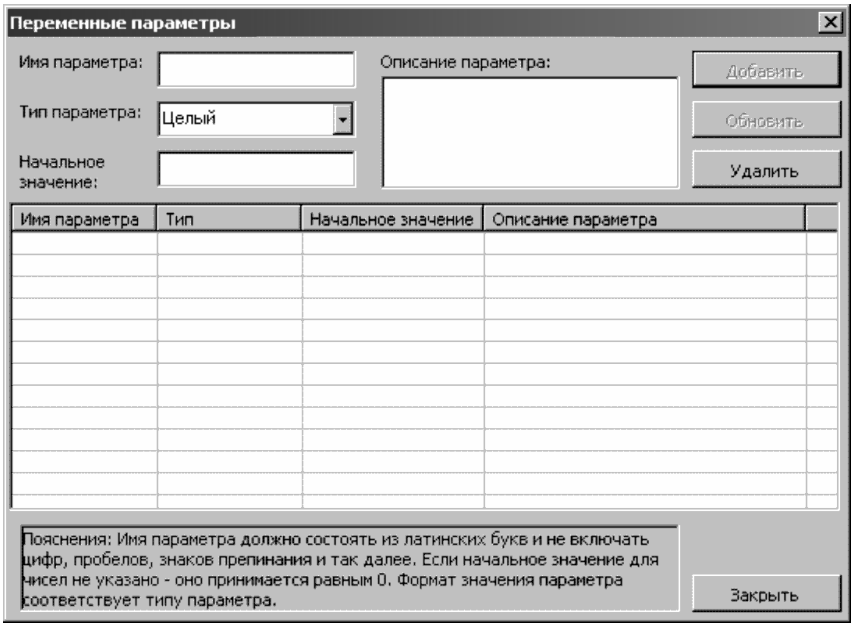

Рис. 17. Диалоговое окно редактирования переменных параметров модели

ным переменным языка программирования C++ и, в конечном итоге, именно так реализуются в файле с исходным кодом модели. Всю ответственность за работу с переменными параметрами несет создатель модели. *Pilgrim 5* никак не контролирует их значения и ничего в них не помещает. С другой стороны переменные параметры придают моделям большую гибкость (если знать, как их использовать), а некоторые модели в их отсутствие крайне сложно или вообще невозможно реализовать.

Для того чтобы существовать, переменный параметр должен иметь имя, тип и начальное значение. В диалоге, приведенном на рис. 17, имеются следующие элементы:

- сводная таблица имеющихся параметров в нижней части диалога;
- текстовое поле «Имя параметра». Оно должно состоять из заглавных или строчных латинских букв, без цифр, русских букв и других символов. Если пользователь нарушит это правило, то при попытке добавления или обновления информации о параметре он получит сообщение об ошибке. Следует помнить, что имена параметров чувствительны к регистру. Например, «Zf» и «zF» — это два разных параметра, типы и значения которых полностью независимы друг от друга;
- выпадающий список «Тип параметра». Допустимы четыре типа: целый, дробный, логический и распределение;
- текстовое поле или выпадающий список «Начальное значение». Данный элемент меняет свой внешний вид в зависимости от того, какой тип параметра выбран в предыдущем выпадающем списке. Если был выбран целый или дробный тип, то элемент представляет собой текстовое поле, в которое пользователь может собственноручно ввести желаемое начальное значение (характер значения автоматически проверяется на допустимость в соответствии с типом). Если в качестве типа взят логический, то начальное значение выбирается из выпадающего списка с двумя альтернативами: «Да» («Истина») и «Нет» («Ложь»). Распределение — это специальный тип переменных параметров, уникальный для *Pilgrim 5*. В отличие от остальных типов ему нет применения в обычных программах на C++, и с точки зрения этого языка он реализован в виде целого типа и макроопределений. Однако для имитационных моделей иногда бывает необходимо сменить распределение сервера в зависимости от каких-нибудь вычислений. В этом случае полезно иметь переменные специального типа. Итак, если установлен тип распределение, то пользователь может выбрать начальное значение параметра из выпадающего списка, в котором присутствуют следующие варианты: «Нет», «Нормальное», «Равномерное», «Экспоненциальное», «Бета». Если пользователь сам не выбирает или не вводит начальное значение параметра, конструктор при добавлении параметра запоминает значение по умолчанию в зависимости от его типа: 0 для целого и дробного типов или «Нет» для логического типа и типа распределения (следует иметь в виду, что в последних двух случаях «Нет» имеет совершенно разный смысл);
- текстовое поле «Описание параметра». Представляет собой комментарий, который без изменений войдет в файл исходного кода модели. Дело в том, что по сравнению с графическим представлением модели со всеми пометками и пояснениями, файл с исходным кодом модели нельзя назвать информативным с точки зрения пользователя-непрограммиста. Комментарии позволяют внести в него определенную ясность. Благодаря им становится хотя бы понятно, что означает конкретный параметр. Описание параметра с точки зрения синтаксиса может быть любым — здесь нет никаких ограничений;
- кнопка «Добавить». Становится доступна, после того, как пользователь ввел имя параметра, при условии, что параметр с идентичным именем еще не определен и не присутствует в списке в нижней части

диалога (в такой ситуации кнопка «Обновить» блокируется). Нажатие данной кнопки делает параметр «реальным». Именно после ее нажатия он запоминается в конструкторе и добавляется в список определенных параметров;

• кнопка «Обновить». Становится доступна, если пользователь ввел имя уже существующего параметра или выбрал его из списка. В этом случае пользователь может задать любые новые значения в полях «Тип параметра», «Начальное значение» и «Описание параметра». После нажатия кнопки «Обновить» информация о параметре будет изменена. Эту возможность следует использовать очень осторожно — возможна ситуация, когда «обновленная» переменная уже была задействована в качестве параметра какого-либо узла. В этом случае изменение ее типа приведет к ошибке на стадии трансляции модели в программный код;

• кнопка «Удалить». Становится доступна в тех же случаях, что и кнопка «Обновить», но ее нажатие приведет к полному удалению переменного параметра. С удалением следует быть осторожным по тем же причинам, что и в случае с обновлением.

Перейдем теперь от определения переменных параметров к установке глобальных параметров модели. Диалог редактирования глобальных параметров модели появится на экране после нажатия кнопки с рисунком в виде буквы «М» (рис. 18).

Большая часть параметров, которые можно установить в данном диалоговом окне соответствует аргументам функции modbeg(…), рассмотренной ранее. Такими параметрами являются:

• название модели;
• общее модельное время;
• пользовательский инициализатор датчика случайных чисел (в текущей версии конструктора не редактируется, используется инициализатор по умолчанию);
• контролируемая очередь для построения графика динамики задержек и контролируемый терминатор для построения графика динамики потока (в диалоге нужные

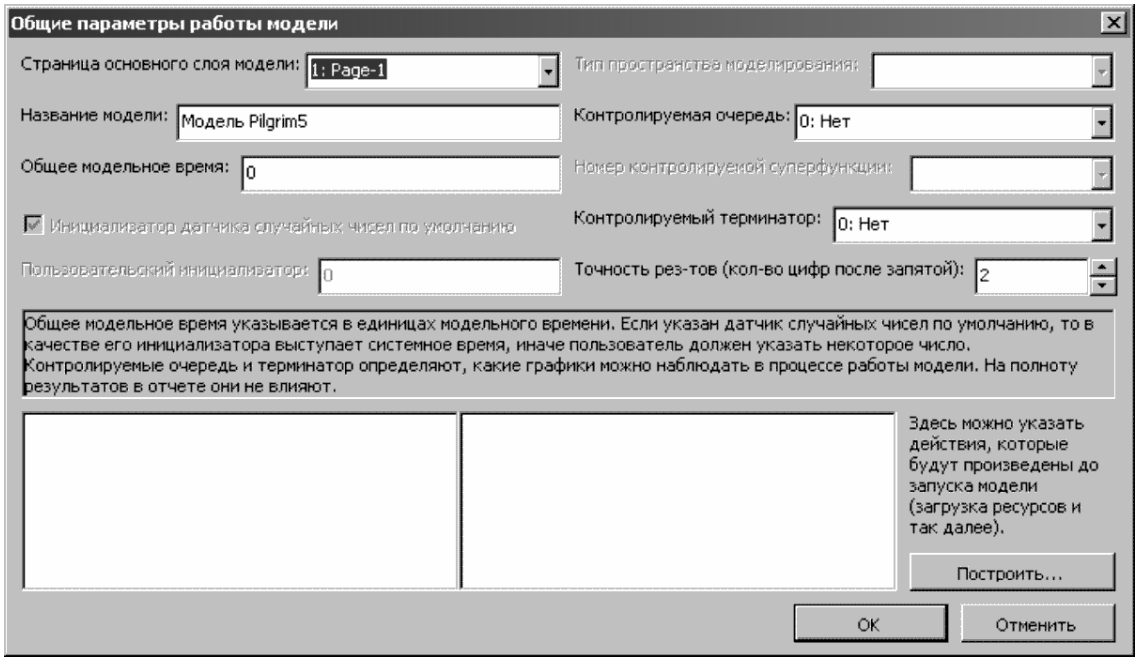

Рис. 18. Диалоговое окно редактирования глобальных параметров модели

очередь и терминатор выбираются из выпадающего списка, иными словами, эти узлы должны присутствовать в схеме до того, как их можно будет выбрать в качестве контролируемых);

- точность результатов моделирования, т. е. количество знаков после запятой;
- тип пространства моделирования и номер контролируемой суперфункции (в настоящей версии конструктора изменение этих параметров не поддерживается).

Помимо перечисленных параметров в диалоге присутствует выпадающий список «Страница основного слоя модели», два текстовых поля и кнопка «Построить», которые не имеют отношения к функции modbeg(…).

Список «Страница основного слоя модели» позволяет выбрать рабочий лист *Visio*, содержащий схему модели, которую следует транслировать в программный код. Это необходимо в том случае, когда в одном документе сохраняется несколько схем моделей на разных листах, что может быть весьма полезно. Следует помнить, что на каждом листе может быть своя схема, однако переменные параметры, а также настройки глобальных параметров модели и параметры отчета едины для всех схем (если они должны отличаться для разных листов, их необходимо изменить вручную).

Два текстовых поля и кнопка «Построить» в нижней части диалога позволяют задавать действия, которые должны быть выполнены моделирующей программой до фактического начала процесса моделирования и связаны со специально разработанной системой построения выражений SCL.

Перейдем к следующей кнопке на инструментальной панели, а именно к кнопке вызова диалога редактирования параметров отчета по модели (с рисунком в форме буквы «R»).

Диалоговое окно, появляющееся при нажатии этой кнопки (рис. 19) содержит намного меньше элементов, чем предыдущее. Большая часть редактируемых в нем параметров соответствует аналогичным параметрам функции modend(…).

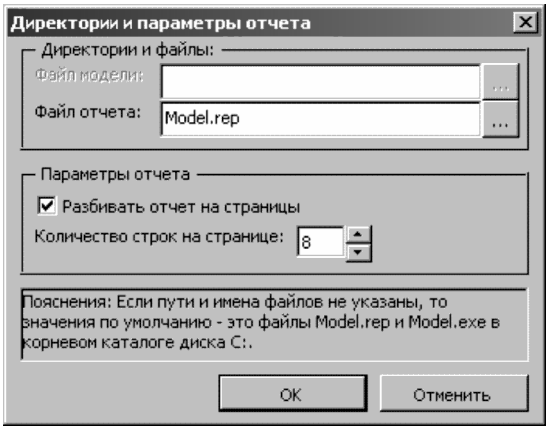

Рис. 19. Диалоговое окно редактирования параметров отчета по модели

В данном диалоге можно выбрать имя для файла отчета, а также его формат (наличие/отсутствие разбивки на страницы и количество строк на странице). Текстовое поле «Файл модели» предназначено для указания имени исполняемого файла модели. Оно недоступно, поскольку в текущей версии конструктора автоматическая компиляция и компоновка исполняемого файла не производится.

Переходим к последней (и самой важной) кнопке на инструментальной модели — кнопке вызова диалога трансляции модели в программный код (с нанесенными на нее символами «C++»).

Именно с помощью диалога, отображаемого при нажатии этой кнопки, создатель модели может получить ее представление, понятное компилятору с языка C++ и, впоследствии, создать ее исполняемый файл (рис. 20). Для того чтобы это сделать, необходимо нажать кнопку «Транслировать».

В верхней части окна расположен список событий трансляции: в процессе трансляции конструктор проверяет переменные параметры модели, действия до начала процесса моделирования, глобальные параметры модели и отчета, находит вначале генераторы, затем все узлы и связи между ними и соединяет это все воедино. В списке

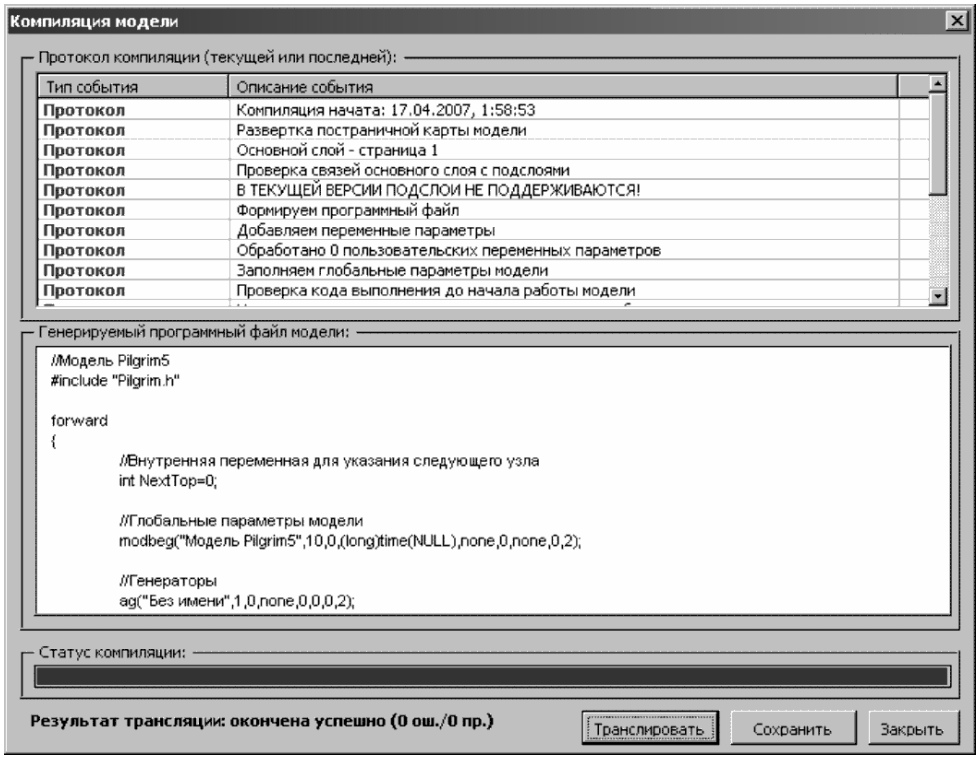

Рис. 20. Диалоговое окно трансляции модели в программный код

событий трансляции появляется запись о каждом шаге конструктора, а также все сообщения об ошибках и предупреждения.

Программный код модели, т.е. текст, соответствующий ее схеме и параметрам, на языке программирования C++, появляется в нижней части диалога.

Если в процессе трансляции конструктором была обнаружена ошибка, о чем свидетельствует запись в списке событий, а также запись «окончена неудачно» в строке «Результат трансляции», то программный код будет неполным и непригодным для дальнейшей обработки. В этом случае необходимо проанализировать сообщение об ошибке, закрыть диалог трансляции (с помощью соответствующей кнопки) и перед повторной трансляцией исправить ошибку в схеме модели или в ее параметрах.

Если трансляция завершилась удачно (в списке событий нет сообщений об ошибках, а в строке «Результат трансляции» появляется запись «окончена успешно»), значит синтаксически, а во многом и семантически модель верна, и ее программный код можно использовать для создания исполняемого файла модели.

Если трансляция модели в конструкторе *Architect* прошла успешно, это на 100% означает, что при обработке ее кода компилятором C++ проблем не возникнет. Однако это всего лишь вопрос синтаксиса. С точки зрения семантики, т.е. смысла того, что происходит в модели, конструктор также выполняет множество проверочных действий (например, наличие входов в генераторы или выходов из терминаторов), но он не может постичь замысел автора модели в целом, поэтому ответственность за то, что именно моделируется в большей степени лежит на самом авторе.

Для того чтобы сохранить полученный код в файл на диске, используется кнопка «Сохранить».

На этом мы заканчиваем знакомство с кнопками инструментальной панели и переходим непосредственно к созданию схемы модели.

В левой части главного окна *Visio* находится палитра «Элементы Pilgrim5», на которой размещены фигуры, соответствующие узлам, имеющимся в среде имитационного моделирования *Pilgrim 5*, а также коннекторы (соединяющие линии) необходимые для создания графа модели, т. е. для обозначения направления переходов транзактов между узлами (рис. 21).

Построение модели производится с помощью метода drag and drop, который часто применяется в среде *Windows* во множестве приложений. Например, для того чтобы добавить в модель узел типа «Генератор», необходимо навести курсор мыши на соответствующий значок в палитре, зажать левую кнопку мыши и, не отпуская ее, перетащить тень от значка в рабочую область документа. При помещении узла в рабочую область ему автоматически присваивается номер, а числовые, логические и остальные параметры принимают значения установленные по умолчанию.

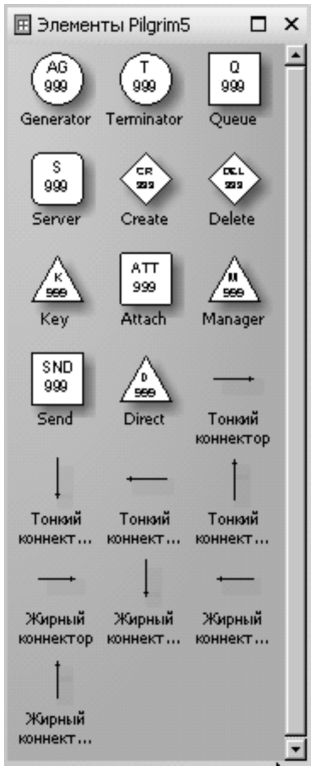

Рис. 21. Палитра «Элементы Pilgrim 5»

Нумерация узлов выполняется по порядку, который не может быть изменен произвольно. Например, вы поместили в документ два генератора, затем очередь и терминатор. В этом случае два генератора будут иметь номера 1 и 2 (в порядке их добавления), очередь — номер 3, а терминатор — номер 4. Если после этого добавить в модель узел-сервер, то он получит номер 5 независимо от того, где в графе модели он будет располагаться, из какого узла в него будут поступать транзакты, и куда из него они будут выходить. Если же перед добавлением сервера вы решите удалить генератор с номером 2, то сервер при добавлении получит именно этот номер — 2. Выражаясь формально, узлы нумеруются непрерывно безотносительно к топологии модели.

Для того чтобы соединить узлы между собой, необходимо перетащить в рабочую область документа коннектор, а затем с помощью уже упомянутого метода drag and drop соединить его концы с нужными узлами. При попадании оконечности (начала или конца) коннектора в область соединения с каким-либо узлом, в месте соединения появляется красная рамка. Отметим, что если какой-либо коннектор в момент трансляции модели оказывается «подвешенным», т. е. один или оба его конца не закреплены, то это не считается ошибкой — такой коннектор просто не учитывается, для транслятора он невидим. В сущности, поскольку коннекторы являются стандартными элементами *Visio*, вместо перетаскивания их с палитры, можно воспользоваться инструментами самого *Visio*. Однако по умолчанию коннекторы в *Visio* не являются направленными, конец коннектора не обозначен стрелкой в отличие от аналогичных коннекторов на палитре. Направление же коннектора чрезвычайно важно для транслятора, который должен знать переходит

транзакт из ключа в очередь или из очереди в ключ. Неаккуратное использование коннекторов может быть чревато ошибками, которые не будут зафиксированы транслятором, поскольку являются чисто семантическими.

Удаление узлов и коннекторов можно произвести с помощью кнопки «Delete» на клавиатуре или с помощью кнопки на главной инструментальной панели *Visio*. Кроме того, узлы и коннекторы можно копировать (при помощи клавиатуры, контекстного меню или кнопок на главной инструментальной панели). Если копируется узел, то при вставке появится новый узел с новым по порядку номером, но с параметрами оригинального узла.

В качестве примера, рассмотрим граф простейшей модели «Бухгалтер». Данная модель отражает процесс обработки сотрудником неких бумаг. Бумаги (транзакты) поступают из некоего источника (генератора) с определенной периодичностью в стопку на столе (очередь), обрабатываются бухгалтером (сервером) с заданной скоростью и помещаются в архив (терминатор).

После размещения всех нужных узлов в рабочей области документа, он будет похож на рис. 22.

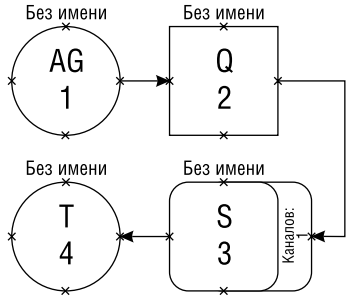

Рис. 22. Граф модели «Бухгалтер»

В данном случае имеется направленный граф, состоящий из узлов необходимых типов с правильной топологией. Однако этот граф еще не модель. Все узлы в нем имеют параметры по умолчанию, а догадаться, что собой представляет этот граф, стороннему наблюдателю довольно сложно.

Чтобы выполнить настройку параметров узла, необходимо произвести на нем двойной щелчок левой кнопкой мыши. В результате на экране отобразится диалоговое окно (рис. 23),

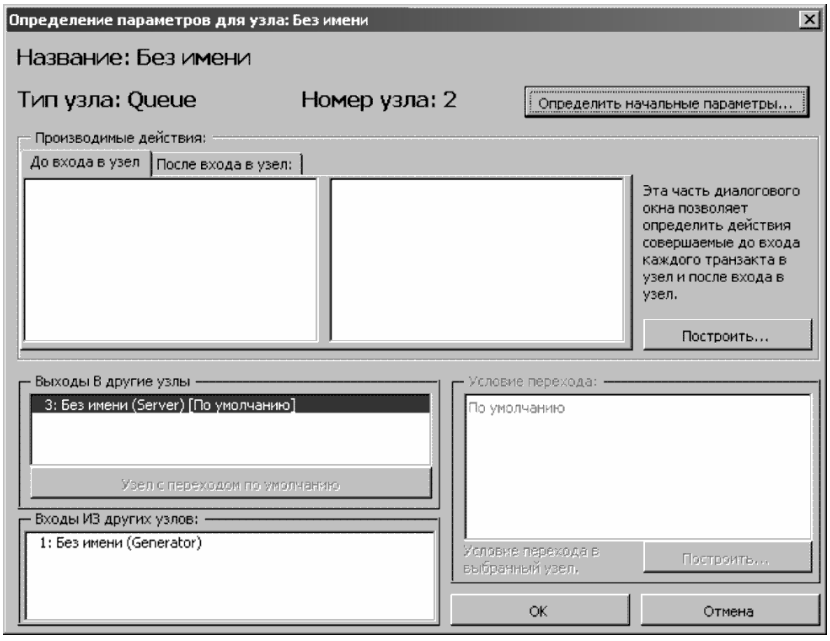

Рис. 23. Диалоговое окно настройки общих параметров узла

которое одинаково для всех типов узлов. Однако в зависимости от типа узла некоторые кнопки могут быть заблокированы (нет смысла планировать действия до входа транзакта в генератор или после его входа в терминатор).

В данном окне можно задавать операции, которые должны быть выполнены до и после входа транзакта в узел, а также условия перехода транзакта в определенные узлы, с которыми соединен данный узел. Оба действия выполняются с помощью системы SCL. В нашем случае важнее всего кнопка «Определить начальные параметры…»

При нажатии на нее появляется второе диалоговое окно, на этот раз специфическое для типа узла, параметры которого нам нужно отредактировать. На рис. 24 изображено окно редактирования параметров сервера, а на рис. 25 — окно редактирования параметров очереди.

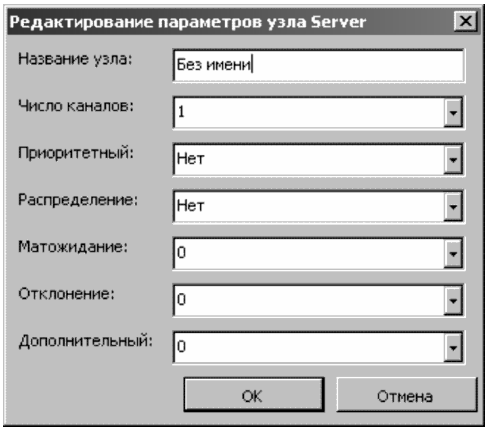

Рис. 24. Диалоговое окно редактирования параметров сервера

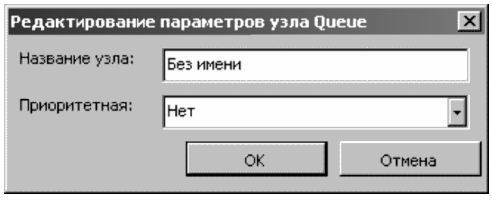

Рис. 25. Диалоговое окно редактирования параметров очереди

В подобных диалоговых окнах количество, тип и названия параметров соответствуют тем, которые применимы для конкретного типа узла. Если параметр получает только фиксированные значения, то любое из них можно выбрать с помощью выпадающего списка. Например, параметр очереди «Приоритетная» имеет логический тип, поэтому в выпадающем списке присутствует 2 значения: «Да» и «Нет». Если в числе переменных параметров модели были определены переменные логического типа, то они также будут присутствовать в списке, поскольку могут выступать в качестве параметра «Приоритетная» (в этом случае очередь будет приоритетной или нет в зависимости от того, какое значение имеет на каждый конкретный момент соответствующий переменный параметр). Сказанное выше верно и для других параметров различных типов, применительно ко всем узлам. Это дает большую гибкость при построении моделей.

После того как мы зададим значения параметров для всех узлов нашего графа, изображение модели «Бухгалтер» приобретет следующий вид (рис. 26).

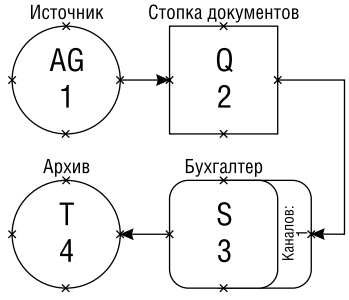

Рис. 26. Граф модели «Бухгалтер» после изменения параметров узлов

Хотя с первого взгляда изменились только подписи над узлами, внутри конструктора уже сохранились и другие их параметры. На данный момент модель готова к трансляции в код на C++.

Отдельно следует объяснить назначение системы «Шаговая цепная логика» (SCL). SCL — это система, построенная на интуитивном интерфейсе и позволяющая пользо-

вателю с легкостью конструировать логические выражения, манипулировать переменными параметрами и вызывать встроенные функции *Pilgrim 5*. Такая система может понадобиться для:

- определения операций, которые необходимо совершить до начала процесса моделирования;
- определения действий до входа транзакта в определенный узел или после входа в него;
- создания критерия выбора транзактом следующего узла на его пути (при наличии нескольких вариантов).

Конструктор *Architect* позволяет задавать целые блоки операций, выполняемых практически в любом месте модели, не требуя при этом от пользователя знания языка программирования (хотя знания правил арифметики и основ логики все же необходимы). Это делается при помощи диалоговых окон, которые могут быть открыты пользователем посредством кнопки «Построить…»

Предположим, что до входа транзакта в определенный узел, надо вычислить значение переменного параметра, который выступает в качестве параметра этого узла. Для этого следует произвести двойной щелчок на значке узла в рабочей области документа (на экране появится диалоговое окно настройки общих параметров узла, изображенное на рис. 23), после чего нажать на кнопку «Построить…» в верхней части этого диалога. В результате на экране появится окно, изображенное на рис. 27.

В этом окне можно вызывать функции *Pilgrim 5*, такие как открытие или закрытие ключа, взятие случайного числа, перенастройка генератора, загрузка склада ресурсами и другие (рис. 28), изменять параметры транзакта (рис. 29) или переменные параметры. Более того, любое из упомянутых действий может быть запрограммировано таким образом, чтобы выполнять-

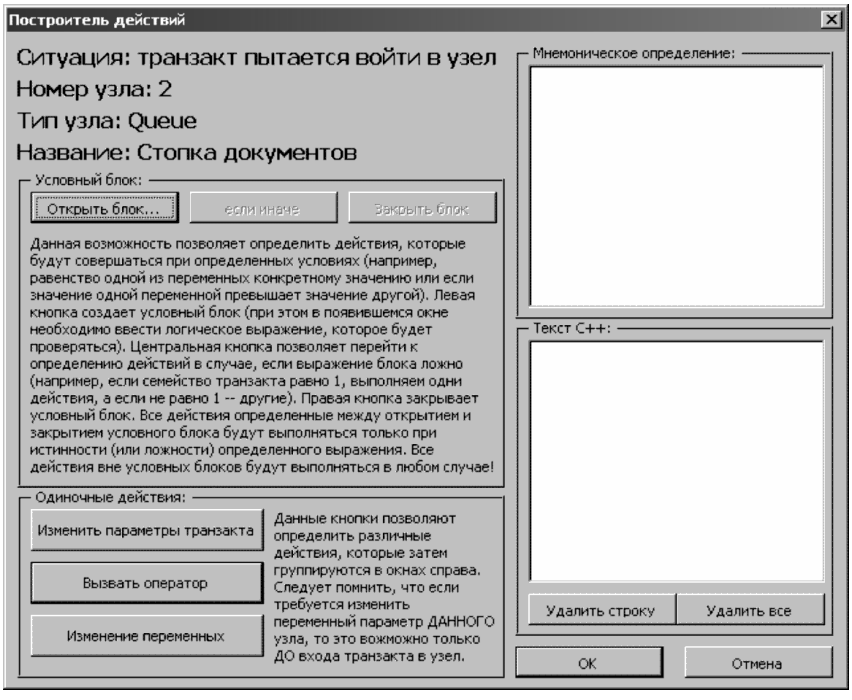

Рис. 27. Диалоговое окно «Построитель действий»

Рис. 28. Диалоговое окно «Выбор оператора Pilgrim5»

Рис. 29. Диалоговое окно «Изменение параметров текущего транзакта»

ся только при соблюдении определенных условий, задаваемых в виде логических выражений, опять-таки без программирования.

В данном случае нам необходимо произвести изменение переменного параметра. Для этого необходимо нажать на кнопку «Изменение переменных». В результате на экране возникнет новое диалоговое окно, изображенное на рис. 30.

В этом окне можно строить выражение с помощью арифметических действий над числовыми и логическими переменными и константами с использованием уже существующих в модели переменных параметров, а также внутренних переменных *Pilgrim 5* (например, параметров транзактов). Пользователю необходимо «собрать» нужное выражение и последовательно закрыть три открытых диалоговых окна с помощью кнопки «OK».

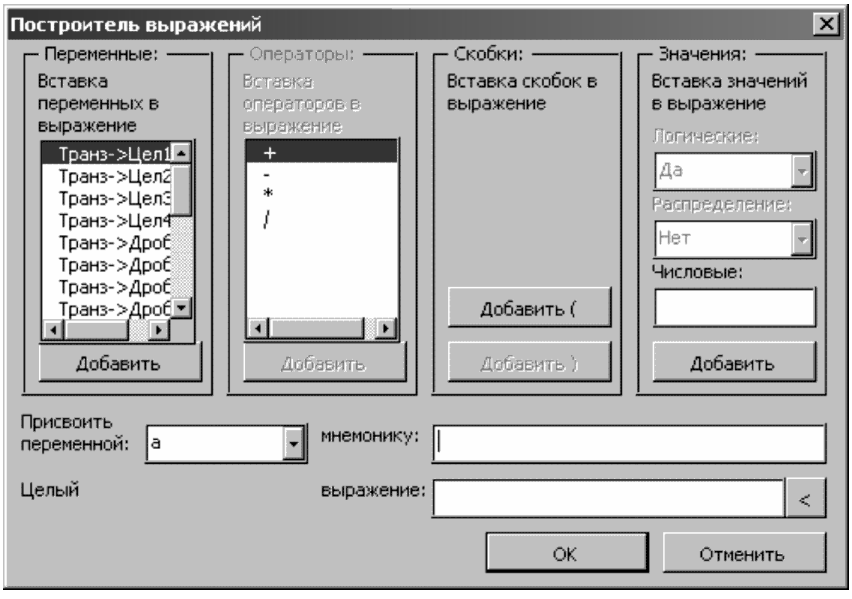

Рис. 30. Диалоговое окно «Построитель выражений»

Выпадающие (и обыкновенные) списки с контекстно-чувствительным содержимым — это одна из ключевых особенностей конструктора *Architect*. Например, в случае построителя выражений невозможно выполнить арифметические действия над логическими переменными, равно как и логические операции — над целыми или дробными переменными.

Другой ключевой особенностью *Architect*, реализованной в рамках SCL, является двойственное представление выражений. С точки зрения программы, удобнее всего хранить выражения в виде текста на C++, поскольку именно в такой форме они войдут в программный код после трансляции. Однако для пользователя такое представление выражения может быть непонятным, поэтому все выражения в *Architect* представлены в двух формах: в форме кода и в мнемонической форме, которая проще читается. Например, на рис. 31 изображено диалоговое окно построителя условий, в котором создается некоторое логическое высказывание.

В нижней части этого окна выражение представлено в двух видах:

- «`t-iu0==t-iu1 || a=0`» — эта форма, оптимальная для программы и привычная программистам на C++;
- «`[Транз-Цел1]=[Транз-Цел2] ИЛИ [a]=0`» — более громоздкая форма, однако она более понятна пользователям.

На этом мы закончим рассмотрение базовых возможностей системы SCL.

Дополнительные возможности

В заключение данной статьи отметим несколько дополнительных возможностей *Visio*, поддерживаемых конструктором *Architect*, которые позволяют не только создать имитационную модель, но и оформить ее. Пользуясь этими средствами можно придать документу со схемой модели законченный вид и сразу включить его в отчет. Таким образом, документ содержит в себе двойную функциональность.

Речь идет о средствах форматирования. Дело в том, что конструктор обрабатывает только фигуры с палитры «Элементы Pilgrim 5» безотносительно к тому, какого они цвета или какой в них применяется шрифт, а также внутренние переменные. Все ос-

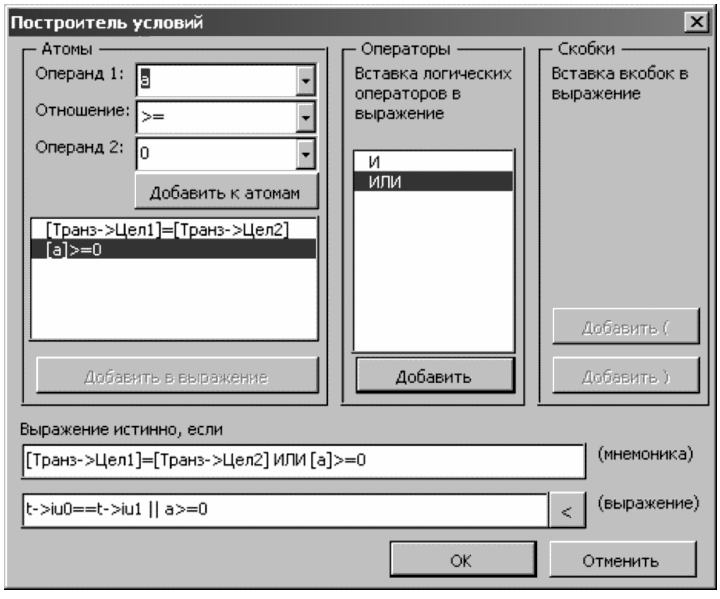

Рис. 31. Диалоговое окно «Построитель условий»

тальные элементы, будь они стрелками, геометрическими фигурами, текстом или чем-нибудь иным, конструктор не обрабатывает и вообще не видит.

Для того чтобы придать документу с моделью оформленный вид необходимо использовать:

- различные шрифты и стили для надписей в составе узлов (рис. 32);
- различные стили линий (рис. 33);
- дополнительные палитры и их элементы (рис. 34);
- цвета и стили заливки (рис. 35);
- текст (рис. 36).

Все нехитрые перечисленные приемы оформления положительно влияют на читабельность модели.

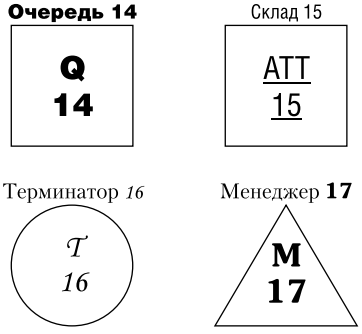

Рис. 32. Различные шрифты и стили

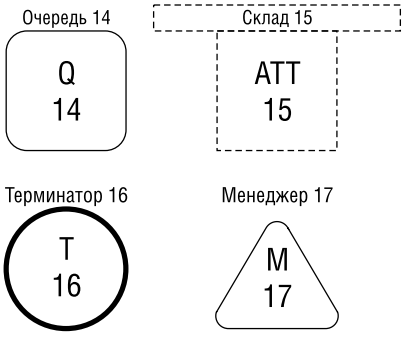

Рис. 33. Различные стили линий

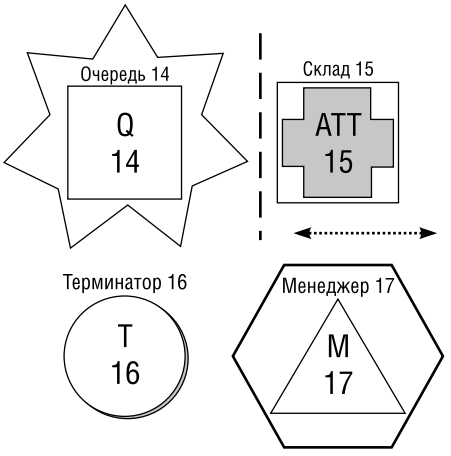

Рис. 34. Дополнительные элементы

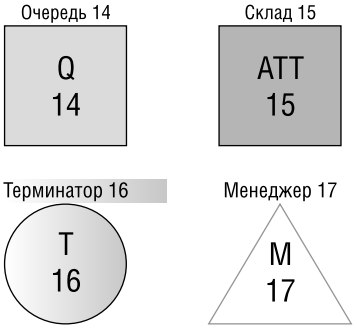

Рис. 35. Выделение цветом

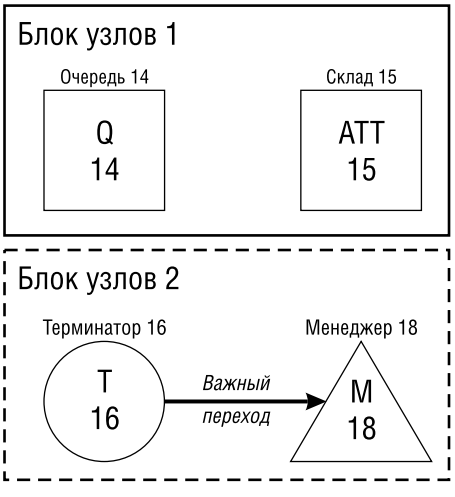

Рис. 36. Использование текста

Пример работы

В завершении данной статьи рассмотрим процесс построения законченной модели с использованием графического конструктора *Architect*. В качестве примера возьмем одну из моделей, часто используемых для обучения имитационному моделированию в среде *Pilgrim 5* — модель «Кафе» (рис. 37).

Смысл моделируемой реальной ситуации заключается в следующем:

- несколько потенциальных клиентов пытаются попасть в кафе;
- если перед кафе имеется очередь, то часть клиентов остается ждать, а часть уходит сразу;

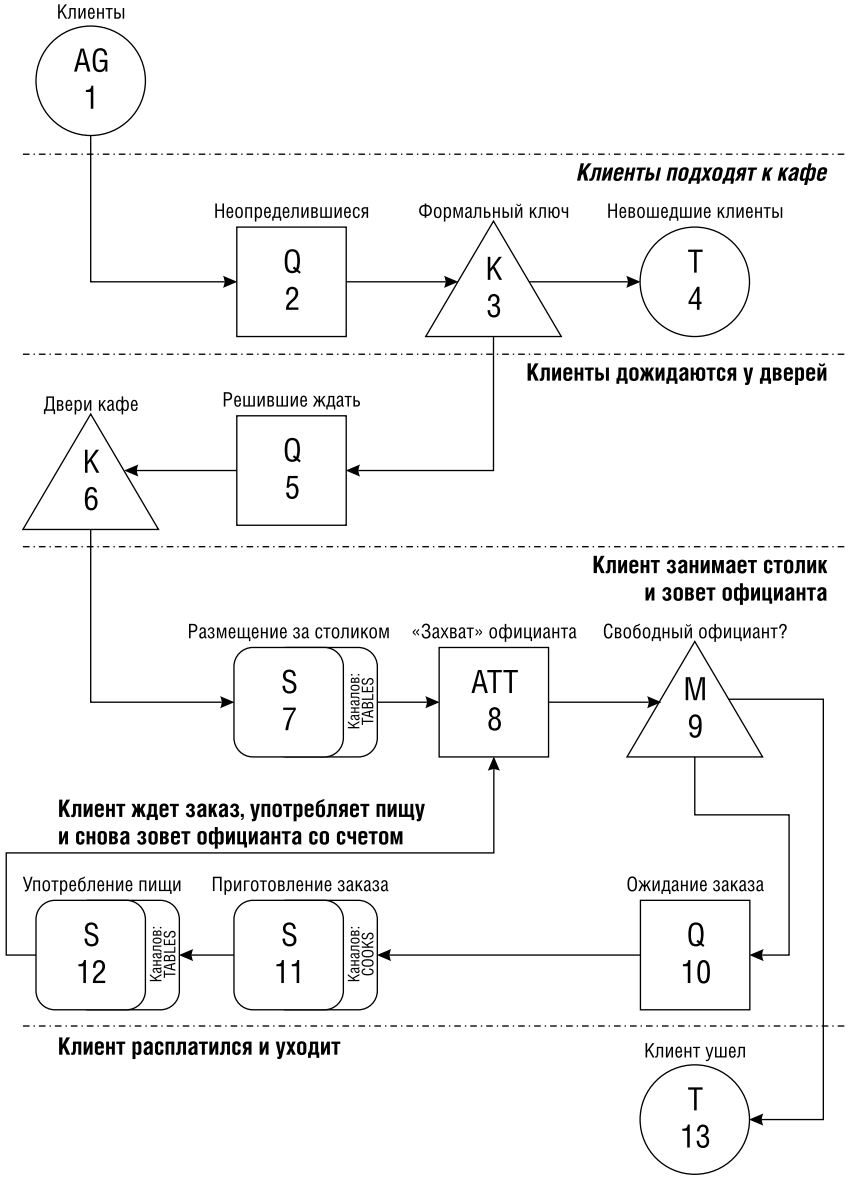

Рис. 37. Модель «Кафе с двойным захватом официантов»

• после входа в кафе клиентам необходимо некоторое время, чтобы раздеться, помыть руки и т. д.;
• расположившись за столиком клиенты ждут официанта;
• после того как заказ принят, требуется некоторое время на его приготовление и прием пищи;
• далее снова подходит официант, чтобы клиент расплатился;
• в конце процесса клиент уходит.

Схема и количество узлов модели, соответствующей описанной ситуации, также как ее программа, могут быть различными в зависимости от того, какой уровень детализации требуется, а также от подхода к построению модели, который применил ее составитель. Например, как столик в кафе, так и официант в модели могут быть представлены в виде активных ресурсов (узлов типа «Сервер») или пассивных ресурсов (узлов типа «Склад»); модель может быть построена с использованием циклов или без них и т. д.

Вариант модели «Кафе», схема которой приводится на рисунке, благодаря своей нетривиальности приобрел название «Модели с двойным захватом официантов». Сам прием «двойного захвата» берет свое начало из следующих наблюдений:

1. Наиболее очевидным было бы предположить, что ожидание официанта в модели должно быть представлено в виде узла типа «Сервер», поскольку с точки зрения теории массового обслуживания, официант интуитивно обслуживающий прибор. Однако такой путь построения модели, т. е. данного ее участка, блокирует клиентов от каких-либо действий в период ожидания обслуживания официантом, что не соответствует действительности.

2. В реальной ситуации ожидание официанта, для того чтобы сделать заказ, и ожидание его, чтобы расплатиться — два разных участка всего процесса похода в кафе, и эти участки могут быть значительно удалены друг от друга по времени. Вместе с тем, количество официантов в кафе за время посещения его конкретным клиентом редко увеличивается или уменьшается. Более того, как правило, тот же официант, который принял заказ, приносит счет за него. Клиенты имеют дело в обоих случаях с одними и теми же официантами.

В результате анализа указанных фактов при построении модели официанты были представлены в виде фиксированного количества пассивного ресурса (узел 8 на рис. 37), который транзакт-клиент в течение своего жизненного цикла в модели захватывает дважды (отсюда и название приема).

1. После выхода из узла 7 «Размещение за столиком», освобождая его после входа в узел 12 «Употребление пищи».
2. После выхода из узла 12, освобождая его перед входом в узел 13 «Клиент ушел».

Чтобы транзакт не двигался по замкнутому кругу (по пути узлов 8→9→10→11→→12→8→9→10→...) в одном из пользовательских полей транзакта передается флаг цели захвата официанта. Изначально он имеет значение константы ORDER (заказ), которое меняется на MONEY (соответствующее необходимости получить счет) перед входом в узел 12. В свою очередь в узле 9 производится проверка указанного флага и транзакт направляется по нужному маршруту, который в целом выглядит следующим образом: узлы 8→9→10→11→12→8→9→13.

Ниже приведен текст модели.

```
//Модель «Кафе с двойным захватом официантов»
#include "Pilgrim.h"
forward
{
   //Внутренняя переменная для указания
   //следующего узла.
   int NextTop=0;
   //Значение-флаг, показывающее, что клиент
   //зовет официанта, чтобы сделать заказ.
```

```
long ORDER=1;
//Клиент зовет официанта,чтобы расплатиться.
long MONEY=2;
//Интервал прибытия клиентов.
double timeCLI=3.000000;
//Время на размещение за столиком.
double timeSIT=2.000000;
//Время приготовления заказа.
double timeORD=3.000000;
//Время употребления пищи клиентом.
double timeEAT=15.000000;
//Количество столиков в кафе
long TABLES=5;
//Количество людей в очереди перед кафе,
//после которых не занимают пришедшие.
long qMAX=3;
//Колмчество официантов.
long WAITERS=2;
//Количество поваров или собирающих заказ.
long COOKS=1;
//Количество посетителей в кафе на данный
//момент.
long CustomersInCafe=0;
//Число людей,  при котором есть еще один
//свободный столи
long OneFreeTableLeft=0;к
//Глобальные параметры модели.
modbeg("Модель кафе с двойным захватом
       официантов",15,480,(long)time(NULL),
       none,0,none,0,2);
//Изменение переменной.
OneFreeTableLeft=TABLES-1;
//Загрузка [WAITERS] ресурса в узел 8.
supply(8,none,WAITERS);
//Генераторы.
ag("Клиенты",1,0,unif,3,2,0,2);
network(dummy,dummy)
{
  top(2):
     //Условия переходов.
     if(1) NextTop=3; else
     {
     }
     queue("Неопределившиеся",none,
     NextTop);
     place;
top(3):
  //Условия переходов.
  if(addr[5]->tn==qMAX) NextTop=4; else
  {
  if(1) NextTop=5; else
  {
  }
  }
  key("Формальный ключ",NextTop);
  place;
top(4):
  term("Невошедшие клиенты");
  place;
top(5):
  //Условия переходов.
  if(1) NextTop=6; else
  {
  }
  queue("Решившие ждать",none,NextTop);
  place;
top(6):
  //Условия переходов.
  if(1) NextTop=7; else
  {
  }
  key("Двери кафе",NextTop);
  place;
top(7):
  //Действия до входа в узел.
  if(CustomersInCafe==OneFreeTableLeft)
  {
  hold(6);
  }
  t->iu0=ORDER;
  //Условия переходов.
  if(1) NextTop=8; else
  {
  }
  serv("Размещение за столиком",TABLES,
      none,unif,timeSIT,2,0,NextTop);
  //Действия после входа в узел.
  clcode
  {
  CustomersInCafe=CustomersInCafe+1;
  }
  place;
top(8):
  //Условия переходов.
  if(1) NextTop=9; else
  {
```

```
    }
    attach ("Захват официанта",1,none,
           NextTop);
    place;
  top(9):
    //Условия переходов.
    if(t->iu0==MONEY) NextTop=13; else
    {
    if(1) NextTop=10; else
    {
    }
    }
    manage("Свободный официант?",NextTop);
    place;
  top(10):
    //Условия переходов.
    if(1) NextTop=11; else
    {
    }
    queue("Ожидание заказа",none,NextTop);
    place;
  top(11):
    //Условия переходов.
    if(1) NextTop=12; else
    {
    }
    serv("Приготовление заказа",COOKS,none,
        norm,timeORD,1,0,NextTop);
    place;
  top(12):
    //Действия до входа в узел.
    t->iu0=MONEY;
    //Условия переходов.
    if(1) NextTop=8; else
    {
    }
    serv("Употребление пищи",TABLES,none,
        unif,timeEAT,5,0,NextTop);
    //Действия после входа в узел.
    clcode
    {
    detach(8,1);
    }
    place;
  top(13):
    //Действия до входа в узел.
    rels(6);
    detach(8,1);
      CustomersInCafe=CustomersInCafe-1;
      term("Клиент ушел");
      place;
    fault(123);
    }
    modend("Cafe.rep",1,8,none);
    return 0;
}
```

Важен тот факт, что приведенная схема модели довольно далека от элементарной и получена с помощью графического конструктора *Architect*. Приведенный текст модели был автоматически сгенерирован и отформатирован, включая комментарии, на основании только этой схемы, настройки параметров узлов, а также настройки переменных и глобальных параметров модели. То есть все операции по составлению описанной модели и созданию ее исходного кода были выполнены в конструкторе *Architect* без программирования вручную, а также без карандаша и бумаги.

В дальнейшем конструктор *Architect* предполагается дополнить возможностью внесения в схему модели групп логически объединенных узлов «одним нажатием кнопки».

В среде имитационного моделирования *Pilgrim 5* есть структуры связанных в определенной последовательности узлов, встречающиеся в моделях наиболее часто. Их можно назвать структурными шаблонами. Каждая неучебная модель уникальна по целям своего создания и уровню детализации. Однако некоторые части многих моделей похожи как две капли воды.

В качестве наиболее красноречивого примера можно привести «схему зарядки». Она применяется для загрузки транзактов в модели замкнутого типа и, чаще всего, состоит из двух блоков, один из которых включает генератор, управляемый генератор и терминатор, а второй – генератор и терминатор. Эта схема переходит из модели в модель без изменений, однако пользователь вынужден каждый раз создавать ее заново по одному узлу. В будущих версиях *Architect* предполагается исправить этот недостаток.