Языки описания потока управления (control flow)

Поскольку целью настоящего отчёта является исследование средств графического программирования, потенциально пригодных для описания УА РВ, применяемых для управления БА КА, среди описываемых графических языков и средств программирования языки, сконцентрированные на описании потока управления (control flow), и родственные им языки описания потоков работ (workflow), заслуживают особого внимания.

7.1. Графические схемы (блок-схемы) алгоритмов и программ

Графические блок-схемы алгоритмов и программ исторически появились достаточно рано и изначально являлись лишь средством документирования, помогавшим задействованным в процессах жизненного цикла ПО людям лучше понять структуру программы. Однако затем появился целый ряд средств, позволяющих интерпретировать язык графических блок-схем, или генерировать по ним текст программы на том или ином языке программирования (см, например, главу 5).

Блок-схемы алгоритмов и программ были достаточно быстро стандартизированы, имеются как международный, так и российский стандарты (ISO 5807-85, ГОСТ 19.003-80, ГОСТ 19.701-90) [23].

Теоретической основой графического языка блок-схем служит управляющий граф программы. Схема отражает порядок исполнения отдельных операторов (действий) в программе, которые отображаются с помощью набора стандартизированных графических примитивов, как показано на рисунке 28. Процесс, или действие отображается с помощью прямоугольника, решение (ветвление потока управления в зависимости от истинности или ложности логического условия) – с помощью ромба. Передача управления отображается с помощью линий (стрелок). Имеются специальные символы для отображения начала и конца алгоритма. Блок-схемы вертикально ориентированы, что может затруднять их восприятие и манипулирование с ними. Надо отметить, что ГОСТ 19.701-90 предусматривает средства и для отображения параллельных ветвей алгоритма и параллельно выполняемых процессов. Более того, как описано в стандарте, он может применяться и для описания процессов, выполняемых людьми, т.е. в современном понимании – бизнес-процессов организации и протекающих в ней потоков работ (workflow).

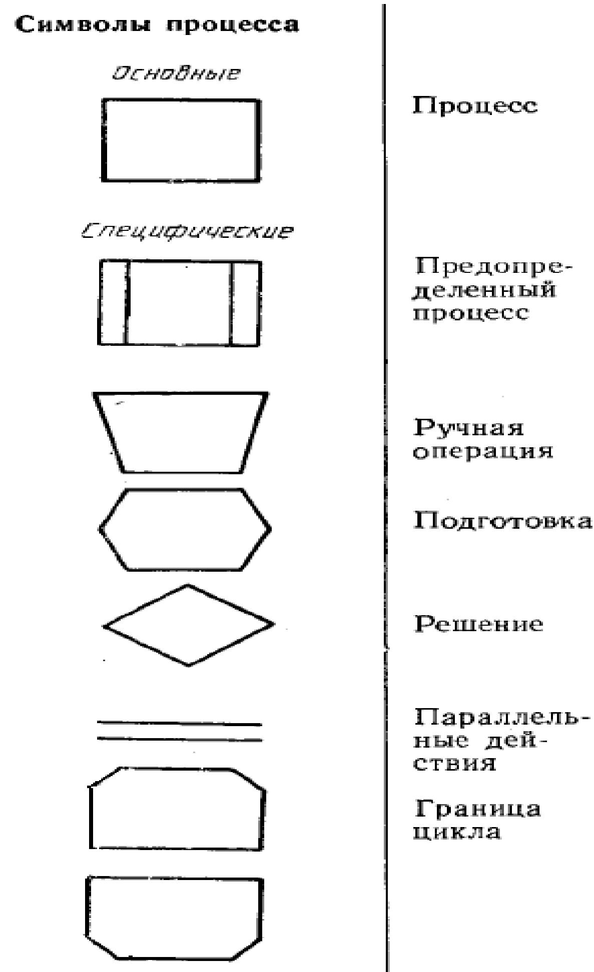

Рисунок 28. Базовые графические примитивы блок-схем алгоритмов и программ

Следует отметить, что возможности стандарта ГОСТ 19.701-90 вообще достаточно широки, он предусматривает средства для графического описания не только схем программ (т.е. фактически потока управления – controlflow), но и схемы данных, схемы работы системы, схемы взаимодействия программ и схемы ресурсов автоматизированной системы.

Как уже отмечалось, нотация графических блок-схем программ поддерживается большим количеством инструментальных средств. В частности, система ГРАФКОНТ, созданная в СГАУ и НТЦ «Наука», г. Самара, по заказу ГНПРКЦ «ЦСКБ-Прогресс» [10], позволяет отображать графически программу, наряду с другими представления, и в виде ее блок-схемы, что отражено на рисунке 29.

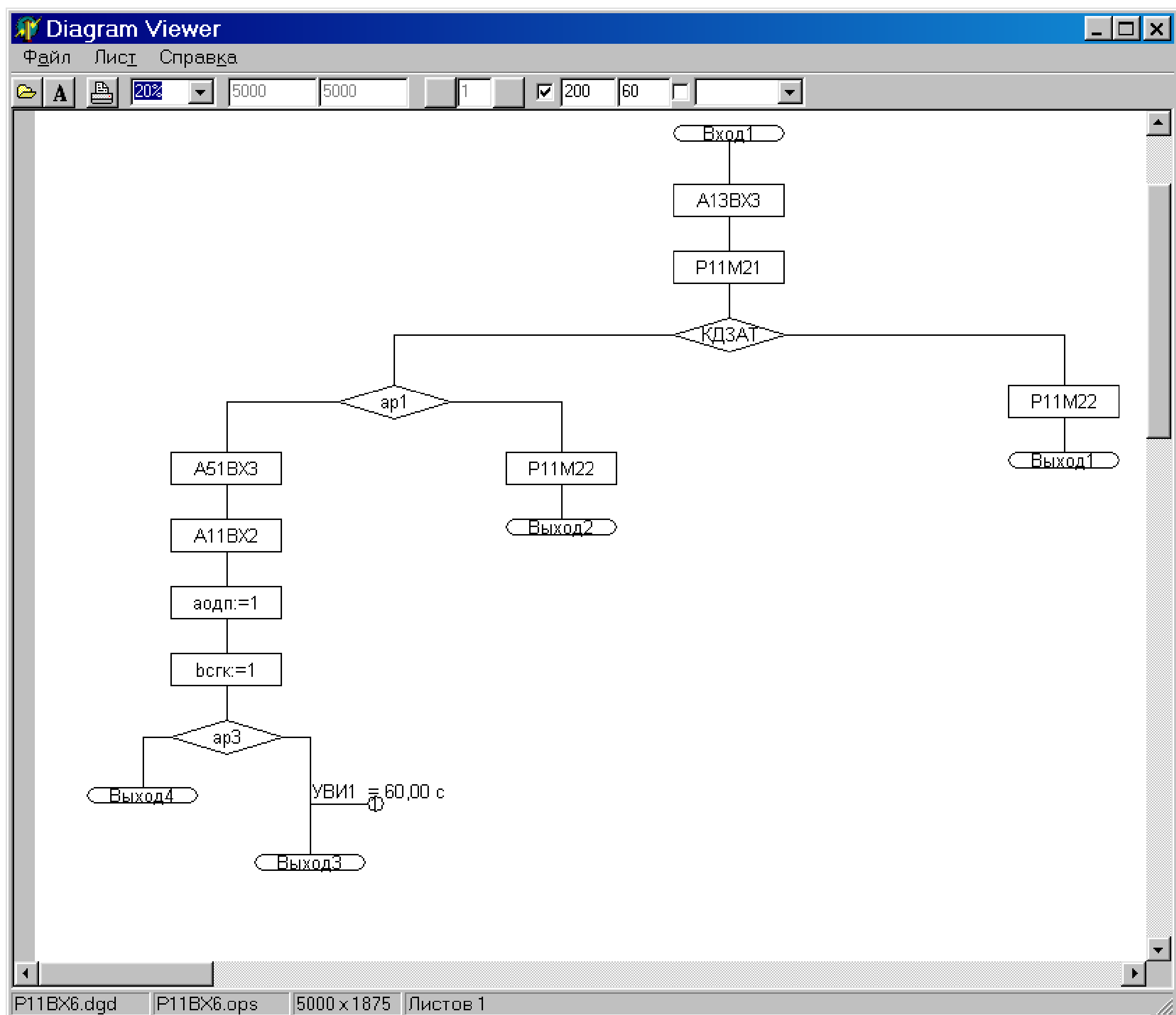

Рисунок 29. Отображение блок-схемы программы в системе ГРАФКОНТ

Блок-схемы алгоритмов стали основой также для их дальнейшей модификации и усовершенствования, в частности, при их приведении к некоторым эргономическим требованиям (запрет разрывов и блоков-соединителей и некоторых других конструкций), добавлении был получен графический язык ДРАКОН, являющийся одной из важнейших частей технологии разработки БПО «Графит-Флокс» в Центре автоматики и приборостроения им. Пилюгина.

7.2. Технология графосимволического программирования ГРАФ

На кафедрах информационных систем и технологий и программных систем Самарского государственного аэрокосмического университета была создана и развивается технология так называемого графосимволического программирования (ГСП) ГРАФ [16], использующая для описания программ в качестве базового графический язык. Первоначально языки и

поддерживающая его инструментальная система GRAPH разрабатывались как средство автоматизации программирования, ориентированное на создание программного обеспечения САПР технических изделий. Теоретической основной этого языка можно считать, с некоторыми допущениями, управляющий граф программы.

Вершины (прямоугольники), соответствующие выполнению операторов, соединяются линиями, отражающими передачу управления. При этом на линии надписывается условие (предикат), актуализирующий передачу управления в случае своей истинности.

Графическая модель программы строится в технологии ГРАФ с использованием некоторого исходного базиса, «строительного материала». В качестве такового выступают две категории: базовые модули и типы данных. Базовые модули представляют собой перечень локальных вычислимых функций, на основе которых в конечном итоге порождаются все объекты технологии ГРАФ - акторы, агрегаты и предикаты. Типы данных описывают семантический аспект строения данных, обрабатываемых базовыми функциями. Как формулируют авторы языка (А.Н. Коварцев), «если вкладывать в понятие программирование традиционный смысл, то в графосимволическом программировании оно начинается на этапе конструирования новых граф-программ из имеющегося набора вычислимых функций, точно так же, как программы, написанные на алгоритмических языках программирования, составляются из конечного набора операторов языка».

Построение исходного множества вычислимых функций (базовых модулей) производится при этом на некотором языке программирования, например, С++.

В технологии ГРАФ основными программными единицами являются объекты. По способу порождения и функциональному назначению различаются три типа объектов: акторы, агрегаты и предикаты. При этом они действуют в рамках некоторой «предметной области программирования».

Под предметной областью программирования понимается разработка программного обеспечения для некоторой области практических интересов (авиационные двигатели, бизнес, медицинские приборы и т.д.), использующая общую область данных и общую область знаний. В предметной области, как правило, заранее уже определен терминологический базис (параметров, переменных или констант).

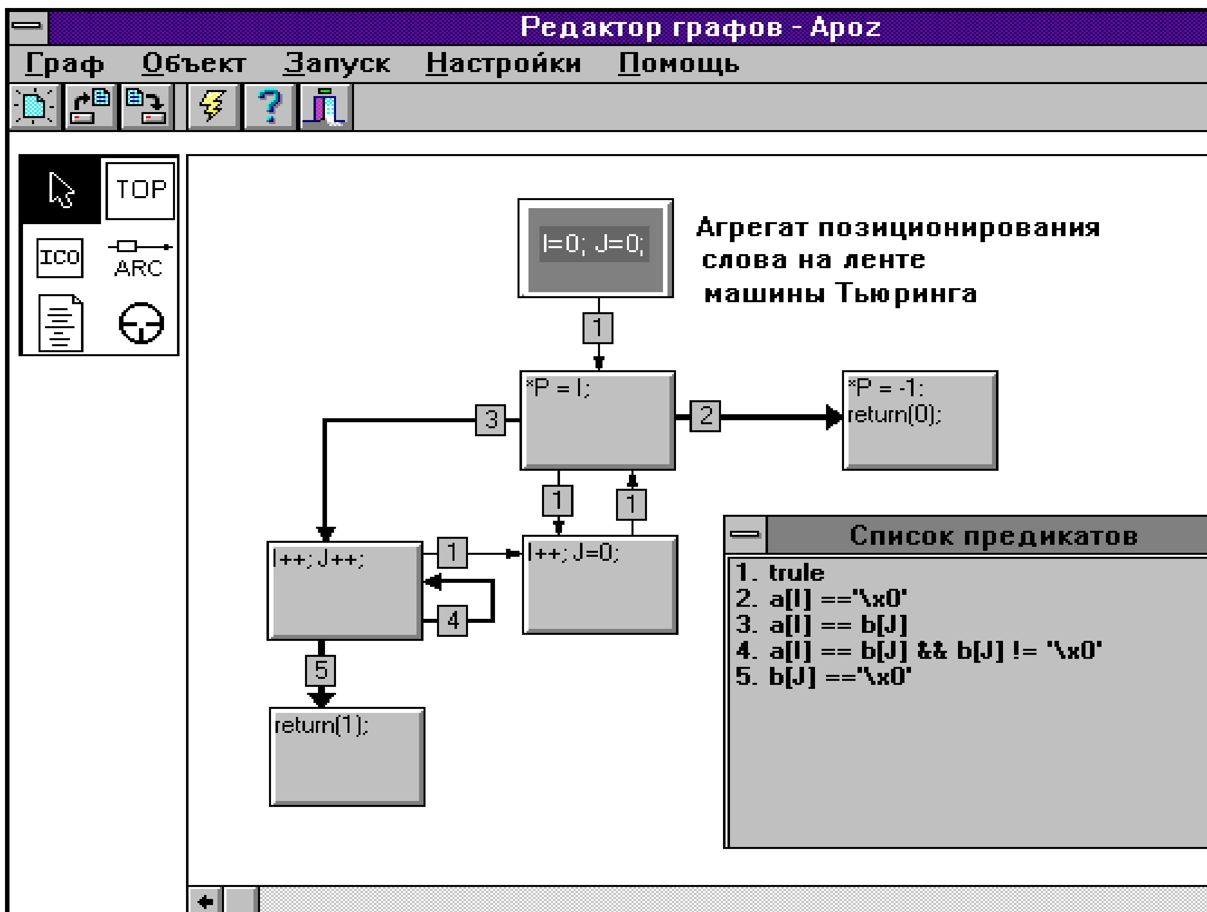

Рисунок 30. Пример графической программы в технологии ГРАФ

Так, в теории газотурбинных двигателей перечень параметров, их обозначение и содержание регламентировано государственными стандартами, которые во многом унифицированы с международными стандартами. Аналогичное положение дел имеет место в самолетостроении, ракетостроении, радиоэлектронике и т.д. Под предметной областью программирования в технологии ГРАФ понимается среда программирования, состоящая из общего набора данных (словарь данных) и набора программных модулей (словарь и библиотека программных модулей).

Программирование в рамках технологии ГРАФ начинается с формирования так называемого «словаря данных» предметной области программирования, который служит целям каталогизации данных, спецификации их семантики и областей значений. Словарь данных представляет собой таблицу, в которой каждому данному задается уникальное имя, тип, начальное значение и краткий комментарий его роли в предметной области. Технология ГРАФ имеет четкие стандарты описания и документирования программных модулей, представления и поддержки информационного обеспечения предметной области. Таким образом, для каждой предметной области строится единая информационная среда, позволяющая унифицировать проектирование написание программных модулей разными разработчиками.

Под объектом в технологии ГРАФ понимается специальным образом построенный в рамках технологии программный модуль, выполняющий определенные действия над данными предметной области.

Информационный интерфейс базовых модулей представляет собой отношение, устанавливающее связь между типами данных базового модуля (формальными параметрами), и данными предметной области. В результате установления связи порождаются новые объекта либо акторы, либо предикаты. Это отношение в технологии ГРАФ формируется "паспортизацией" типов данных базовых модулей. В этом смысле оно совместно с базовым модулем определяет понятие актора или предиката.

Структура автоматически компилируемого текста объекта-агрегата достаточно проста. Текст программы состоит из стандартного для всех объектов заголовка, структур данных, описывающих граф-программы, и обращения к стандартной программе – «граф-машине» [16]. С помощью компилятора языка Си текст агрегата компилируется в объектный модуль и помещается в библиотеку объектных модулей предметной области. Особенностью используемого способа построения межмодульного информационного интерфейса является то, что формируемые (автоматически) программные коды и информационные связи не зависят друг от друга. Модификация любого из объектов (актора, предиката или агрегата) не требует переделки кодов других объектов, входящих в предметную область. Более того, на самом деле при порождении акторов, предикатов или агрегатов никаких программных конструкций, описывающих информационный интерфейс, для них не создается. Акторы и предикаты на данном этапе - это лишь "фантомные" конструкции, не имеющие текстов программ, а описанные в виде "паспортов" в информационном фонде предметной области программирования.

7.3. Диаграммы деятельности и активностей в UML

Язык UML имеет несколько видов диаграмм, предназначенных для описания динамических аспектов систем, что соответствует общему представлению о том, что поведение может рассматриваться в нескольких аспектах. Поведение системы как смена состояний отражается в диаграмме конечных автоматов (состояний) языка UML (см. главу 6), взаимодействие объектов – в диаграммах последовательностей и взаимодействия. Наконец, поведение может характеризоваться некоторой последовательностью действий, исполняемых элементами, входящими в состав системы [31]. В такой постановке задача описания системы становится очень близкой задаче описания потока управления (control flow) в алгоритме. Вместе с тем, следует отметить, что предназначение диаграммы деятельности в языке UML более широкое, она позволяет описывать не только процесс исполнения некоторой программы с помощью ЭВМ, но и в широком смысле выполнение бизнес-

процессов в некоторой организации, причем ряд процессов может протекать параллельно.

Надо отметить, что по сравнению с диаграммами активностей предыдущих версий языка UML в версии UML 2.0 данный вид диаграмм претерпел ряд существенных изменений [31,32]. Например, вместо семантики конечных автоматов для моделирования деятельности в качестве базового используется механизм, подобный сетям Петри, что позволяет рассматривать дополнительно потоки объектов и данных. Отметим, что это изменение можно рассматривать, как недостаточно оправданное с точки зрения концептуальной чистоты усложнение. Фактически оно частично превращает диаграммы деятельности языка UML 2.0 в диаграммы потоков данных. Чрезмерная громоздкость и сложность языка UML 2.0 вызывает в настоящее время многочисленную критику [39]. Кроме того, на нотацию UML 2.0 оказала влияние нотация моделирования бизнес-процессов BPMN, что привело к горизонтальной ориентации диаграммы деятельности в UML 2.0 по сравнению с вертикальной ориентацией в языке UML 1.4.

Рисунок 31. Основные графические примитивы диаграмм деятельности UML 2.0

Центральным аспектом моделирования деятельности в языке UML 2.0 является последовательность действий и условий их выполнения, а также поток объектов, которые являются необходимыми для выполнения отдельных действий или являются их результатами. На рисунке 32 представлен пример диаграммы деятельности UML 1.4

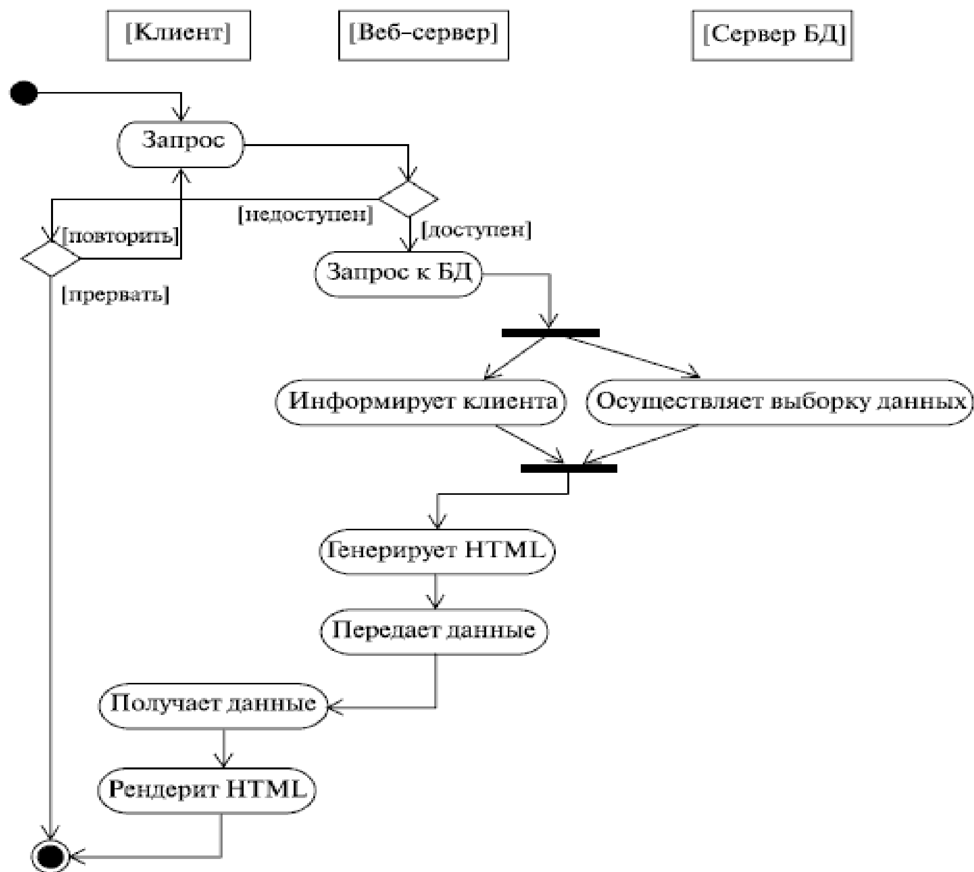

Рисунок 32. Пример диаграммы деятельности UML

Действие (action) представляет собой в UML элементарную единицу спецификации поведения, которая не может быть далее декомпозирована в форме деятельности.

Важным аспектом представления диаграмм деятельности в UML является возможность использования так называемых дорожек (swimlanes). На практике при моделировании бизнес-процессов дорожки наиболее часто соответствуют организационным единицам, выполняющим то или иное действие.

Особенностями диаграммы деятельности UML является наличие узлов синхронизации (разветвлений и соединений) потоков. Еще одной важной возможностью является вложенность деятельностей, т.е. некоторое действие может быть декомпозировано с помощью вспомогательной диаграммы деятельности, существуют также графические примитивы отображения вызова деятельности. Имеются символы передачи сигнала и приема события, что роднит (в том числе благодаря такому же внешнему виду), UML 2.0 с языком SDL. Возможно также, например, задание специального вида деятельности – обработчика исключения, сопоставляемого в качестве запасного некоторому другому примитиву деятельности на диаграмме.

Язык UML в настоящее время является индустриальным стандартом при разработке информационных систем, и поддержан целым рядом инструментальных средств, среди которых можно назвать Rational Rose, Borland Together, MagicDraw, Visual Paradigm, и др. Вариант Rational Rose

Real-time используется NASA, в частности, при создании программного обеспечения космического телескопа Вебба.

Тем не менее, в них средства автоматической генерации кода (текста программы) ориентированы прежде всего на генерацию шаблонов классов, и не предусматривается создания кода для логики управления, отображаемой диаграммой деятельности.

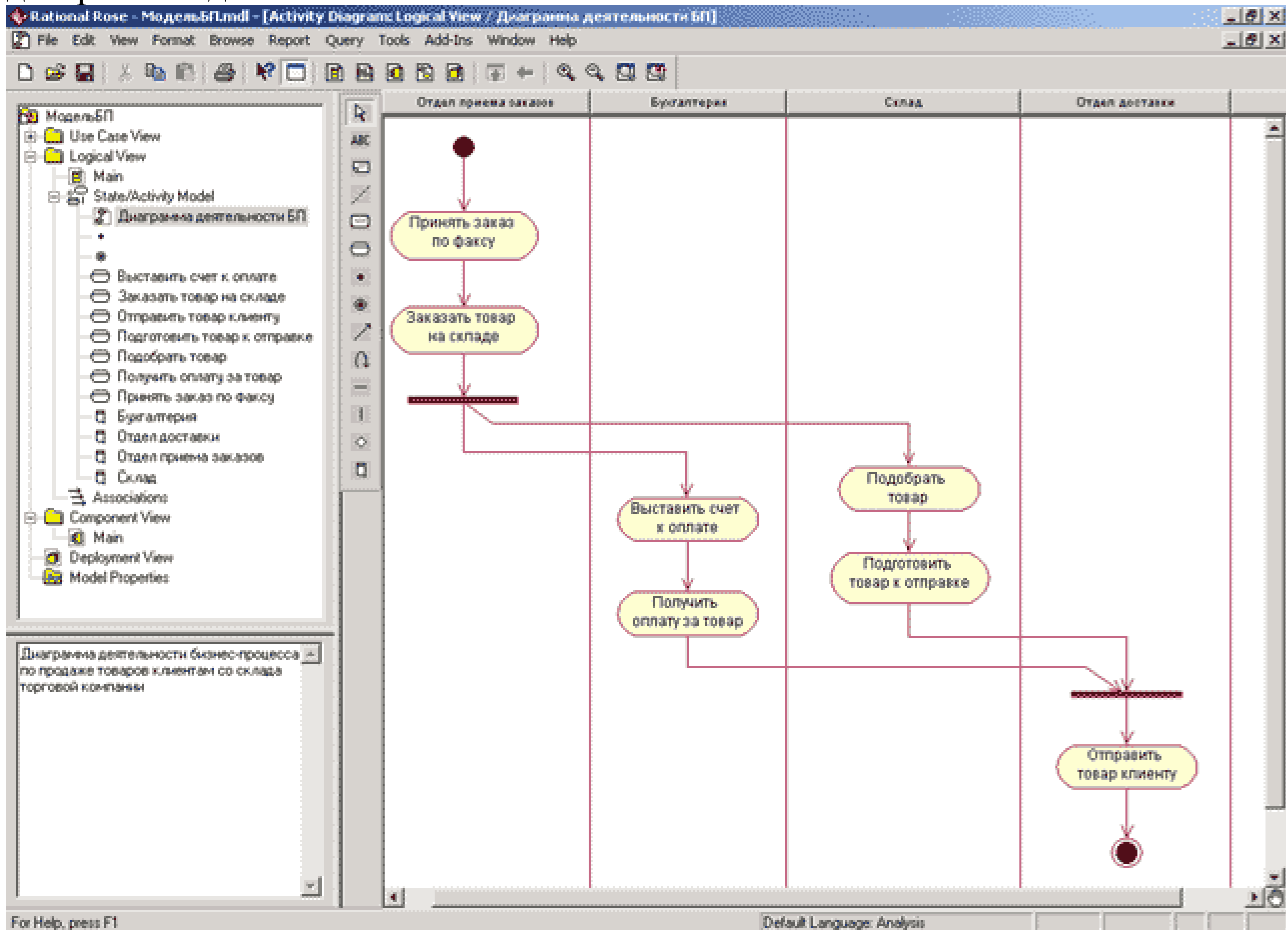

Рисунок 33. Диаграмма деятельности UML с «дорожками» в среде Rational Rose

Кроме того, диаграммы деятельности, несмотря на богатые выразительные возможности в версии UML 2.0, не имеют средств для отражения параметров реального времени.

В связи с этим применение диаграмм деятельности UML в качестве средства графического программирования управляющих алгоритмов реального времени, в том числе - для БВС КА, не представляется целесообразным.

7.4. Язык потоковых диаграмм FC (Flow Chart) системы IsaGraf

Многие производители инструментальных средств, поддерживающих стандарт МЭК 61131-3, не ограничиваются поддержкой языков стандарта. Так, например, в популярный пакет IsaGraf включен язык FC (Flow Chart, или

язык потоковых диаграмм). Программы на этом языке описываются в виде граф-схем алгоритмов (см. рисунок 34).

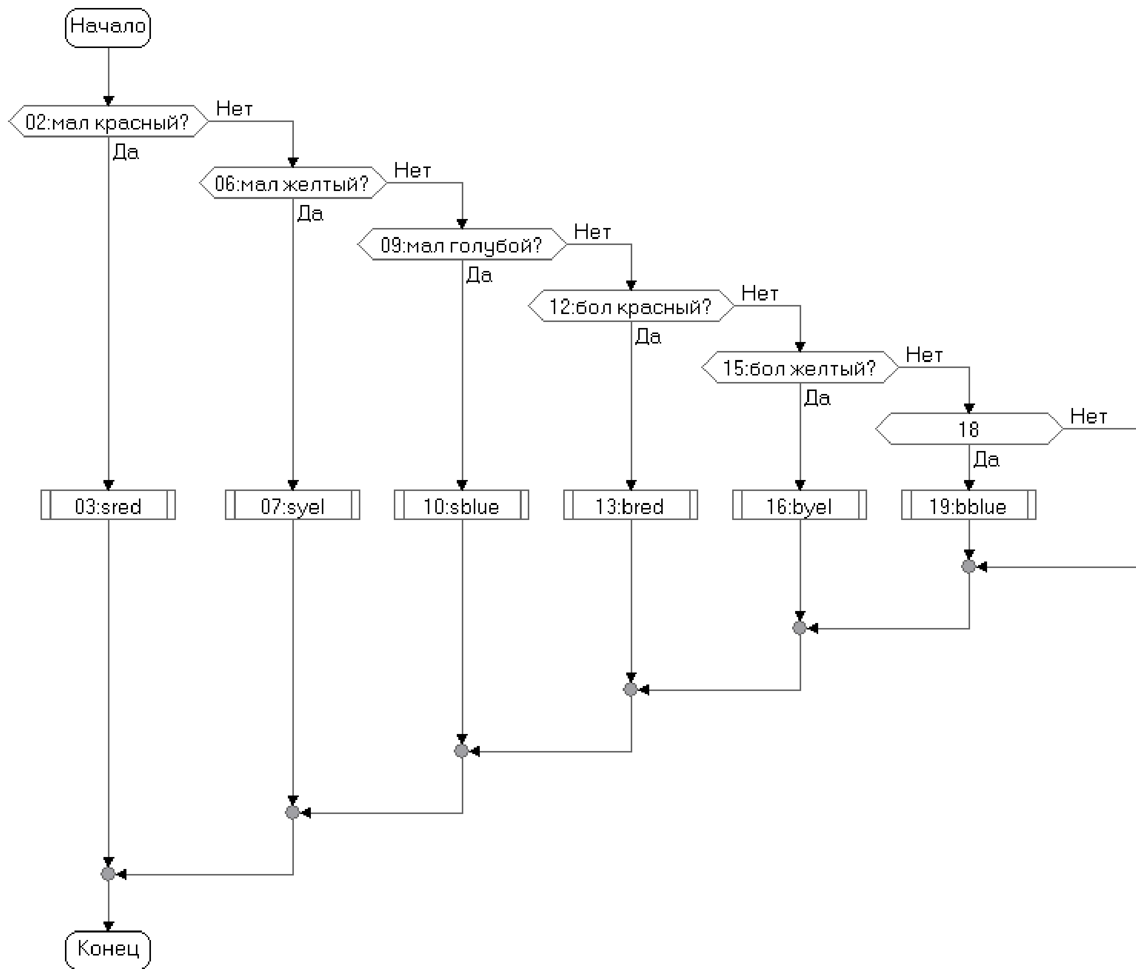

Рисунок 34. Графический язык Flow Chart системы IsaGraf

Язык FC, как видно из представленного примера, является типичным языком отражения потока управления (control flow), с несколько измененной по сравнению со стандартными блок-схемами системой обозначений. При этом в системе IsaGraf подразумевается использование языка FC совместно с языками стандарта МЭК 61131-3, в частности, конкретные действия, выполняемые блоками, описываются на языке ST.

7.5. Язык ДРАКОН и технология ГРАФИТ-ФЛОКС

Язык ДРАКОН разработан НПЦ автоматики и приборостроения имени Н.А. Пилюгина и Российской академией наук (Институт прикладной математики имени М.В. Келдыша) как обобщение опыта работ по созданию БПО космического корабля «Буран». На базе ДРАКОНА в НПЦ АП построена автоматизированная технология проектирования программных

систем под названием «ГРАФИТ-ФЛОКС» [26]. Данная технология была успешно использована в ряде космических проектов: «Морской старт», «Фрегат», «Протон-М» и др. Несмотря на «космическое» происхождение, авторами языка он активно популяризируется и декларируется его широкая применимость в самых разных сферах человеческой деятельности – от описания технологических процессов до обучения, что приводит к несколько спорным примерам (см. рисунок 35).

В языке ДРАКОН используются два типа элементов – графоэлементы и текстовые надписи (текстоэлементы). Графоэлементы языка ДРАКОН называются иконами, которые могут объединяться в составные иконы - макроиконы. Шампур-блок - часть ДРАКОН-схемы, имеющая один вход в верхней части диаграммы и один выход внизу, включающая элементы, расположенные на одной вертикали. «Ветки» - смысловые блоки, куски алгоритма, представляющие составной оператор языка ДРАКОН. С помощью веток удобно формализовать смысловое разбиение проблемы, программы или процесса на части и дать частям удобные смысловые названия.

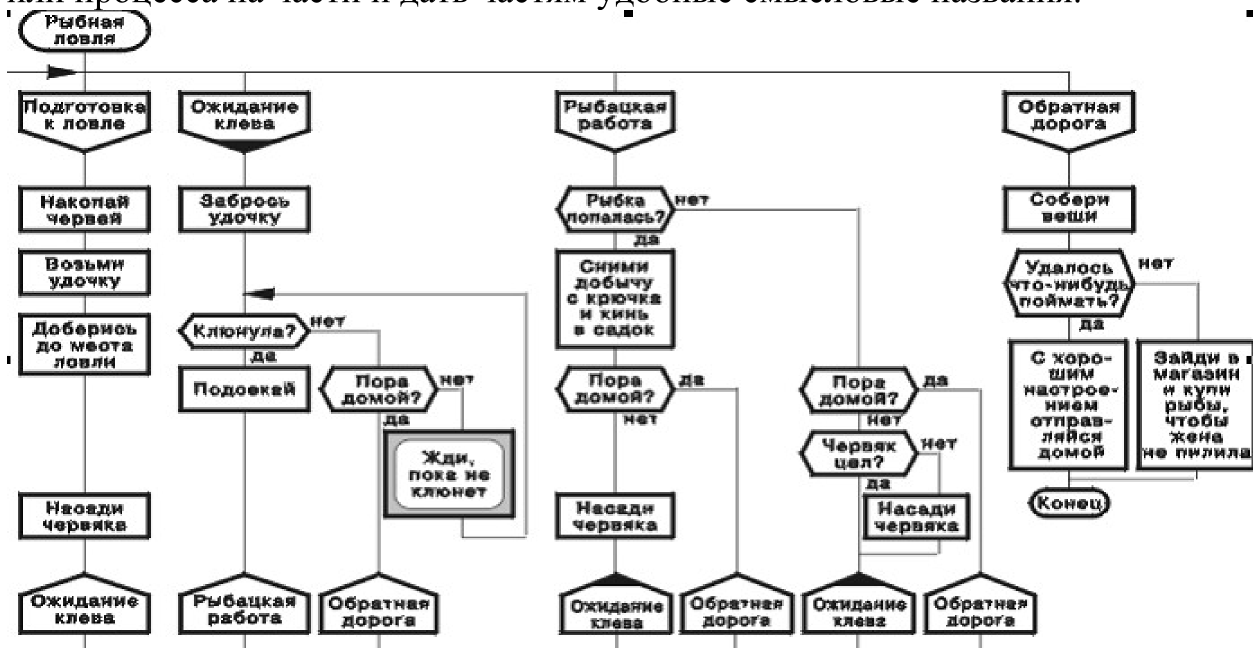

Рисунок 35. Пример схемы на языке ДРАКОН

Алгоритмическая часть описания программы на ДРАКОНЕ отделяется от описания декларативного (в частности, структур данных). При практическом использовании в НПЦ АП имени Пилюгина помимо описания программы в среде ГРАФИТ используется описание на специальном языке ФЛОКС. Это делает при необходимости возможным создание гибридных визуальных языков программирования, например, ДРАКОН-ПАСКАЛЬ. В таких языках к визуальному синтаксису языка ДРАКОН по определенным правилам присоединяется текстовый синтаксис (например, языка Pascal).

При этом во многом удается снизить зависимость от квалификации и результативности труда программистов при разработке БПО за счет того, что в значительной степени создание программ для ракет перекладывается на плечи специалистов по управлению бортовыми системами, которые не являются программистами. Как пишет автор языка В.Д. Паронджанов, в

технологии приближаются к принципу «программирования без программистов» [26]. Причина этого в том, что при решении практических прикладных задач специалисты по управлению бортовыми системами досконально владеют материалом и хорошо знают постановку задачи. В отличие от них программисты могут не владеть в достаточной степени «физикой» протекающих на борту КА процессов. Это позволяет значительно сократить издержки, ускорить ход работ и сократить число ошибок по принципу «испорченного телефона», вызванных взаимным непониманием между специалистами по управлению и программистами.

Несмотря на то, что в качестве графоэлементов в ДРАКОНЕ используются несколько иные «иконы», нежели в классических блок-схемах алгоритмов, а его авторами блок-схемы критикуются за ряд недостатков (для устранения которых на диаграммы ДРАКОНА накладывается ряд ограничений по правилам начертания), несомненно, что фактически ДРАКОН основан на языке блок-схем алгоритмов и программ.

Говоря о возможной применимости языка ДРАКОН для решения задач создания БПО КА ДЗЗ, следует отметить следующее. В технологии разработки сложных комплексов управляющего ПО, в том числе – БПО КА ДЗЗ, подразумевается параллельное выполнение множества программ на борту под управлением операционной системы реального времени. При этом одна и та же управляющая программа может иметь несколько «входов», т.е. допускать многократное включение.

В языке ДРАКОН, несмотря на наличие операций реального времени, понятие «входа» не предусматривается. Это принципиальный момент, не позволяющий напрямую применить ее при создании БПО КА ДЗЗ.

Следует также отметить, что УА РВ могут выполнять различные по своей природе операции - выдача команд управления, занесение запросов на выполнение программ, формирования признаков в программной памяти, и др.

Эту особенность можно отразить и, задействуя на полную мощь возможности графического подхода, эффективно применить за счет использования разных по начертанию графических примитивов. ДРАКОН этого не предусматривает, для всех операций используется один и тот же графоэлемент - прямоугольник. Это затрудняет понимание логики УА РВ при его описании в нотации ДРАКОНА по сравнению, например, с нотацией, используемой в технологии ГРАФКОНТ [10].

Еще одним моментом, отличающим диаграммы ДРАКОНА, является строго вертикальная ориентация этих схем. В то же время, горизонтальная ориентация позволяет более четко отразить структуру управляющей программы, особенно во временном аспекте.

7.6. Язык Р-схем

Еще одним классическим примером графического языка, описывающего структуру передачи управления, является язык Р-схем, разработанный в СССР еще в 1970-х годах Вельбицким [25] и его коллегами и применявшийся некоторое время на некоторых предприятиях, в том числе, общего машиностроения, при документировании создаваемого программного обеспечения. Эти схемы отличаются от большинства стандартов построения блок-схем тем, что используется граф, нагруженный по дугам, в то время как традиционная блок-схема использует граф, нагруженный по вершинам (узлам). Такие схемы примерно на 25% компактнее классических блок-схем. На рисунке 36 представлены два примера, показывающих представление условной и циклической конструкции программы в виде схем Вельбицкого.

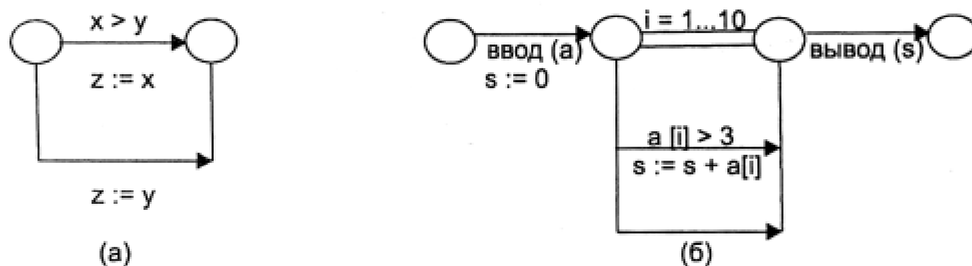

Рисунок 36. Примеры условной и циклической Р-схем Вельбицкого

Использование Р-схем для графического описания УА РВ затрудняет тот факт [30], что она не предусматривает средств адаптации к специфике конкретной БЦВМ, а также отсутствие возможностей для описания параллельного функционирования программ в режиме реального времени.

7.7. Algorithm Builder для микроконтроллера Atmel

Одной из, несомненно, удачных реализаций графической среды и используемого в ней языка при программировании промышленных контроллеров следует признать «визуальный ассемблер» Algorithm Builder, целевой платформой которого являются 8-битные микроконтроллеры фирмы Atmel [17,40]. Algorithm Builder – яркий представитель систем визуального программирования специального назначения. Интересным является тот факт, что базовым языком программирования, на котором производится генерация текста программы по графическому описанию, служит язык ассемблер. Ограниченные ресурсы 8-битных микроконтроллеров и стремление разработчиков встраиваемого ПО добиться максимальной функциональности при минимуме ресурсов оправдывают его использование во многих проектах. Визуальная нотация при этом в большинстве случаев позволяет добиться не только достаточно высокого уровня показателей качества кода, но и лучшей его «читаемости», что приводит к большей пригодности к его сопровождению, что немаловажно при создании устройств с продолжительным сроком эксплуатации. Она также позволяет преодолеть

типично высокий для ассемблера барьер понимания путем применения «подпрограмм в картинках» с упрощенными механизмами выбора используемых конструкций.

Таким образом, исключается «промежуточное звено» в виде языка программирования высокого уровня. Среда предназначена для производства полного цикла разработки начиная от ввода алгоритма, включая процесс отладки и заканчивая программированием кристалла. При этом допускается разработка программы как на уровне ассемблера, так и на макроуровне (с манипуляцией многобайтными величинами со знаком).

В отличие от простого использования классического ассемблера программа вводится в виде алгоритма с древовидными ветвлениями и отображается на плоскости, в двух измерениях. Сеть условных и безусловных переходов отображается графически, в удобной векторной форме. Программа к тому же освобождается от многочисленных имен меток, которые в классическом ассемблере неизбежны. Становится наглядной логическая структура программы. По сведениям разработчиков [40], графические технологии раскрывают новые возможности для программистов. Визуальность логической структуры уменьшает вероятность ошибок и сокращает сроки разработки. По оценке пользователей, по сравнению с классическим ассемблером, время на разработку программного обеспечения сокращается в 3-5 раз.

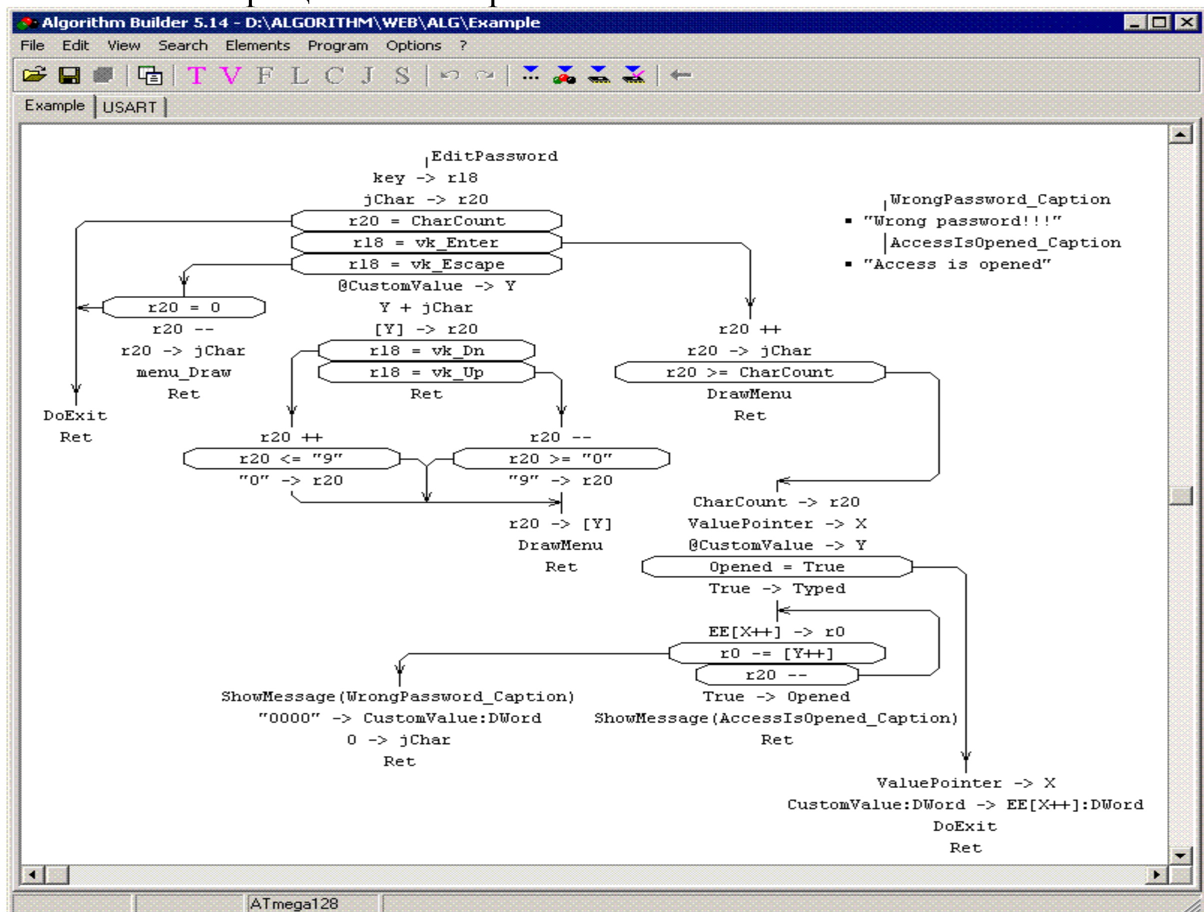

Рисунок 37. Графический язык среды Algorithm Builder

Пример графической программы в среде Algorithm Builder приводится на рисунке 37.

Среда объединяет в себе графический редактор, компилятор алгоритма, симулятор микроконтроллера, внутрисхемный программатор. При использовании внутрисхемного программатора микроконтроллер подключается к COM порту компьютера через адаптер.

Языки графического описания структур данных

Наряду с графическими языками, описывающими процессы, в программировании важную роль играют графические средства описания структур данных. С точки зрения их применения не только для документирования, но и генерации программ можно отметить, что после описания структуры данных в графической нотации некоторые инструментальные средства позволяют автоматически получать:

1) пригодные для практического применения описания структуры БД на языке SQL.
2) генерировать на том или ином языке программирования экранные формы, сопоставляемые полям БД и позволяющие манипулировать данными.

Классическим средством такого рода является ERwin.

8.1. Язык ER-диаграмм и стандарт IDEF1X

ERwin Data Modeler - это CASE-средство для проектирования и документирования баз данных, которое позволяет создавать, документировать и сопровождать базы данных, хранилища и витрины данных. Модели данных помогают визуализировать структуру данных, обеспечивая эффективный процесс организации, управления и администрирования таких аспектов деятельности предприятия, как уровень сложности данных, технологий баз данных и среды развертывания. ERwin позволяет наглядно отображать сложные структуры данных. Удобная в использовании графическая среда упрощает разработку базы данных и автоматизирует множество трудоемких задач, уменьшая сроки создания высококачественных и высокопроизводительных транзакционных баз данных и хранилищ данных. Данное решение улучшает коммуникацию в организации, обеспечивая совместную работу администраторов и разработчиков баз данных, многократное использование модели, а также наглядное представление данных в удобном для понимания и обслуживания формате.

ERwin имеет два уровня представления модели - логический и физический. На логическом уровне данные не связаны с конкретной СУБД, поэтому могут быть наглядно представлены даже для неспециалистов, Физический уровень данных - это по существу отображение системного каталога, который зависит от конкретной реализации СУБД. ERwin позволяет проводить процессы прямого и обратного проектирования БД. Это

означает, что по модели данных можно сгенерировать схему БД или автоматически создать модель данных на основе информации системного каталога. ERwin взаимодействует с такими популярными средствами программирования, как PowerBuilder, Visual Basic, Delphi и позволяет автоматически генерировать текст программы, который готов к компиляции и выполнению. При этом для разных сред разработки реализована различная техника кодогенерации.

Теоретической основой ERwin является модель «сущность-связь», предложенная Ченом. ER-диаграммы затем были стандартизированы в стандарте IDEF1X. Прямоугольниками принято отображать элементы данных («сущности»), линиями – связи между ними (над линией надписывается, какая именно имеет место связь). Например, элемент данных «профессор» может связываться с элементом данных «студент» с помощью связи «преподает», а «студент» - с элементом «группа» связью «принадлежит», и т.д.

На рисунке 38 показан экран во время работы программного средства ERwin

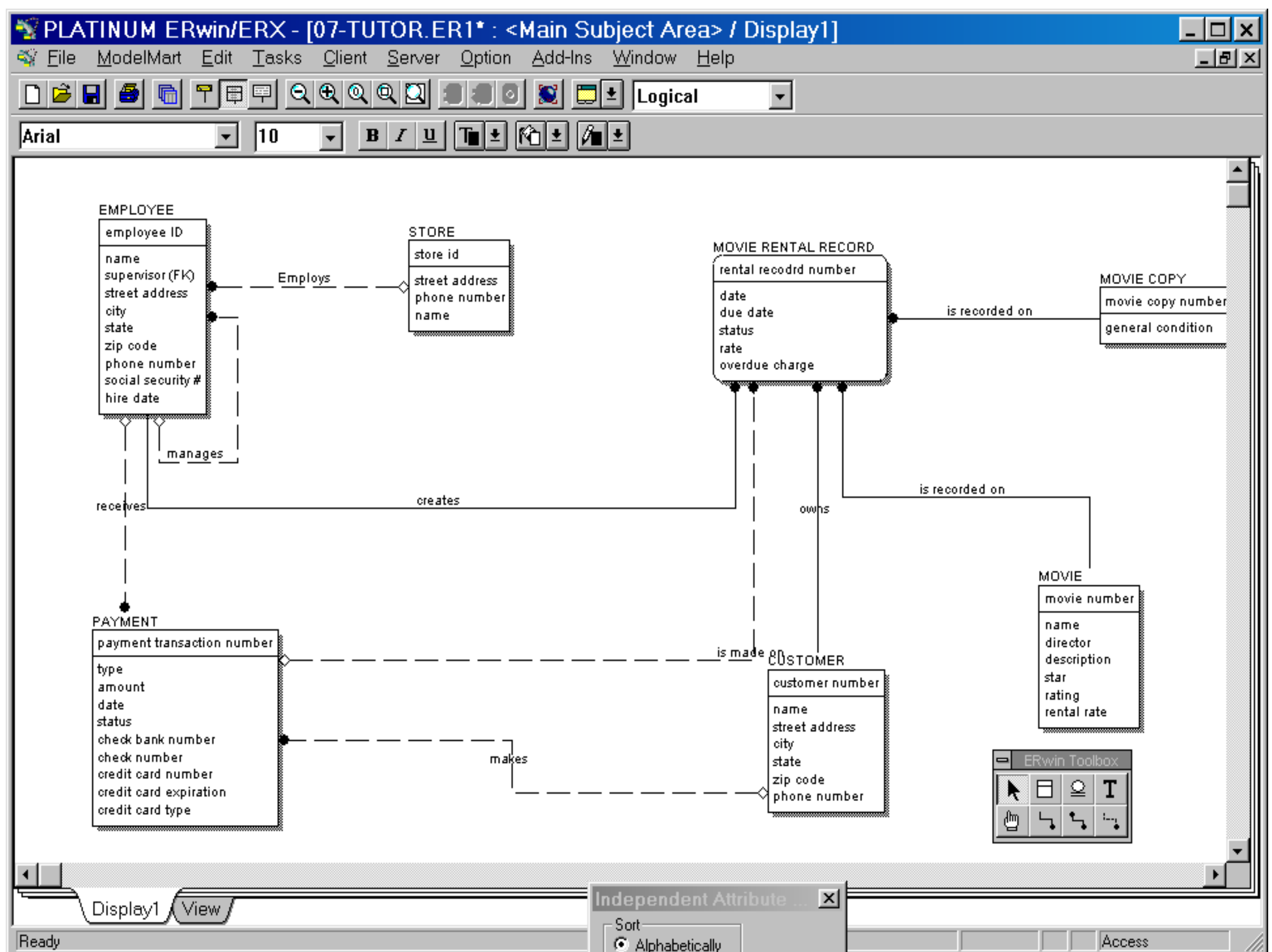

Рисунок 38. Моделирование данных графическими средствами ERwin (ER-диаграммой)

8.2. Диаграммы классов UML

Как уже отмечалось, UML представляет собой комбинированную методологию с использованием большого числа диаграмм, представляющих проектируемую систему в различных аспектах. При этом, конечно, разные виды диаграмм имеют разную степень важности и частоты использования при практическом применении UML. Диаграмма классов представляет собой, можно сказать, «сердцевину», «ядро» технологии использования UML (технологии разработки ПО, называемой RUP – Rational Unified Process), в отличие от многих других видов диаграмм она практически всегда присутствует в UML-ориентированной документации.

Диаграмма классов описывает классы и объекты создаваемой системы. Классы можно рассматривать как структуры данных с точки зрения нашей классификации (хотя собственно модель данных в этом случае дополняется перечислением методов, применимых к данным). Класс на диаграмме изображается в виде прямоугольника, разделенного горизонтальными линиями на три части. В первой части указывается название класса. Как правило, имя класса состоит из одного, максимум двух слов. Вторая часть содержит перечень атрибутов класса, которые характеризуют тот или иной объект этого класса в модели предметной области. Третья часть содержит перечень операций, отражающих его поведение в модели предметной области (рис. 3.1). Пример диаграммы классов UML приводится на рисунке 39.

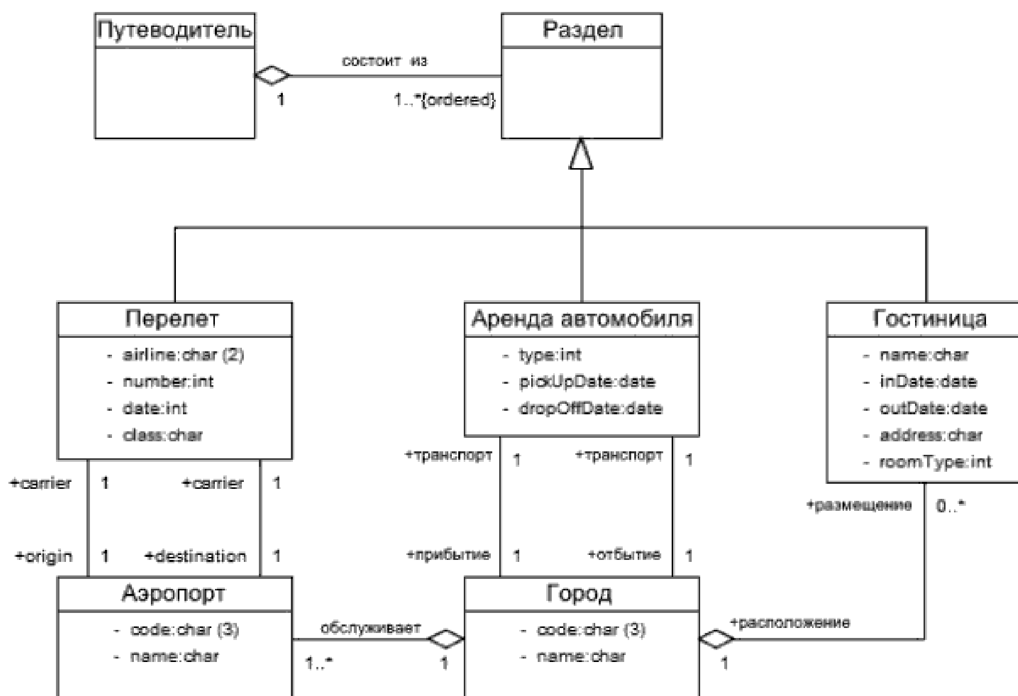

Рисунок 39. Пример диаграммы классов языка UML

Фактически диаграмма классов UML является достаточной заготовкой для генерации шаблонов классов на одном из популярных объектно-ориентированных языков программирования (C++, Java, C#, и т.д.), что и используется в большом количестве CASE-средств и интегрированных сред разработки программ IDE (например, Borland Development Studio, Microsoft Visual Studio). **Именно о генерации шаблонов (заголовков) классов без фактической реализации их внутренней логики по диаграмме классов UML и говорится обычно, когда рекламируется возможность «автоматической генерации кода по визуальным диаграммам»**.

Диаграмма классов является основным документом, содержащим информацию об архитектуре объектно-ориентированной программной системы.

На диаграмме классов отображаются, помимо пиктограмм, отображающих сами классы, зависимости между классами. Однако зависимости на диаграммах изображаются не всегда, а только в тех случаях, когда их отображение является важным для понимания модели. Часто они лишь подразумеваются, т. к. логически следуют из природы классов. Зависимости между классами могут быть нескольких основных видов. Базовый вид отношений между объектами - это ассоциация. Это некоторая связь между объектами, которая может отражать широкий спектр различных по семантике зависимостей. Ассоциация может иметь имя, показывающее природу отношений между объектами, при этом связь может быть направленной, данный факт может отражаться при помощи треугольного маркера. Однонаправленная ассоциация может изображаться стрелкой. Кроме направления ассоциации, на диаграмме могут быть указаны роли, которые каждый из вступающих в зависимость классов играет в данном отношении, и кратность, то есть количество объектов, связываемых данным отношением. Сама ассоциация может иметь какие-то свойства, и в этом случае для ее описания может быть выделен специальный класс, отображаемый соответствующей пиктограммой на диаграмме, связываемой в данном случае с линией отношения пунктиром. Специфическое отношение между классами - связь типа "часть-целое" (например, двигатель и автомобиль). Такой вид связи называется агрегацией. При этом выделяют простое и композитное агрегирование и говорят о собственно агрегации и композиции. Композиция – более сильная разновидность агрегации, когда подразумевается, что часть может принадлежать лишь одному объекту-владельцу, и при его уничтожении автоматически уничтожаются все объекты-части. Еще одно важнейшее отношение между классами на диаграмме UML – это ассоциация вида «являться», или «IS», отражающее наследование, или иерархию классов. О наследовании идет речь, когда один класс порождается от другого, с копированием (возможным переопределением части и добавлением новых) его свойств и методов.

9. Языки описания потоков данных (dataflow)

Как отмечалось выше, значительную группу графических языков программирования образуют языки, ориентированные на описание потоков данных. При этом могут подразумеваться получатели и отправители этих потоков различного класса. Упор может быть сделан на передаче данных между процессами (как в диаграммах DFD), или на передаче информации от одной структурной единицы информационной системы (сервер, ЭВМ, БД, банк данных, терминал, и др.), к другой.

В значительной степени распространение языки описания потоков данных получили благодаря использованию при создании средств промышленной автоматики стандарта МЭК61131-3. Интерес представляет история появления этого стандарта. К концу 1980-х годов на рынке присутствовали сотни инструментальных средств программирования ПЛК (фактически, каждый производитель ПЛК имел собственное, несовместимое с другими средство программирования своих контроллеров). В этих условиях создание единого стандарта программирования ПЛК стало ключевой задачей, стоящей перед отраслью. Эту задачу взяла на себя Международная Электротехническая Комиссия (МЭК). Результатом стала разработка стандарта МЭК 61131, состоящего из восьми частей, из которых наиболее важной является третья часть - стандарт МЭК 61131-3, описывающий языки программирования ПЛК. Целью создания стандарта была унификация языков программирования ПЛК и предоставления разработчикам ряда аппаратно-независимых языков, что, по замыслу создателей стандарта, обеспечило бы простоту переносимости программ между различными аппаратными платформами и снимало бы необходимость изучения новых языков и средств программирования при переходе разработчика на новый ПЛК. Первая версия стандарта была опубликована в 1993 году, в настоящее время действует вторая редакция стандарта, вступившая в силу в 2003 году.

Однако в настоящее время можно утверждать, что поставленные цели в полном объеме не были достигнуты. Группировка в рамках одного стандарта пяти различных языков дезориентирует пользователей и затрудняет изучение стандарта. Языки во многом дублируют функции друг друга.

Стандарт определяет лишь внешний вид графических языков, в результате фактически, инструментальные средства различных производителей программно несовместимы между собой. Они используют различные, несовместимые форматы файлов проекта, что делает невозможным прямой перенос проекта с одного инструментального средства на другое (следует отметить, что организация PLCOpen разрабатывает стандарт представления проектных файлов МЭК на базе XML, однако существенного распространения этот стандарт не получил).

LD (Ladder Logic, МЭК61131-3)

Язык LD (Ladder Diagram) является графическим языком, программа на котором представляет собой некий аналог релейно-контактной схемы (электрической схемы). Пример программы на данном языке приведен на рисунке 40.

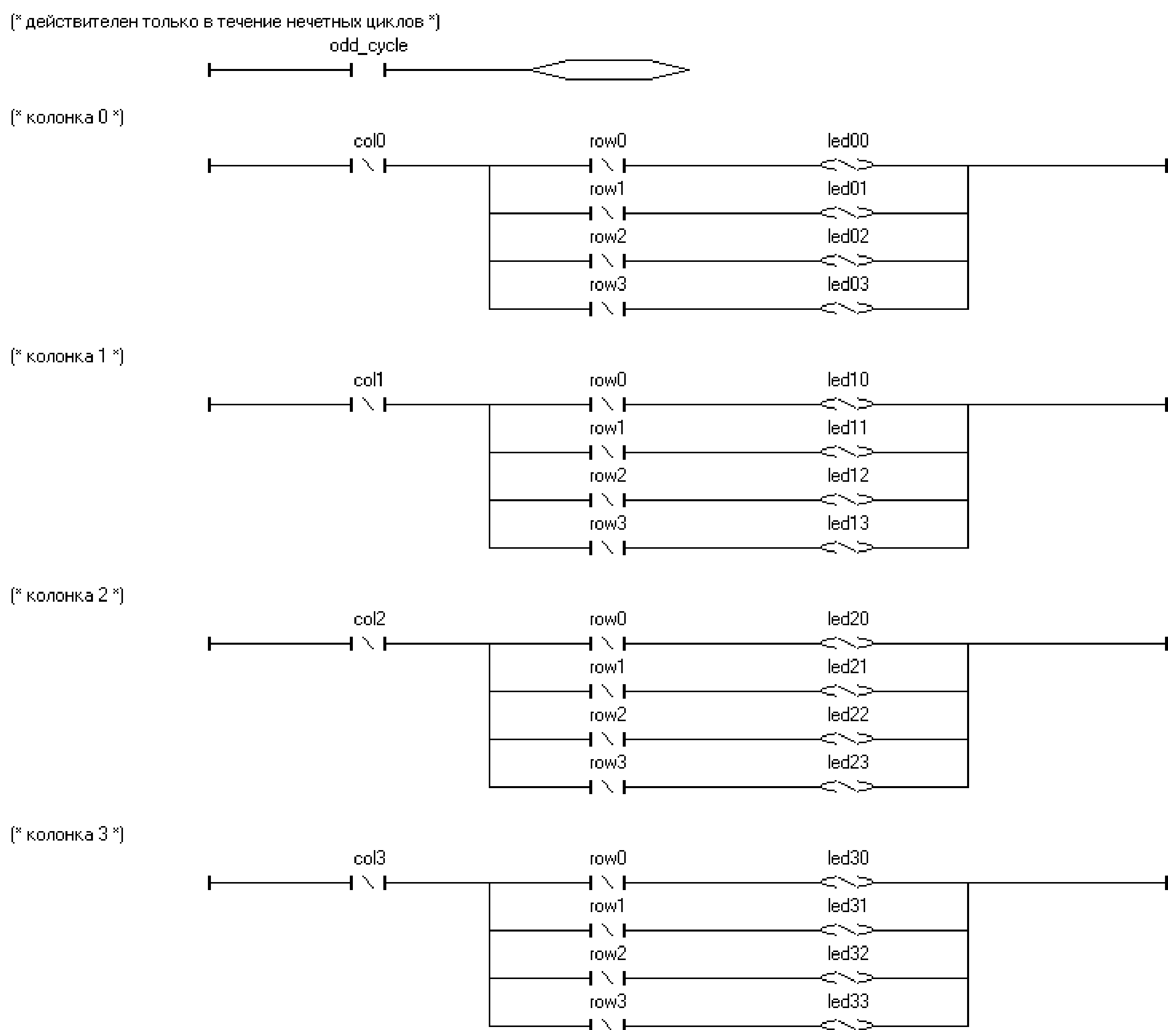

Рисунок 40. Пример диаграммы на графическом языке LD

По замыслу авторов стандарта, такая форма представления программы способствует переходу специалистов в области релейной автоматики на ПЛК. Однако в современных условиях, когда релейно-контактные устройства практически вытеснены из промышленной автоматики (оставаясь

доминирующими лишь в некоторых специфических отраслях, таких, как железнодорожный транспорт), данный подход вряд ли представляется целесообразным. При большом количестве «реле» в схеме она становится сложной для понимания, анализа и построения. К тому же, в такой программе могут возникать ситуации, в которых ее поведение будет зависеть от порядка расположения цепей в «лестнице», аналогично тому, как в реальной релейно-контактной схеме ее поведение в ситуации логических гонок зависит от фактического порядка срабатывания реле.

Еще одним недостатком языка LD заключается в следующем. Язык, эксплуатирующий аналогию с релейно-контактными схемами может быть эффективно использован только для описания процессов, имеющих «двоичный» характер. Лестничные диаграммы сложны в отладке и сопровождении в силу своей неструктурированной природы.

FBD (МЭК61131-3)

Язык FBD (Functional Block Diagram, Диаграмма Функциональных Блоков) является графическим функциональным языком программирования, так же, как и LD, использующим аналогию с электрической (принципиальной) схемой. Программа на языке FBD представляет собой совокупность функциональных блоков, соединенных линиями связи («цепями»), которые обычно фактически являются переменными программы. Каждый блок представляет собой функцию и может иметь, в общем случае, произвольное количество входов и выходов. Начальное значение переменных–цепей задаются с помощью специальных блоков – входов или констант, выходные цепи могут быть связаны либо с физическими выходами контроллера, либо с глобальными переменными программы. Пример программы на языке FBD приведен на рисунке 41.

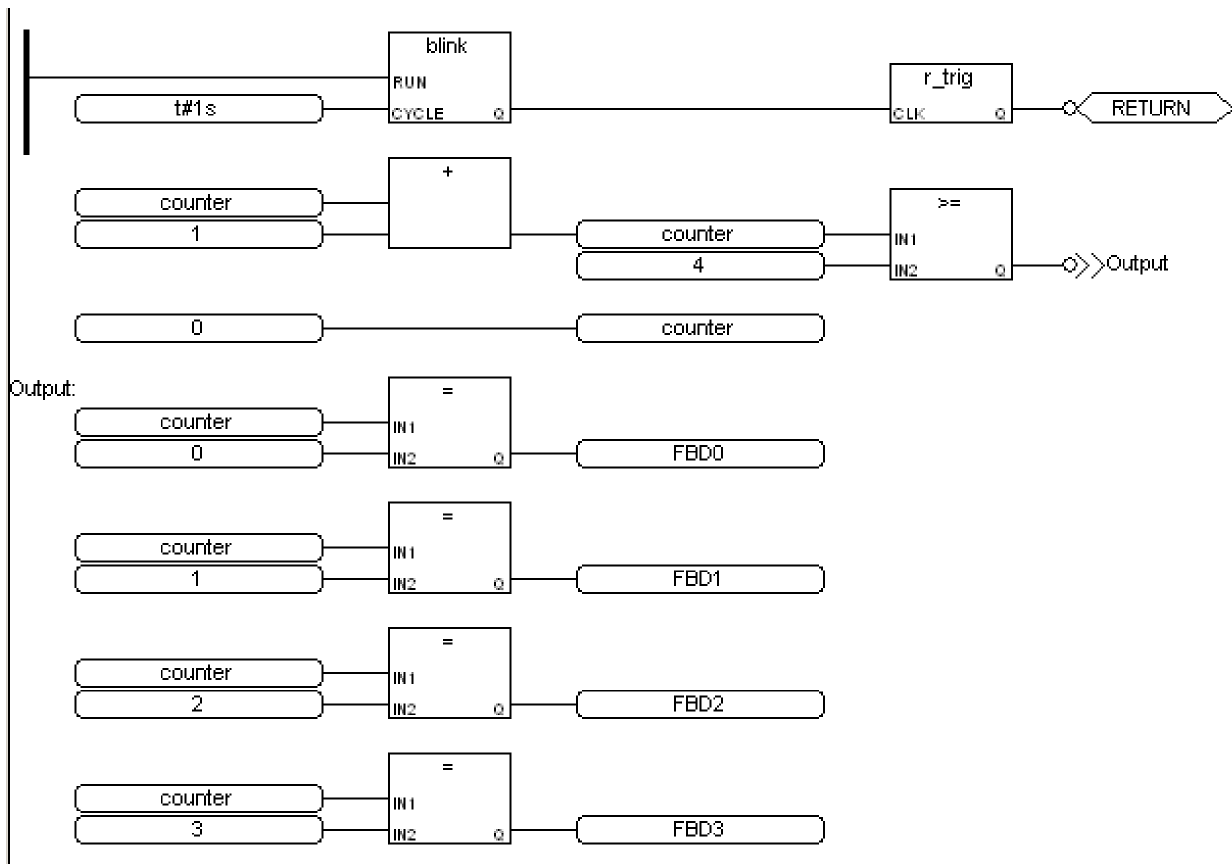

Рисунок 41. Графическая программа на языке FBD

На практике язык FBD является наиболее часто используемым языком стандарта МЭК. Графическая форма представления, простота в использовании, легко осуществляемое повторное использование функциональных диаграмм делают язык FBD незаменимым при разработке программного обеспечения ПЛК.

Вместе с тем, нельзя не отметить и некоторые недостатки FBD. **Хотя FBD обеспечивает легкое представление функций обработки как «непрерывных» сигналов, в частности, функций регулирования, так и логических функций, в нем неудобным и неочевидным образом описывается поведение системы во времени**, т.е. реализуются те участки программы, которые было бы удобно представить в виде конечного автомата. Это фактически исключает возможность его использования при создании бортовых УА РВ для КА ДЗЗ.

Также, к сожалению, надо отметить, что общие недостатки стандарта МЭК 61131-3 в значительной степени относятся именно к языку FBD. В частности, разные инструментальные средства обладают различными наборами функциональных блоков языка FBD (за исключением небольшого количества базовых блоков), поэтому при переносе проекта на другое инструментальное средство все диаграммы функциональных блоков придется строить и отлаживать заново. Более того, набор функциональных блоков даже в рамках одного инструментального средства не фиксирован, производители ПЛК поставляют собственные библиотеки функциональных блоков, поэтому нельзя говорить о программной переносимости проектов

между ПЛК различных семейств, даже в случае использования одного и того же инструментального средства программирования.

DFD (Data Flow Diagram)

Вероятно, наиболее известной графической нотацией среди языков описания потоков данных в программировании является нотация DFD – Data Flow Diagram. Она является базовой во многих CASE-системах и может использоваться как для описания как произвольных бизнес-процессов, так и программ. Основными элементами диаграмм потоков данных являются внешние сущности; процессы; накопители данных; потоки данных.

Пример DFD-диаграммы приводится на рисунке 42. Под внешней сущностью (External Entity) понимается материальный объект, являющийся источником или приемником информации. В качестве внешней сущности на DFD диаграмме могут выступать заказчики, поставщики, клиенты, склад, банк и другие. К сожалению, DFD методология не оформлена как стандарт. По этой причине в диаграммах потоков данных используются различные условные обозначения.

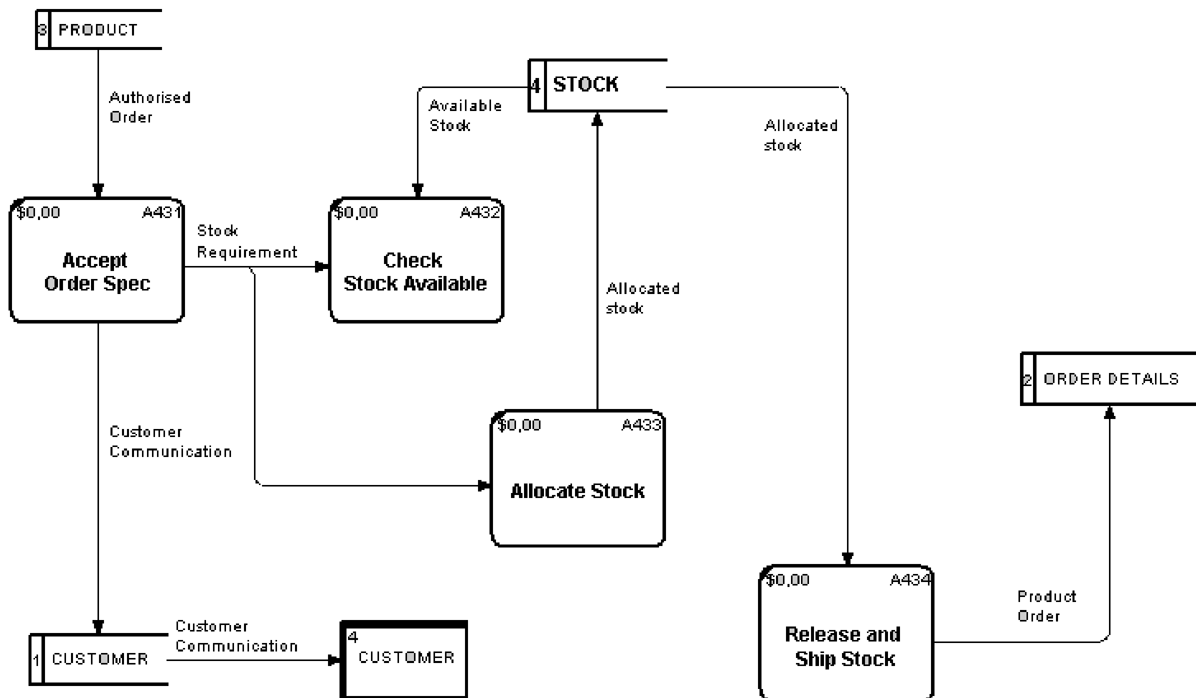

Рисунок 42. Пример DFD-диаграммы

Определение некоторого объекта в качестве внешней сущности указывает на то, что он находится за пределами границ анализируемой информационной системы. Процессы представляют собой преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом.

Номер процесса служит для его идентификации. В поле имени вводится наименование процесса в виде предложения с глаголом в неопределенной форме (вычислить, рассчитать, проверить, определить,

создать, получить) и поясняющими существительными, например: «Напечатать адрес получателя», «Акцептовать счет». Информация в нижнем поле символа процесса указывает, какое подразделение организации, сотрудник, программа или аппаратное устройство выполняет данный процесс. Накопители данных предназначены для изображения неких абстрактных устройств для хранения информации, которую можно туда в любой момент времени поместить или извлечь, безотносительно к их конкретной физической реализации. Накопители данных являются неким прообразом базы данных информационной системы организации. Внутри символа указывается его уникальное в рамках данной модели имя, наиболее точно, с точки зрения аналитика, отражающее информационную сущность содержимого, например, «Поставщики», «Заказчики», «Счета-фактуры», «Накладные». Символы накопителей данных в качестве дополнительных элементов идентификации могут содержать порядковые номера.

Поток данных определяет информацию, передаваемую через некоторое соединение (кабель, почтовая связь, курьер) от источника к приемнику. На DFD диаграммах потоки данных изображаются линиями со стрелками, показывающими их направление. Каждому потоку данных присваивается имя, отражающее его содержание.

Диаграммы потоков данных строятся по иерархическому принципу. Контекстная диаграмма верхнего уровня определяет границы модели. Как правило, она имеет звездообразную топологию, в центре которой находится главный процесс, соединенный с приемниками и источниками информации, являющимися внешним окружением моделируемой информационной системы. На первом уровне иерархии показываются основные внутренние процессы системы и соответствующие им внешние сущности, накопители и потоки данных. Для каждого процесса диаграммы первого уровня может быть произведена декомпозиция, которая, в свою очередь, также может быть раскрыта более подробно. Декомпозиция процессов заканчивается, когда достигнута требуемая степень детализации или отображаемые на очередном уровне диаграмм процессы являются элементарными и не могут быть разделены.

DFD-диаграммы поддерживаются различными инструментальными средствами, среди наиболее распространенных можно отметить BPwin. Можно отметить среди преимуществ DFD легкую и естественную интеграцию с ER-диаграммами «сущность-связь» [9]. В частности, в CASE-системе Power Designer компании Sybase в модуле для построения ER-модели Data Analyst исходные данные для модели "сущность-связь" могут быть получены из DFD-моделей, созданных в модуле Process Analyst.

Также известны некоторые алгоритмы автоматического преобразования иерархии DFD в структурные карты, и есть CASE-средства это выполняющие, на основе же структурных карт становится возможной генерация логической структуры программы.

В целом DFD-диаграммы, как и ER-диаграммы, хорошо подходят для применения при создании информационных систем – СУБД, хранилищ

данных, и т.д., и не соответствуют по выразительным возможностям, т.к. в них фактически отсутствует последовательность (тем более с привязкой ко времени) выполняемых действий.

LabVIEW

LabVIEW (сокращение английского Laboratory Virtual Instrumentation Engineering Workbench) [41] представляет собой мощную среду моделирования эксперимента со встроенным графическим языком G, описывающим потоки данных между функциональными преобразователями («приборами»). Разработчиком LabVIEW является американская корпорация National Instruments. Первая версия LabVIEW была выпущена в 1986 году для Apple Macintosh, в настоящее время существуют версии для UNIX, GNU/Linux, Mac OS и пр., а наиболее развитыми и популярными являются версии для Microsoft Windows. LabVIEW используется в системах сбора и обработки данных, а также для управления техническими объектами и технологическими процессами. Идеологически LabVIEW очень близка к SCADA-системам.

Графический язык программирования «G», используемый в LabVIEW, основан на архитектуре потоков данных. Последовательность выполнения операторов в таких языках определяется не порядком их следования (как в императивных языках программирования), а наличием данных на входах этих операторов. Операторы, не связанные по данным, выполняются параллельно в произвольном порядке.

Программа LabVIEW называется и является виртуальным прибором (англ. Virtual Instrument) и состоит из двух частей:

1. блочной диаграммы, описывающей логику работы виртуального прибора;
2. лицевой панели, описывающей внешний интерфейс виртуального прибора.

На рисунке 43 представлен пример блочной диаграммы в LabVIEW на языке G.

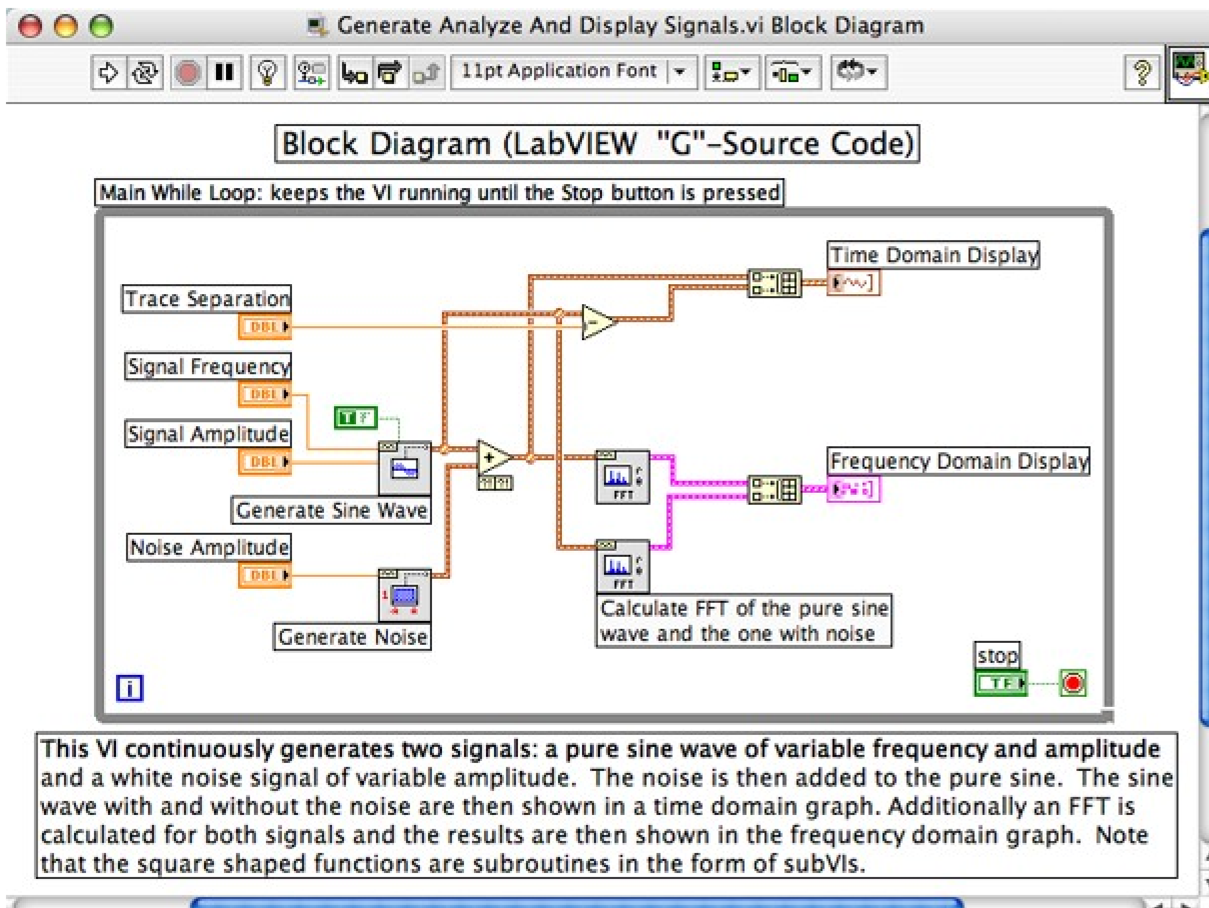

Рисунок 43. Пример блочной диаграммы графического языка G в LabVIEW

Виртуальные приборы могут использоваться в качестве составных частей для построения других виртуальных приборов. Лицевая панель виртуального прибора содержит средства ввода-вывода, такие как кнопки, переключатели, светодиоды, верньеры, шкалы, информационные табло и т. п. Они используются человеком для управления виртуальным прибором, а также другими виртуальными приборами для обмена данными.

Блочная диаграмма содержит функциональные узлы, являющиеся источниками, преемниками и средствами обработки данных. Также компонентами блочной диаграммы являются терминалы («задние контакты» объектов лицевой панели) и управляющие структуры (являющиеся аналогами элементов текстовых языков программирования, таких как условный оператор «IF», операторы цикла «FOR» и «WHILE» и т. п.). Функциональные узлы и терминалы объединены в единую схему линиями связей.

LabVIEW поддерживает огромный спектр оборудования различных производителей и имеет в своём составе (либо позволяет добавлять к базовому пакету) многочисленные библиотеки компонентов:

- для подключения внешнего оборудования по наиболее распространённым интерфейсам и протоколам (RS-232, GPIB 488, TCP/IP и пр.);
- для удалённого управления ходом эксперимента;
- для управления роботами и системами машинного зрения;

- для генерации и цифровой обработки сигналов;
- для применения разнообразных математических методов обработки данных;
- для визуализации данных и результатов их обработки (включая 3D-модели);
- для моделирования сложных систем;
- для хранения информации в базах данных и генерации отчетов;
- для взаимодействия с другими приложениями в рамках концепции COM/DCOM/OLE и пр.

Вместе с тем LabVIEW — достаточно простая и интуитивно понятная система. Неискушённый пользователь, не являясь программистом, за сравнительно короткое время (от нескольких минут до нескольких часов) способен создать программу для сбора данных и управления объектами, обладающую удобным человеко-машинным интерфейсом. Например, средствами LabVIEW можно быстро превратить старый компьютер, снабжённый звуковой картой, в мощную измерительную лабораторию.

Специальный компонент LabVIEW — Application Builder, позволяет выполнять LabVIEW-программы на тех компьютерах, на которых не установлена полная среда разработки.

Simulink

Система Simulink [42] в значительной степени похожа на систему LabVIEW. Отличительной особенностью является тесная интеграция, предполагающая совместное использование, с системой моделирования MATLAB.

Simulink – это интерактивный инструмент для моделирования, имитации и анализа динамических систем. Он дает возможность строить графические блок-диаграммы, имитировать динамические системы, исследовать работоспособность систем и совершенствовать проекты. Simulink полностью интегрирован с MATLAB, обеспечивая немедленным доступом к широкому спектру инструментов анализа и проектирования. Simulink также интегрируется с Stateflow для моделирования поведения, вызванного событиями. Эти преимущества делают Simulink популярным инструментом для проектирования систем управления и коммуникации, цифровой обработки и других приложений моделирования.

Simulink - графическая система настроенная на использование “мыши”. Она позволяет конструировать моделируемую систему «перетаскиванием» блоков на экране в рабочую область и последующей установкой их параметров. Simulink может работать с линейными, нелинейными, непрерывными, дискретными, многомерными системами.

В Simulink имеется набор библиотек элементов, в том числе:

- Sources - Источники сигналов
- Sinks – Средства отображения
- Discrete – Дискретные элементы
- Linear - Линейные элементы
- Nonlinear – Нелинейные элементы
- Connections – Соединители
- Дополнительные блоки

На рисунке 44 приводится пример графической диаграммы Simulink.

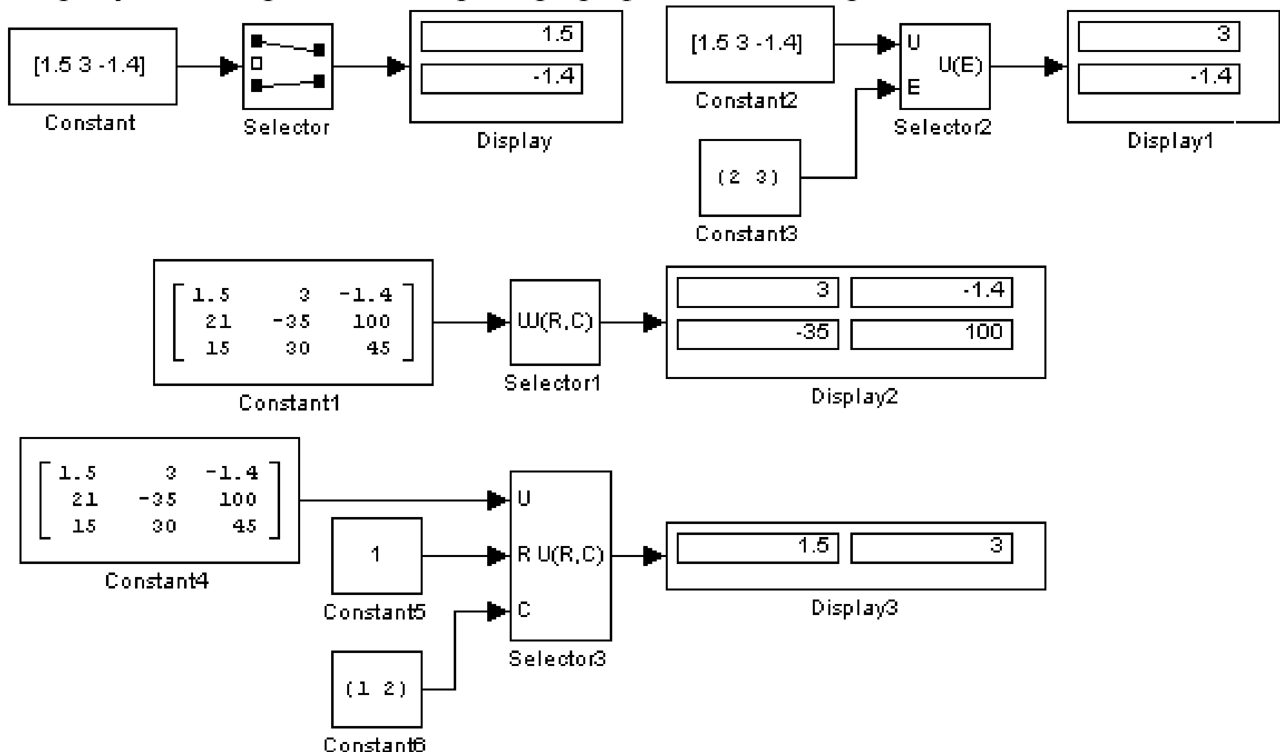

Рисунок 44. Пример диаграммы Simulink

Библиотека соединителей, например, содержит следующие узлы: ввод/вывод в порт, мультиплексор, демультиплексор, пересылка, прием, изменение и запись получаемой информации, передача, хранение данных, построение подсистем, задание качества подсистем, переключатель входов, ограничитель уровня сигнала, блок задания сигнала. Дополнительные блоки включают такие, как Additional Sinks – дополнительные средства визуализации сигналов при помощи которых можно исследовать спектральный состав сигнала или построить график функции корреляции двух сигналов; Additional Discrete – здесь содержатся допонительные блоки для исследования дискретных систем. Эти блоки являются аналогами стандартных блоков с тем исключением, что для каждого из этих блоков можно задать начальные условия; Additional Linear – содержащиеся здесь блоки являются аналогами стандартных блоков с тем исключением, что для каждого из этих блоков можно задать начальные условия; Transformations – в эту группу входят блоки, производящие перевод из одной системы координат в другую, а также блоки преобразований температур в различных исчислениях; Flip Flops – содержащиеся здесь блоки имитируют работу

различных триггеров (RS, D, JK); Linearization – содержащийся здесь блок выполняет либо взятие производной входного сигнала либо его линеаризацию.

Непосредственным аналогом диаграммы Simulink (как и, например, языка LD) можно считать принципиальную электрическую схему прибора. Интеграция с MATLAB дает возможность использования в качестве конструктивного блока функцию, реализованную в этой среде. Достоинством Simulink является также то, что он позволяет пополнять библиотеки блоков с помощью подпрограмм написанных как на языке MATLAB, так и на языках С + +, Fortran и Ada

Prograph

Язык Prograph – один из наиболее ранних по времени возникновения из рассмотренных в настоящей работе. Он является визуальным языком, ориентированным на описание потоков данных. Prograph использует пиктограммы («иконы») для представления операций над данными. Такие среды программирования, ориентированные на Prograph, как Prograph Classic и Prograph CPX для Windows и MacOS, были доступны на рынке на протяжении многих лет. Из современных систем, использующих Prograph, можно отметить интегрированную среду разработки Marten для Mac OS X.

Исследования, связанные с языком Prograph (сокращение от Programming in Graphics), начались в университете Акадии в 1982 году, предпосылками чего стали дискуссии в области функциональных языков программирования, ориентированных на обработку потоков данных. В этих дискуссиях в качестве вспомогательного иллюстративного материала активно использовались диаграммы. Тот факт, что диаграммы более удобны для восприятия человеком, подтолкнул исследователей к работам в направлении создания «исполняемых» диаграмм. Так началась история Prograph – визуального языка потоков данных. Руководителем работ был доктор Томас Петржиковский, соавторами – Стэн Матвин и Томал Малднер. Основой ранних версий был язык Паскаль (затем использовались и другие языки, в частности, Пролог).

В настоящее время права на Prograph принадлежат компании Pictorius.

Предпосылками для начала работ по Prograph послужил тот факт, что к 1970м годам в программировании стал очевидным кризис, связанный с попытками создания сложных программных систем в рамках крупных проектов. В подобных проектах даже незначительные изменения порождали серьезные и трудные для понимания побочные эффекты. В качестве рецепта решения проблемы некоторыми учеными было предложено отказаться от характера традиционных систем программирования, ориентированных на

описание логики программы, в пользу большего упора на описание способов манипуляций данными. Результатами таких попыток стали объектно-ориентированное программирование и программирование, ориентированное на потоки данных (dataflow programming). Prograph идет дальше в развитии данной концепции, вводя его комбинацию с полностью визуальной средой программирования. Объекты в ней представляются в виде шестиугольников с двумя выделенными сторонами, одна из которых представляет поля данных, а другая – методы, применяемые для их обработки (см. рисунок 45).

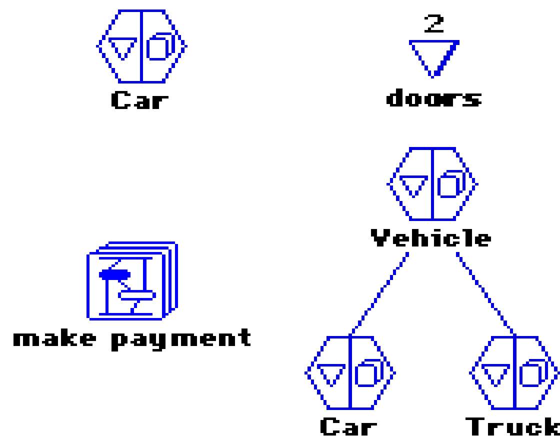

Рисунок 45. Графическое представление объектов в Prograph

Двойной щелчок мышью на любой из сторон открывает окно, более детально представляющее особенности объекта. Двойной щелчок на стороне, сопоставленной методам обработки данных, открывает реализованные и унаследованные методы. При щелчке мышью в центре пиктограммы объекта открывается окно, показывающее логику. В Prograph каждый метод представляется набором пиктограмм, каждая иконка при этом содержит инструкции. Потоки данных при этом представляются ориентированным графом. При этом используется вертикальная ориентация диаграммы – входящие потоки данных появляются сверху, результирующие (в случае их наличия) исходят вниз. На рисунке 46 представлен пример визуальной программы на Prograph, реализующей сортировку в базе данных. Заштрихованная полоска вверху обозначает единственный входной аргумент – объект базы данных, обрабатываемый затем несколькими путями, символизируемыми линиями.

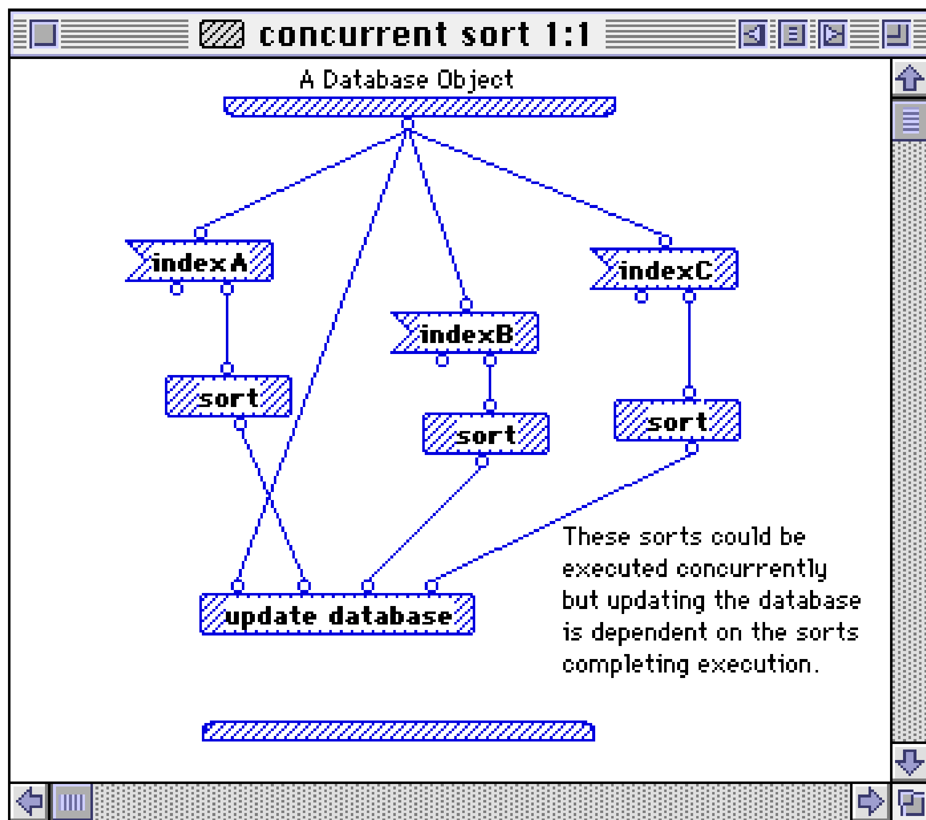

Рисунок 46. Графическая программа на Prograph

В языках потоков данных операция может быть произведена, как только для нее готовы входные данные. Это означает, в частности, что каждая из операций может осуществляться параллельно, в один и тот же промежуток времени. В приведенном примере все три операции формирования индексов могут выполняться параллельно (в случае, если ЭВМ поддерживает такие возможности). Таким образом, программа на Prograph (как и на других языках, ориентированных на потоки данных), является «по своей природе параллельной», т.е. поддерживающей многопроцессорные системы.

Ветвления и циклы в такой программе вводятся путем модификации операция с помощью аннотаций. Например, цикл, вызывающий метод `doit` для списка входных данных, формируется путем перетаскивания иконы метода и последующего присоединения к нему циклического модификатора. Еще одной интересной возможностью, поддерживаемой Prograph, является то, что разрешено подавать сам метод в качестве входных данных, что делает Prograph в некоторой степени динамическим языком.

Интегрированная среда визуальной разработки Prograph поддерживает также и визуальную отладку. Поддерживаются традиционные точки останова и пошаговый режим исполнения программы. Возможен контроль текущего состояния данных и динамическое отражение стека времени исполнения. Важнейшей при этом является возможность изменения визуальной программы «на лету», без остановки процесса отладки. Это реализует без преувеличения уникальный подход к отладке, когда ошибки могут устраняться без необходимости перекомпиляции программы.

Из числа приложений, написанных на Prograph, можно отметить электронную таблицу Spreadsheet 2000 с весьма необычными возможностями.

Ptolemy

Проект «Птолемей» (Ptolemy) [44] начал развиваться в Университете Калифорнии в Беркли в 1986 году и сплотил вокруг себя неформальную группу единомышленников. Текущая версия системы носит название «Птолемей II» (Ptolemy II), и представляет собой уже третье поколение инструментальной системы для проектирования ПО.

Первое поколение инструментальной системы было создано в период 1986-1991 гг. под названием Gabriel [45]. Она была написана на языке программирования Лисп и ориентирована (отметим, как и LabView и Simulink), на обработку сигналов. Основной использованной методикой моделирования была технология синхронных потоков данных (SDF - synchronous dataflow). Использовалась как последовательная, так и параллельная техника синхронизации. Gabriel включал в себя генератор кода на ассемблере для сигнальных процессоров DSP (например, семейства Motorola).

Классический «Птолемей» начал создаваться профессорами Эдвардом Ли и Дейвом Мессешрмиттом в 1990 г. на языке программирования С++. Он был первой средой моделирования, поддерживавшей на системном уровне множество моделей вычислений, комбинируемых иерархическим путем. Помимо SDF, в нем были реализованы двоичные потоки данных (BDF – boolean dataflow), динамические потоки данных (DDF – dynamic dataflow), многомерные синхронные потоки данных (MDSDF - multidimensional synchronous dataflow), и предметная область сетей процесов and (PN - process networks). Авторами из Gabriel был перенесен модуль генерации кода для сигнальных процессоров, помимо которого были реализованы генераторы кода на языках Си и VHDL [44]. Была поддержана возможность конструирования и выполнения модели на присоединенном к инструментальному компьютеру сигнальном процессоре. Добавили возможность построения моделей с дискретными событиями.

В систему были также введены т.н. «высокоуровневые компоненты», значительно повысившие выразительность визуального синтаксиса языка Ptolemy.

Текущая версия Ptolemy II начала реализовываться с 1996 года. Основным побуждением был перенос среды на использование в качестве базового языка программирования Java. Помимо этого, в Ptolemy II реализован полиморфизм компонент относительно предметных областей (когда компонент может быть спроектирован для использования в различных предметных областях). Добавлен так называемый модальный режим, при котором конечные автоматы комбинируются с иными моделями вычислений. Ptolemy II имеет сложную систему встроенных типов, включающую наследование и полиморфизм данных (когда компоненты проектируются для работы с различными типами данных). Была, кроме того, введена гетерохронная модель потоков данных. Визуальной оболочкой-конструктором Ptolemy стала среда Vergil, сама построенная с применением Ptolemy (набор средств Diva для построения интерфейса пользователя). Была добавлена возможность создания моделей, которые могут использоваться, как Java-апплеты в web-браузерах. Введена возможность моделирования беспроводных мобильных систем. Была также встроена поддержка ряда экспериментальных предметных областей, таких, как системы реального времени и распределенные вычисления (DDE - distributed discrete events), синхронизированная мультизадачность (TM - timed multitasking), и др.

В отличие от Gabriel и классического Ptolemy, в Ptolemy 2 применена иная техника кодогенерации, причем реализованы два подхода. В рамках первого подхода взамен компоненты-генератора кодов, генератор кода «Коперник» (Copernicus) использует специальную среду – надстройку над Java-компилятором. Второй подход применяет компоненты генерации кода, но использует архитектуру, в которой интерфейсы компонент строятся как типовые акторы Ptolemy, но использованы специальные «помощники» для генерации кодов для конкретной платформы.

Главная область применения Ptolemy, как подчеркивают авторы [44] – создание ПО для встроенных систем (embedded software).

Исполняемые в Ptolemy модели систем строятся над моделью вычислений. Модель вычислений может быть рассмотрена, как набор «законов физики», управляющих взаимодействием компонент модели. В случае, если модель описывает механическую систему, модель вычислений может быть действительно основанной на физических законах. Однако, наиболее часто это – просто набор правил, в рамках которых проектировщик строит модель. Правила взаимодействия компонент модели также называют семантикой модели вычислений. Модель вычислений при этом может иметь более чем одну семантику, в которых могут быть различные наборы правил, тем не менее, налагающих идентичные ограничения на поведение. Выбор модели вычислений определяется предметной областью, в которой предполагается использование модели. Например, для «чисто вычислительной системы», когда некоторый конечный набор исходных данных должен быть преобразован в конечный набор выходных данных, адекватной будет императивная семантика традиционных языков

программирования, таких, как C, C++, Java или MATLAB. При моделировании механических систем семантика должна позволять работать с параллельно идущими процессами и непрерывным временем, в этом случае более подходящей будет модель вычислений, такая, как в Simulink, Saber, Hewlett-Packard's ADS или VHDL-AMS [44]. Некоторые модели вычислений, например, могут быть реализованы настраиваемыми аппаратными средствами, в то время как другие плохо соответствуют такому подходу в силу их изначально последовательной природы. Выбор неподходящей модели вычислений может поставить под угрозу качество проекта, приведя разработчика к более дорогостоящей и/или менее надежной реализации. Целью создания Ptolemy II была поддержка возможности построения и взаимодействия моделей, принадлежащих к широкому спектру различных моделей вычислений.

Почти все модели вычислений в Ptolemy II поддерживают так называемое «актор-ориентированное проектирование». Это контрастирует с объектно-ориентированным подходом и дополняет его, подчеркивая параллелизм и взаимодействие между компонентами. Компоненты, называемые *акторами*, исполняются и взаимодействуют с другими акторами. Как и объекты, акторы имеют четко определенный интерфейс. Этот интерфейс оставляет за скобками внутреннее состояние и особенности поведения актора, описывая лишь, как он взаимодействует с внешней средой. Интерфейс включает так называемые *порты*, представляющие собой точки взаимодействия актора, и параметры, используемые для настройки выполняемой актором операции. Часто (но не всегда) значения параметров задаются априори и не меняются в процессе выполнения модели. Центральным моментом в актор-ориентированном проектировании являются каналы коммуникации, которые передают данные от одного порта к другому в соответствии со схемой передачи сообщений. В отличие от объектно-ориентированного программирования, когда объекты, как правило, взаимодействуют, передавая управление с помощью вызова методов, при актор-ориентированном подходе они взаимодействуют путем посылки сообщений по каналам коммуникации. Это подразумевает, что акторы связываются с каналами, но не напрямую с другими акторами. Пример визуального описания модели в Ptolemy II представлен в среде Vergil на рисунке 47.

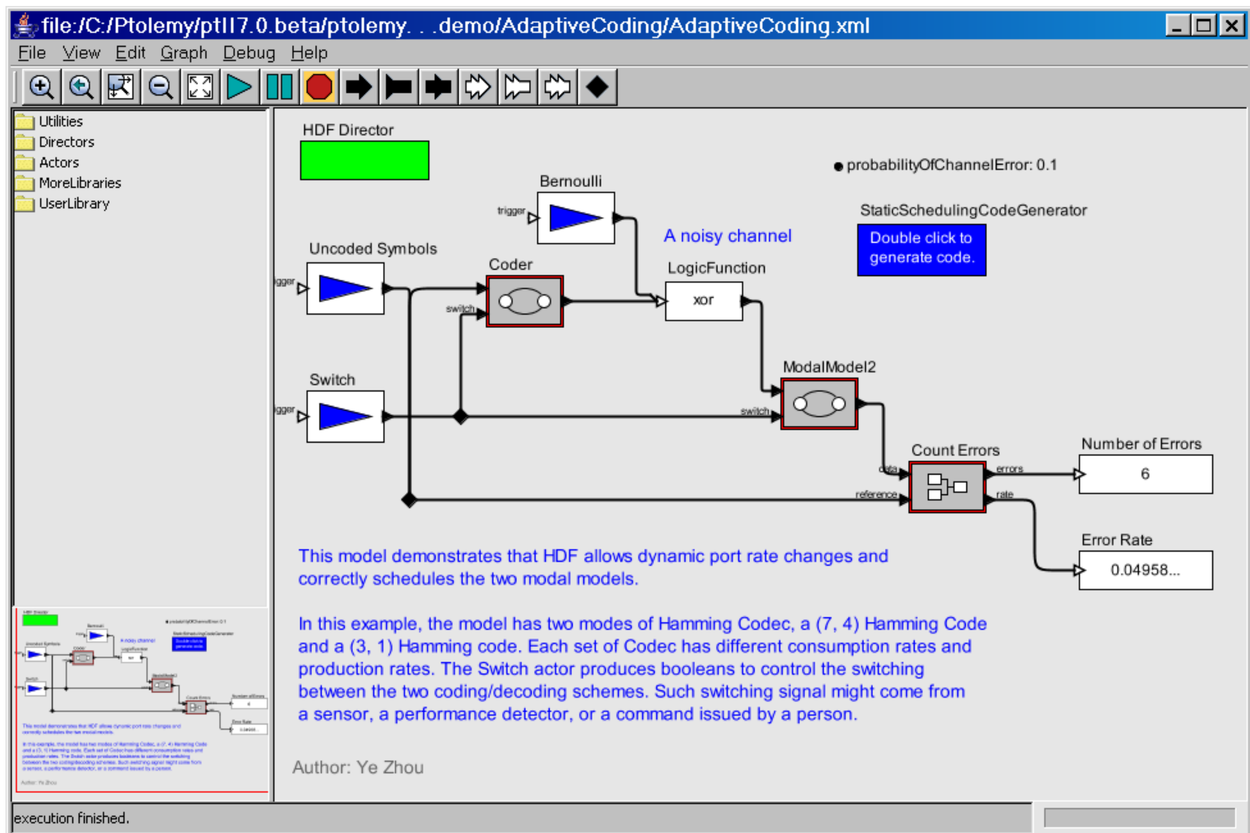

Рисунок 47. Пример визуальной модели Ptolemy

Подобно акторам, модель в целом может иметь внешний интерфейс. Этот интерфейс называется иерархической абстракцией, описывает он внешние порты и внешние параметры. На этой основе модели могут связываться друг с другом. При этом внешние параметры модели могут использоваться для задания внутренних параметров акторов, задействованных в модели.

Как заявляют авторы [44], в отличие от таких языков описания архитектуры ПО, как Wright [46] и Rapide [47], Ptolemy II может быть назван языком проектирования архитектуры и обладает более богатыми возможностями благодаря тому, что он ориентируется не на отдельных компонентах и их связях, а на сборках компонент и полиморфен относительно предметных областей.

Несмотря на то, что в целом Ptolemy может рассматриваться как разновидность систем описания потоков данных, в нем присутствует реализация кардинально отличной предметной области – FSM (Finite State Machines, конечные автоматы). Сущности в этой области представляют не акторов, а состояния, а коннекторы – переходы между состояниями.

Поскольку Ptolemy изначально ориентировался на встраиваемые приложения, он не мог обойти стороной проблематику систем реального времени. В нем имеются реализации нескольких предметных областей, так или иначе связанных с системами реального времени. Рассмотрим их подробнее.

Discrete-Events – DE

В предметной области дискретных событий (discrete-events) DE акторы взаимодействуют с помощью последовательности событий, происходящих в те или иные моменты на временной оси. Каждое событие характеризуется значением и временем. Акторы могут быть либо процессами, реагирующими на события (реализуемыми в виде нитей Java), либо функциями, вызываемыми при поступлении нового события. Ориентируется данная предметная область Ptolemy на применение в области телекоммуникаций и используемые в ней языки VHDL и Verilog.

Это, а также принципиально асинхронный характер (события происходят в произвольные моменты времени) данной модели не позволяет ее использовать для создания БПО КА ДЗЗ.

Giotto

Предметная область Giotto, реализованная Кристофером Мейром Кришем, поддерживает специфическую модель вычислений, разработанную Томом Хензингером, Кристофом Киршем, Беном Горовицем и Хаяном Ченом. Она управляется временем в том смысле, что каждый актор вызывается циклически с установленным периодом. Эта предметная область специально предназначена для работы совместно с предметной областью FSM для реализации модальных моделей. Она нужна для систем «жёсткого»реального времени с предварительным распределением ресурсов. К сожалению, модель конечных автоматов FSM все-таки не является моделью, отражающей поток управления, равно как и периодический вызов программ не соответствует специфике УА РВ, имеющих начало и конец, что не позволяет использовать и эту, наиболее подходящую к специфике КА модель предметной области Ptolemy, в разработке БПО.

Timed Multitasking – TM

Предметная область TM в Ptolemy II разработана Джи Лю и поддерживает разработку выполняемого параллельно ПО реального времени. Это предполагает функционирование программ под управлением диспетчера задач реального времени с приоритетами, типичного для ОС РВ. В TM каждый актор концептуально рассматривается, как параллельная задача под управлением ОС РВ. Имеется механизм представления

однородного и монотонного модельного времени. Каждый актор имеет назначенное время исполнения T, но он задерживает свое выполнение до тех пор, пока не будет иметь доступ к процессору на протяжении требуемого временного отрезка. Однако, при этом начало выполнения актора определяется поступлением на его вход новых исходных данных, т.е. процесс исполнения управляется событиями. В итоге концептуально исполнение актора начинается в некоторый момент времени t, а выходные данные получаются в момент времени t+T+P, где T – время исполнения актора, а P – время ожидания, пока актор стоит в очереди на выполнение в соответствие со своим приоритетом. К сожалению, поскольку и тут момент начала исполнения действия определяется асинхронно по событию поступления исходных данных, эта модель не может быть признана адекватной проблематике проектирования УА РВ для КА ДЗЗ.

Событийный подход

Графическое программирование в IBM Visual Age

Несмотря на то, что частично объектно-ориентированный и событийный подходы были рассмотрены выше в предшествующих главах, наиболее ярким и характерным представителем графического языка, сочетающего оба из них, можно признать язык, реализованный в среде разработки программ IBM Visual Age [48]. Она ориентирована, прежде всего, на разработку бизнес-приложений, включая системы для онлайновой обработки транзакций и системы поддержки решений. VisualAge позволяет профессиональным разработчикам строить клиентские части прикладных систем со сложным графическим интерфейсом, проектировать деловую логику работы приложений с доступом к локальным и удаленным ресурсам. VisualAge представляет собой чисто обьектно-ориентированное средство разработки (изначально реализованным языком является Smalltalk), включающее набор визуальных интерактивных инструментов, библиотеку готовых компонент и набор средств для построения клиент-серверных приложений. Библиотека готовых компонент поддерживает устройства мультимедиа, коммуникации через протоколы APPC, TCP/IP, NetBIOS, программные интерфейсы CICS External Call Interface, EHLLAPI, Message Queue Interface (MQI), работу с реляционными базами данных семейств DB2, Oracle, Sybase, и многое другое.

Прежде чем перейти к описанию визуального построения приложений, можно отметить ряд замечательных возможностей, предоставляемых VisualAge, таких, как повторное использование кода, поддержка моделей SOM и DSOM, возможности групповой разработки приложений с использованием центрального репозитория.

Практически все визуальные средства нуждаются в дополнении функциями, которые не могут быть представлены в виде графических конструкций и требуют текстового выражения. Визуальные средства дополняются текстами, написанными на тои или ином языке программирования.

VisualAge поддерживает ряд языков, существуют версии для Java, C++, Си, Бейсика, Фортрана и других языков, однако базовым является «чистый» объектно-ориентированный язык Smalltalk. Сама среда разработки VisualAge создана на Smalltalk. Среди разнообразных средств визуального программирования VisualAge интересен именно наиболее последовательно реализованной концепцией обьекто-ориентированной технологии. Приложения, написанные на Smalltalk, соответствуют схеме MVC (Model-

View-Controller). Model является набором объектов, выражающих бизнес-логику приложения, View представляет объекты из пользовательского интерфейса, Controller состоит из объектов, преобразующих действия пользователя с представителями набора View в запросы к объектам Model.

Как показано в настоящей монографии, средства визуального программирования могут использоваться по-разному. Только для визуального определения пользовательского интерфейса, который затем встраивается в традиционную программную среду. Более глубоко, для визуального конструирования и интерфейса, и контролирующих интерфейс объектов, но программированием основной логики приложения традиционными методами. Наконец, для полного визуального построения приложения.

VisualAge реализует концепцию построения приложений из готовых компонент – так называемых «деталей» (parts). Это означает, что программа создается путем соединения и по определенным правилам деталей между собой. Так же, как на заводе из радиодеталей собираются платы, а из них компьютеры, так и программные компоненты в Visual Age могут образовывать составные детали (сборочные единицы), а они, в свою очередь, готовое приложение.

В VisualAge «деталь» - это законченный программный объект с описанным внешним интерфейсом, устанавливающим свойства детали. Существует два вида деталей в VisualAge:

- *Видимая деталь* - деталь, имеющая видимое представление во время исполнения программы. Видимые детали, такие как окна, кнопки, поля ввода и тому подобное, составляют пользовательский интерфейс приложения. Они, в свою очередь, могут состоять из других видимых деталей, невидимых деталей и соединений между ними.
- *Невидимая деталь* - деталь, имеющая видимого представления во время исполнения программы. Невидимые детали обычно представляют поведение данных в приложении. Они могут состоять из других невидимых деталей и соединений.

Внешний интерфейс детали определяет то, как она может взаимодействовать с другими деталями в приложении. Он состоит из следующих свойств:

1. *Атрибуты* - данные, к которым имеют доступ другие детали. Эти данные описывают сущность и состояние детали. Например, это может быть остаток на банковском счете, имя человека, метка на кнопке ввода.
2. *События* - сигналы, посылаемые деталью для уведомления об изменении ее состояния: например об открытии окна или изменении значения какого-либо атрибута.
3. *Действия* - операции или функции, которые деталь может выполнять. Действия могут быть инициированы другими деталями посредством соответствующих связей.

Связи определяют то, каким образом детали взаимодействуют друг с другом. Можно установить связи между деталями, встроенными программами(scripts) и другими связями. Введено четыре типа связей:

- *Атрибут - Атрибут* соединяет два значения данных таким образом, что при изменении одного из них так же изменяется и другое.
- *Событие - Действие* вызывает выполнение действия при возникновении события.
- *Событие - Встроенная программа* запускает выполнение встроенной программы при возникновении события.
- *Атрибут - Встроенная программа* запускает выполнение встроенной программы тогда, когда значение атрибута должно быть вычислено.

Такой подход к деталям в системе VisualAge позволяет реализовать концепцию визуального объектно-ориентированного программирования. Объектно-ориентированного - потому что детали являются полноценными программными объектами со свойствами инкапсуляции данных, наследования и полиморфизма. На рисунке 48 представлен пример простого приложения с графическим интерфейсом.

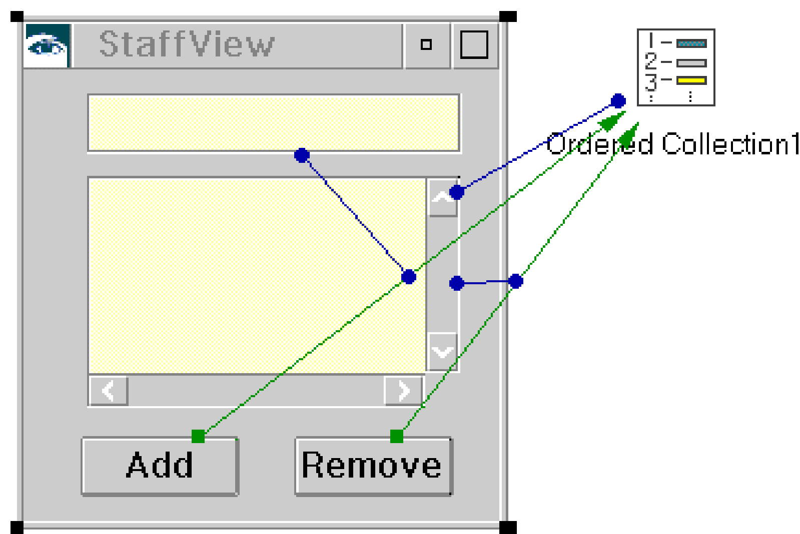

Рисунок 48. Пример приложения в графической среде Visual Age

Очевидно, что для создания такого приложения необходимы следующие детали: поле ввода, список, две кнопки и что-нибудь для запоминания списка. Необходимые детали могут быть размещены на форме приложения путем выбора из предоставляемой Visual Age палитры деталей. Для запоминания списка может использоваться невидимая (не отображаемая в пользовательском интерфейсе) деталь Ordered Collection, которая позволяет добавлять и удалять объекты любого типа.

Логику программы можно описать следующим предложением: "При нажатии кнопки Add (событие) необходимо добавить содержимое поля ввода в Ordered Collection, и отобразить находящееся в ней элементы в Списке; при нажатии кнопки Remove удалить выделенный элемент Списка из Ordered Collection". В соответствии с этим установлены связи между деталями. При

нажатии кнопка генерирует событие 'clicked', связываемое путем визуального конструирования (рисования линии) с действием 'add' (добавить), выполняемым объектом Ordered Collection. Штриховая линия говорит о том, что параметр для вызываемого действия не указан (в случае необходимости Visual Age позволяет добавлять параметры). Так как в коллекцию добавляется содержимое поля ввода, атрибут 'object' поля ввода связан с атрибутом 'anObject' связи Кнопка- Ordered Collection.
Для того, чтобы содержимое Ordered Collection отображалось в Списке, атрибуты 'items' Списка и 'self' Коллекции соединены. Аналогичным образом построены связи кнопки "Remove", только вызываемым действием Ordered Collection будет 'remove', а его параметром - атрибут 'selected item' Списка.

Таким образом в Visual Age возможно построение приложений, без необходимости написания ни единой строки программы, а всего лишь путем визуального задания логики взаимодействия компонент с помощью набора типовых связей.

Очевидно, что успех реальной разработки напрямую зависит от качества и количества доступных деталей. Они могут быть созданы как визуальным построением, программированием на языке Smalltalk, так и наследованием от существующих систем. Например, если в рассмотренном примере список сотрудников необходимо хранить в БД, задача несколько усложняется, но подход к созданию приложения не изменится. Достаточно использовать другую деталь вместо Ordered Collection - 'Запрос к базе данных'. Она обладает более широким набором свойств, среди которых будут использоваться действие 'выполнить запрос' ('executeQuery') и атрибут 'результирующая таблица' ('resultTable'). Действие выполняет заданное SQL выражение, а деталь 'результирующая таблица' позволяет работать с полученным результатом. Визуальная программа при этом меняется несущественно (см. рисунок 49).

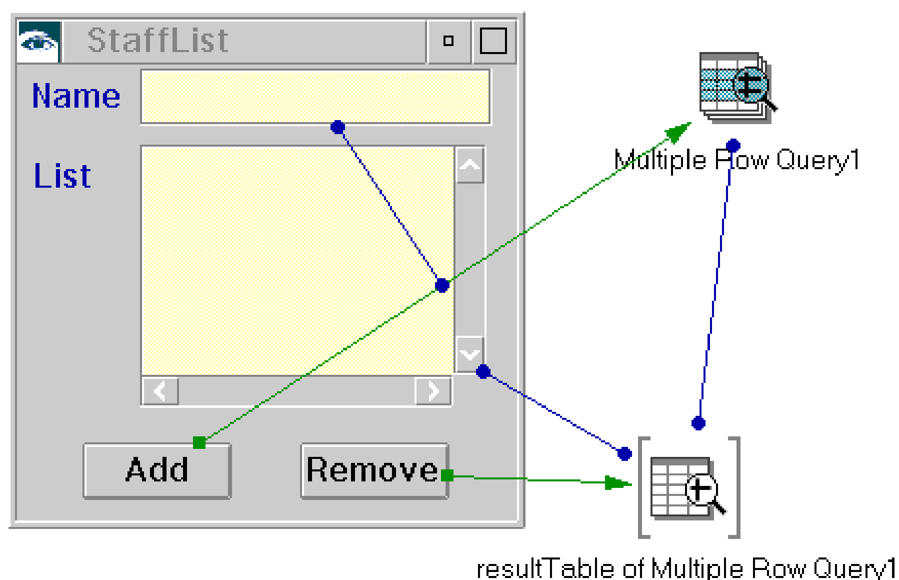

Рисунок 49. Пример приложения, работающего с СУБД, в среде Visual Age

Как и в случае других визуальных систем программирования, в результате применения концепции графического программирования в Visual Age происходит переход к новому стилю ведения проектов по разработке ПО. Становится возможным более простое и производительное взаимодействие программистов и пользователей (ранее не всегда способных понять друг друга), то есть коллектив разработчиков может состоять из «фабрикантов» - создателей «деталей» и «сборщиков» - специалистов в предметной области, которые, оперируя более близкими им понятиями, будут способны быстро создавать приложения без необходимости изучать программирование как таковое.

Передача методов, данных и событий в системе HiAsm

Визуальный конструктор программ HiAsm [50, 51] опирается на комбинированную объектно-ориентированную компонентную идеологию. Программная система и поддерживаемый ей графический язык разработаны в РФ, немаловажным достоинством является их бесплатность. Какие именно программы, для каких платформ могут быть созданы в среде HiAsm, зависит от так называемого «пакета» (или набора пакетов), установленных в среде разработки. Имеются пакеты FPC (Delphi-ориентированный), Web, Fasm (генерация текста программы на ассемблере), PocketPC, Qt. Пакет включает в свой состав палитру элементов (компонент, из которых строится визуальная схема приложения), один или несколько типов проектов, а также функциональный модуль для генерации текста программы на том или ином языке программирования с предполагаемой последующей трансляцией для получения исполняемого приложения.

Программа HiAsm конструируется полностью визуально. Элементы программы переносятся на поле редактирования путем выбора на панели элементов с последующим щелчком на рабочем столе программы. Построение алгоритма программы осуществляется путем соединения элементов линиями (связями). При этом важной и семантически наполненной является их ориентация, линии связей могут идти исключительно от правой стороны одного элемента к левой стороне другого или от нижней стороны одного к верхней стороне другого. В отдельных случаях возможно замкнуть элемент связями сам на себя. Некоторые элементы могут быть составными, т.е. состоящими из нескольких других, или иметь изменяемое количество «точек входа» («портов») для связей. В одну «точку» элемента HisAsm может вести только одна линия, поэтому для объединения и ветвления связей используются специальные элементы.

Элементы отображаются в виде прямоугольников. Каждый элемент может иметь точки входа для событий (справа), методов (слева), данных (сверху), и свойств (снизу компонента). Не все компоненты имеют все разновидности точек входа. Так, у компонент «Кнопка» (Button) имеет только событие и метод, «Надпись» - метод и свойство, более сложный компонент «Медиа» имеет все разновидности точек входа. В среде имеется достаточно широкий выбор встроенных компонент, реализующих как элементы графического пользовательского интерфейса, так и математические операции (включая универсальный парсер алгебраических выражений с возможностью использования библиотеки математических функций), управление внешними устройствами, структуры и базы данных, Интернет-протоколы, мультимедиа-средства, и многое другое.

Линии связей фактически представляют собой изображение потоков данных, событий и вызовов методов объектов. В связи с тем, что в технологии HiAsm представляются не только потоки данных, ее можно считать комбинированной и более развитой, нежели чистые визуальные языки описания потоков данных. Основным видом потоков в HiAsm является поток типа «события-данные».

Как правило, после выполнения компонентом того или иного метода (работы) результат выдается в поток вместе с событием, которое формирует в этом случае поток «события-данные». Как правило, большинство методов компонента перед выполнением работы проверяют поток на наличие данных и если данные есть(и их формат совпадает с требуемым), эти данные используются. На рисунке 50 представлен пример графической программы в среде HiAsm.

Помимо чисто визуального программирования, HiAsm предоставляет возможность вставки программного текста на языках Java и VB или на том языке, на котором генерируется данный проект пакета, что позволяет использовать отсутствующие в стандартном наборе компонент функции, или реализовывать сложные расчеты или иную программную логику, если запись в виде текста представляется более эффективной. Можно также редактировать программные тексты существующих компонент (на языке Объектный Паскаль, поскольку базой реализации HiAsm послужила система Delphi).

Начиная с версии HiAsm 3.4, в среду введены новые средства отладки и построения схем, доступные через новое контекстное меню линии связи между двумя точками компонента. Достаточно нажать правую кнопку мыши на свободном месте линии, в этом случае вам становится доступным меню из четырех пунктов. Первый - «Разрыв» предназначен для автоматической вставки в связь компонента «Разрыв». Необходимость в этом возникает, когда в схеме присутствует достаточно много связей, идущих через весь проект.

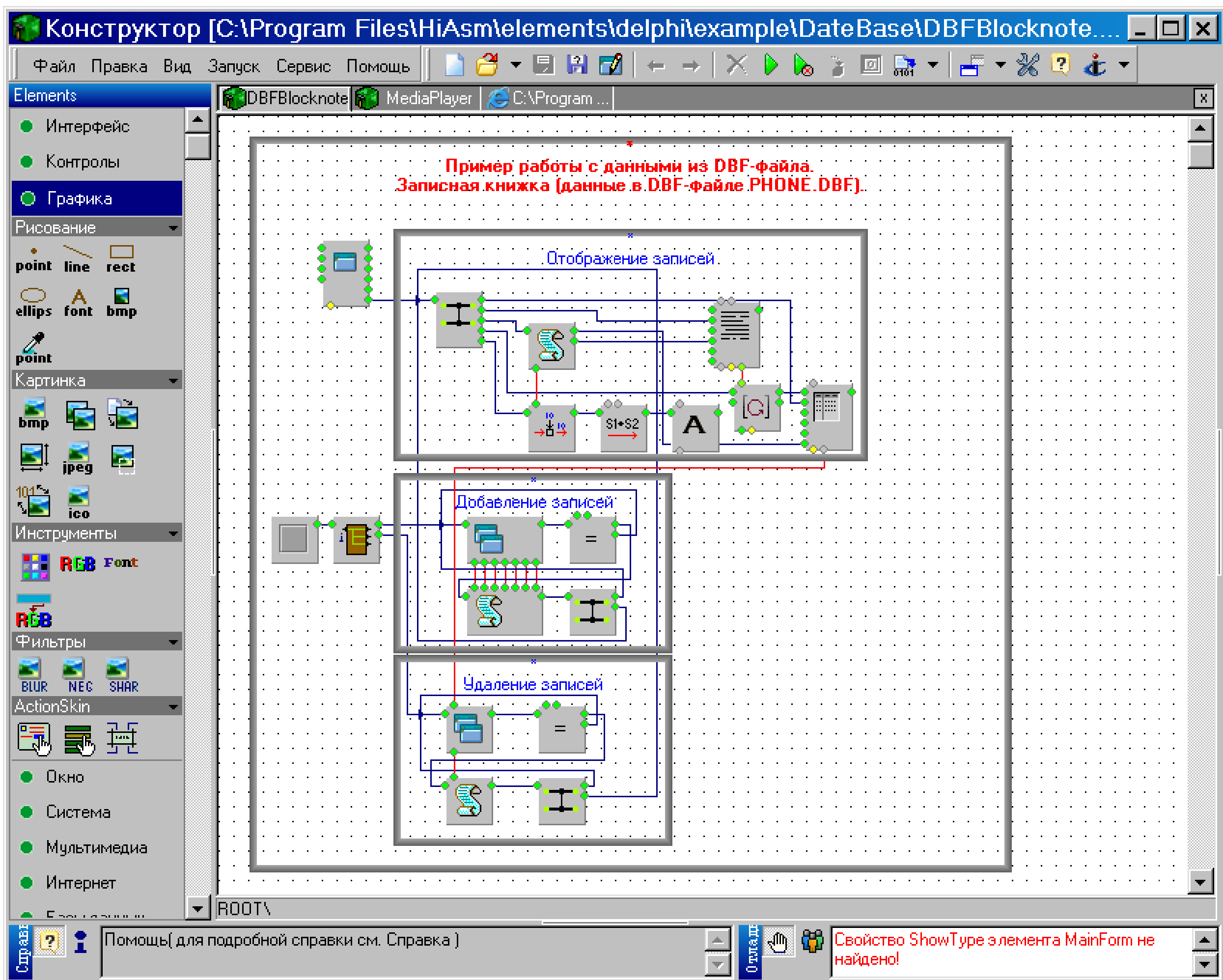

Рисунок 50. Пример экрана разработки визуального конструктора программ HiAsm

В этом случае применение компонента «Разрыв» поможет убрать лишние связи, не ухудшив качество генерируемой программы. «Точка останова» - этот компонент используется для контроля данных в потоке и для останова программы в режиме отладки. Данный компонент убирается из схемы по завершении отладки. «Доступ к данным» - простая вставка компонента DoData. «Узел» - по своему назначению аналогичен элементу Hub и предназначен для разветвления потока. При использовании «Узла» HiAsm пытается автоматически определить, к каким точкам подключить связи того или иного компонента, но возможно сделать это и вручную.

Заключение

Особенности систем управления реального времени КА и рекомендации по использованию графических языков программирования при их разработке.

Для использования на различных стадиях проектирования и разработки ПО систем управления реального времени, включая БПО КА ДЗЗ, графических языков программирования имеется ряд весомых предпосылок, среди которых:

1. Высокая трудоемкость создания ПО, необходимость повышения производительности труда разработчиков.
2. Крайне высокие требования к надежности, которые вызывают необходимость всемерного снижения вероятности внесения ошибок в программы.
3. Сложность непротиворечивой спецификации требований к УА РВ, достижения взаимопонимания между проектировщиками управляемых систем и входящих в них аппаратных средств, алгоритмистами и программистами.
4. Необходимость тщательного документирования разрабатываемых систем, адекватного, незамедлительного и точного отражения в документации всех изменений, вносимых в ПО (поддержания актуальности документации).
5. Вероятность использования различных платформ – например, БЦВМ на борту КА в настоящее время и на перспективу.

При этом существует достаточно большое количество различных визуальных сред и языков графического программирования, основанных на различных принципах и позволяющих строить описания ПО с той или иной точки зрения, что нашло свое отражение при проведении анализа в данной работе.

Если говорить, в частности, об использовании существующих средств при разработке БПО КА ДЗЗ, то это не представляется целесообразным в силу ряда причин, среди которых следующие.

Системы визуального конструирования пользовательского интерфейса неприменимы к БПО автоматических КА ДЗЗ ввиду отсутствия в нем взаимодействия с человеком-пользователем.

Больша′я часть разработок в области визуального программирования носит исследовательский или учебный характер и не может быть использована в промышленных масштабах для решения сложных и ответственных задач большой размерности, к которым относится проектирование БПО КА.

Языки графического описания структур данных практически не имеют значимой области применения при создании БПО КА в силу специфики решаемых на борту задач, не связанных с обработкой сложных структурированных данных, а скорее сводящихся к контролю логических

условий, выдаче команд на БА и запуску требуемых функциональных программ из комплекса БПО. Спецификой УА РВ является необходимость чёткой реализации логики управления БА, что требует соблюдение заданной последовательности проверок условий и выдачи необходимых в соответствии с требуемой циклограммой работы КА управляющих воздействий на бортовые системы и программы.

При этом число возможных состояний УА РВ определяется количеством сочетаний возможных состояний БА КА во все системные моменты времени и является практически необозримым. В силу этого, а также отсутствия в большом числе такого рода средств поддержки реального времени, неприменимыми при разработке БПО КА являются рассмотренные графические средства, ориентированные на состояния (SDL, диаграммы конечных автоматов UML, SFC).

Неадекватными проблематике УА РВ являются и графические языки описания потоков данных, к которым, в частности, относятся FBD и LD из стандарта МЭК 61131-3. В них в явном виде отсутствует отражение последовательности и логики выполнения отдельных операций (основным принципом активации выполнения той или иной функции в языках потоков данных является наличие на входе исходных данных).

Не применяется в силу повышенных накладных расходов и требуемых ресурсов при создании БПО КА ДЗЗ объектно-ориентированное программирование, в силу чего отпадает возможность использования объектно-ориентированных визуальных технологий, таких, как диаграммы классов UML, Visual Age или HiAsm.

Наибольший потенциальный интерес представляют графические языки, описывающие поток управления (control flow). Однако, как продемонстрировано в данной работе, они также недостаточно адекватны задачам проектирования и разработки БПО КА ДЗЗ.

Язык графических блок-схем в чистом виде обладает недостаточными выразительными средствами для отражения особенностей УА РВ для КА, в том числе – особенностей исполнения в реальном времени, и не имеет общепринятой реализации в виде настраиваемого на генерацию программ на произвольных языках инструментального средства.

Технология графосимволического программирования ГРАФ ориентирована на достаточно узкий круг приложений (Windows-приложения с базовым языком программирования Си), помимо этого, она не поддерживает приложений реального времени.

Диаграммы деятельностей и активностей UML, с одной стороны, обладают достаточными выразительными возможностями (возможно, даже несколько ими перенасыщены, что приводит к сложности для восприятия и отсутствию концептуальной целостности). С другой – распространенные CASE-средства, ориентированные на язык UML, не имеют возможности автоматической генерации программ по данным разновидностям диаграмм. В целом же использование UML при создании БПО КА ДЗЗ вообще затруднено тем, что это – объектно-ориентированная технология.

Не вполне удобным и явно нишевым средством, ориентированным на промышленные контроллеры, является язык потоковых диаграмм FC системы IsaGraf.

Система Algorithm Builder, которую можно признать достаточно удачной, также, к сожалению, ориентирована исключительно на узкий сегмент приложений – программирование микроконтроллеров Atmel.

В настоящее время отсутствуют инструментальные средства поддержки Р-схем программ Вельбицкого для приложений реального времени.

Видимо, наиболее близким, в том числе и в силу своего происхождения (разработан в НПЦ АП имени Пилюгина), к задачам создания УА РВ для КА ДЗЗ, является язык ДРАКОН и поддерживающая его технология ГРАФИТ-ФЛОКС. В ней присутствуют и средства поддержки выполнения в реальном времени. Однако, и в ней, во-первых, отсутствуют средства гибкой настройки на генерацию программ на различных ассемблерах или языках высокого уровня. Во-вторых, в принципе не поддерживается понятие входа программы, являющееся ключевым в технологии разработки БПО КА ДЗЗ. И, наконец, в ней в качестве основы применяется графическая нотация блок-схем, недостаточно выразительная, как уже было отмечено выше, для полноценного отражения особенностей УА РВ.

Таким образом, можно сделать следующие выводы.

1. При создании БПО КА ДЗЗ целесообразно применение графических языков и средств программирования, поддерживаемых интегрированными средами разработки.
2. Существующие графические языки недостаточно адекватны специфике УА РВ для КА, и решаемым на различных стадиях ЖЦ БПО проблемам.

В силу этого можно рекомендовать использование при создании БПО КА ДЗЗ графических языков и технологий, в достаточной степени учитывающих их специфику, чего можно достичь путем применения специально разрабатываемых для решения данных задач инструментальных средств.

Литература

1. Соллогуб А.В., Аншаков Г.П., Данилов В.В. Космические аппараты систем зондирования поверхности Земли. Под ред. Д.И. Козлова.-М.:Машиностроение,1993.

2. Управление космическими аппаратами зондирования Земли:Компьютерные технологии / Д.И.Козлов, Г.П. Аншаков, Я.А. Мостовой, А.В. Соллогуб.-М.:Машиностроение,1998.

3. Теоретические основы проектирования информационно-управляющих систем космических аппаратов / В.В. Кульба, Е.А. Микрин, Б.В. Павлов, В.Н. Платонов; под ред. Е.А. Микрина; Ин-т проблем упр. Им. В.А. Трапезникова РАН.-М.:Наука, 2006.

4. Авиастроение. Том 6 (Итоги науки и техники, ВИНИТИ АН СССР). М., 1978.

5. Брукс Ф. Мистический человеко-месяц или как создаются программные системы: Пер. с англ.-СПб.:Символ-Плюс, 1999.

6. Тьюринг А. Может ли машина мыслить? –М.: Физматгиз, 1950.

7. Terry Winograd, Carlos F. Flores. Understanding Computers and Cognition: A New Foundation for Design. Ablex Pub. Corp., Norwood, NJ, 1986.

8. Калянов Г.Н. CASE. Структурный системный анализ (автоматизация и применение).-М.:Лори, 1996.

9. Вендров А.М. CASE-технологии. Современные методы и средства проектирования информационных систем.-М.:Финансы и статистика, 1998.

10. Калентьев А.А., Тюгашев А.А. ИПИ/CALS технологии в жизненном цикле комплексных программ управления. – Самара: Изд-во Самарского научного центра РАН, 2006.

11. Зюбин В.Е. Графические и текстовые формы спецификации сложных управляющих алгоритмов: непримиримая оппозиция или кооперация? – Сб.трудов VII Международной конференции по электронным публикациям "EL-Pub2002", Новосибирск, 2003.

12. МЭК 65B/373/CD, Committee Draft - МЭК 61131-3. Programmable controllers. Part 3: Programming languages, 2nd Ed. // International Electrotechnic Commission. 1998.

14. Билкун С.Н., Маслюк Г.Ф. О структурном программировании // Программирование. 1976. No 5.

15. Рамбо Дж., Блаха М. UML 2.0. Объектно-ориентированное моделирование и разработка 2-е изд. - СПб: Питер, 2007.

16. Коварцев А.Н. Автоматизация разработки и тестирования программных средств. - Самар. гос. аэрокосм. ун-т., Самара, 1999. - 150 с.

17. Зубинский А. Визуальное программирование/ Компьютерное обозрение, 2005, №14(483), с.58-60.

18. Мартынюк В.В. Выделение цепей в схемах алгоритмов // ЖВМ и МФ.-1961.-Т.1,№1.-С.151-162.

19. Янов Ю.И. О логических схемах алгоритмов // Проблемы кибернетики.-М.:Физматгиз,1958.Вып.1. С.75-127.

20. Лавров С.С. Об экономии памяти в замкнутых операторных схемах // Журнал вычисл.математики и мат.физики.-1961.-Т.1.,№4.-С.687-701.

21. Котов В.Е. Введение в теорию схем программ.-Новосибирск:Наука, 1978.

22. Котов В.Е. Сети Петри.-М.:Наука, 1984.

23. ГОСТ 19.003-80. Схемы алгоритмов и программ. Обозначения условные графические

24. ГОСТ 19.701-90 (ИСО 5807-85). Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения.

25. Вельбицкий И.В., Ковалев А.Л., Лизенко С.Л. Графический интерфейс представления алгоритмов и программ // УСиМ. 1988. №4, С.42-47.

26. Паронджанов В.Д. Как улучшить работу ума. Алгоритмы без программистов. –М.:Дело, 2001.

27. J. A. Robinson : Editor's Introduction. Journal of Logic Programming Vol.1 1984.

28. Kay, Alan. "The Early History of Smalltalk", in Bergin, Jr., T.J., and R.G. Gibson. History of Programming Languages - II, ACM Press, New York NY, and Addison-Wesley Publ. Co., Reading MA 1996, pp. 511-578

29. Соловьёв А.С. Интерпретатор языка блок-схем. // Материалы научно-практической конференции “Новые информационные технологии в университетском образовании”. - Новосибирск: Издательство ИДМИ, 1999.-227с.

30. Терехов А.Н. Технология программирования.-М.:Интернет-Университет информационных технологий; БИНОМ, 2007.

31. Леоненков А.В. Самоучитель UML 2.-СПб.:БХВ-Петербург, 2007.-576 с.:ил.

32. Рамбо Дж., Блаха М. UML 2.0. Объектно-ориентированное моделирование и разработка - 2-е изд.-СПб: Питер, 2007. 544 с,ил.

33. Citrin W., Ghiasi S., Zorn B.G VIPR and the Visual Programming Challenge J. Vis. Lang. Comput. 9(2), 1998, p. 241-258

34. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб., Наука, 1998.

35. Мишель Ж. Программируемые контроллеры. Архитектура и применение. М.: Машиностроение, 1992.

36. Шопырин Д.Г., Шалыто А.А. Синхронное программирование // «Информационно-управляющие системы». 2004. № 3, с. 35-42.

37. Maraninchi F. The Argos language: Graphical representation of automata and description of reactive systems. Presented at the IEEE Workshop Visual Lang., Kobe, Japan, 1991. 7 p

38. Зюбин В. Е. Программирование информационно-управляющих систем на основе конечных автоматов: Учеб.-метод. пособие / Новосиб. гос. ун-т. Новосибирск, 2006. 96 с.

39. Robert B. France, Sudipto Ghosh, Trung Dinh-Trong, Arnor Solberg. Model-Driven Development Using UML 2.0: Promises and Pitfalls, IEEE Computer, February 2006. IEEE Computer Society, 2006.

40. Руководство разработчика программ микроконтроллера Amtel home.tula.net/algrom/russian.html

41. Вавилов К.В. LabView и Switch-технология. Методика алгоритмизации и программирования задач логического управления. СПб.: Haiboria.ru, 2005 г. - 68 с.

42. Черных И.В. Моделирование электротехнических устройств в MATLAB, SimPowerSystems и Simulink.-СПб.: Издательство "Питер". 2008 г.: 288 стр.

43. Diomidis Spinellis. Unix Tools as Visual Programming Components in a GUI-builder Environment. // Athens University of Economics and Business, September, 2001.

44. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” Int. Journal of Computer Simulation, special issue on “Simulation Software Development,” vol. 4, pp. 155-182, April, 1994.

45. J. Bier, E. Goei, W. Ho, P. Lapsley, M. O'Reilly, G. Sih and E. A. Lee, “Gabriel: A Design Environment for DSP,” IEEE Micro Magazine, October 1990, vol. 10, no. 5, pp. 28-45.

46. R. Allen and D. Garlan, “Formalizing Architectural Connection,” in Proc. of the 16th International Conference on Software Engineering (ICSE 94), May 1994, pp. 71-80, IEEE Computer Society Press.

47. D. C. Luckham and J. Vera, “An Event-Based Architecture Definition Language,” IEEE Transactions on Software Engineering, 21(9), pp. 717-734, September, 1995.

48. Rick Grehan. Visual Age for Java // Byte, July 1997.

49. Орлов С. Концепция визуального программирования в IBM VisualAge Smalltalk. Материалы конференции "Индустрия программирования '96".

50. www.hiasm.com

51. Компьютерные вести // Software, №44, 2004.