OSTIS-2013
(Open Semantic Technologies for Intelligent Systems)

УДК 681.3:519.6

ПРИНЦИПЫ ПОСТРОЕНИЯ ТЕХНОЛОГИИ ГРАФОСИМВОЛИЧЕСКОГО ПРОГРАММИРОВАНИЯ

Коварцев А.Н., Жидченко В.В., Попова-Коварцева Д.А., Аболмасов П.В.

Самарский государственный аэрокосмический университет им. академика С.П. Королева (национальный исследовательский университет), г. Самара, Российская Федерация

kovr_ssau@mail.ru

В статье рассматривается технология графосимволического программирования (ГСП), реализующая визуальный стиль программирования. Визуальное программирование повышает наглядность представления моделей алгоритмов программ, существенно уменьшает число ошибок, допускаемых на этапе проектирования и кодирования программ, и тем самым повышает надежность разрабатываемых программ. В данной работе обсуждаются наиболее важные аспекты технологии ГСП.

Ключевые слова: технология графосимволического программирования, проектирование сложных программ, автоматизация программирования, визуальное программирование, предметная область, онтологии, автоматизация тестирования, творческие задачи, искусственный интеллект

Введение

Настоящая статья подготовлена по материалам научных исследований, проводимых в течение нескольких лет в Самарском государственном аэрокосмическом университете имени С.П. Королева (СГАУ) в области технологий автоматизации программирования и визуального моделирования вычислительных процессов. На кафедре программных систем СГАУ это научное направление представлено технологией графосимволического программирования (ГСП) ([Коварцев, 1999]).

Широкое внедрение средств вычислительной техники в различные сферы жизни и деятельности человека стимулировало развитие автоматизированных методов и инструментальных средств, применяемых для создания прикладного программного обеспечения (ПО). Производство современного ПО происходит на фоне высоких требований, предъявляемых к качеству создаваемых программ при значительной сложности выполняемых ими функций. “Идеальной” технологией программирования можно считать такую технологию, которая по некоторому достаточно неформальному описанию объекта программирования автоматически генерирует текст синтаксически и семантически корректной программы.

В 80-х годах XX века сформировалась идея о том, что целью программирования является не столько порождение программы как таковой, а создание технологических условий, когда разрабатываемое программное обеспечение легко адаптируется к новым обстоятельствам и новому пониманию решаемой задачи. Р. Хемминг так формулирует этот тезис: “Здравая вычислительная практика требует постоянного исследования изучаемой задачи не только перед организацией вычислений, но также в процессе его развития и, особенно на той стадии, когда полученные числа переводятся обратно и истолковываются на языке первоначальной задачи”. Перечисленные выше обстоятельства привели к осознанию необходимости реализации интегрированного окружения поддержки всего жизненного цикла ПО, что обусловило появление инструментальных средств автоматизации проектирования программных систем (CASE-технологий).

В настоящие время наблюдается возрастание роли визуализации, используемой как в технологиях программирования, так и представлении результатов работы программ. Новый графический подход к решению проблемы автоматизации разработки ПО, основанный на идее привлечения визуальных форм представления программ, в большей степени соответствует образному способу мышления человека. Применение графических методов обещает кардинально повысить производительность труда программиста. Визуальное программирование, бесспорно, обладает достоинством наглядного представления информации и гораздо лучше соответствует природе человеческого восприятия, чем методы традиционного, текстового программирования.

Кроме того, графическая форма записи по сравнению с текстовым представлением программ обеспечивает более высокий уровень их структуризации, соблюдение технологической культуры программирования, предлагает более надежный стиль программирования. За счет использования графических моделей удается не только сократить время разработки параллельных вычислительных процессов, но и повысить их надежность, т.к. графическая нотация допускает формальное математическое описание модели, по которому может быть проведена ее автоматическая верификация и оптимизация.

Интеллектуальная сложность процессов разработки современного ПО настоятельно требует создания инструментальных средств представления, как абстракций, так и программных объектов на всех этапах их разработки. Предлагаемая технология ГСП в какой-то мере ориентирована на решение перечисленных проблем.

1. Концептуальная модель технологии ГСП

Технологию ГСП определим как технологию проектирования и кодирования алгоритмов программного обеспечения, базирующуюся на графическом способе представления программ, преследующую цель полной или частичной автоматизации процессов проектирования, кодирования и тестирования ПО.

Данная технология исповедует два основополагающих принципа:

- визуальную, графическую форму представления алгоритмов программ и других компонент их спецификации;
- принцип структурированного процедурного программирования.

Сразу условимся, что технология ГСП действует в рамках некоторой информационной среды, которую будем называть предметной областью программирования (ПрОП).

Под *предметной областью программирования* понимается среда программирования, состоящая из общего набора данных (словарь данных) и набора программных модулей (библиотека программных модулей).

Для представления алгоритмов в технологии ГСП используется модель объекта с дискретными состояниями. Основу такой модели составляет предположение о том, что для любого объекта программирования тем или иным способом можно выделить конечное число состояний, в которых он может пребывать в каждый момент времени. Тогда развитие вычислительного процесса можно ассоциировать с переходами объекта из одного состояния в другое. В математике такая концепция в качестве способа абстрагирования плодотворно используется достаточно давно: Марковские цепи, теория массового обслуживания, теория формальных грамматик и автоматов, моделирование систем и т.д.

Для уточнения понятия *состояния*, определимся с используемой в технологии ГСП концепцией модели алгоритма. Существует следующие три основных типа универсальных алгоритмических моделей:

Первый тип модели связывает понятие алгоритма с наиболее традиционными понятиями математики - вычислениями и числовыми функциями. Наиболее известная и изученные модели такого типа - *рекурсивные функции*.

Второй - основан на представлении об алгоритме как о некотором детерминированном устройстве, способном выполнять на каждом шаге лишь примитивные операции. Одним из многочисленных представителей этого типа является *машина Тьюринга*.

Третий - это преобразование слов в произвольных алфавитах, в которых элементарными операциями являются подстановки. Среди моделей этого типа наиболее известны *канонические системы Поста, нормальные алгорифмы Маркова* и т.д.

В технологии графосимволического программирования используется первый тип формализации понятия алгоритма, когда программа A интерпретируется как некоторая вычислимая функция:

$$f : in(D) \to out(D),$$

где $in(D)$ - множество входных данных функции f, $out(D)$ - множество выходных данных функции f. Далее для простоты мы не будем разделять множество программных модулей $\{A_i\}$ и приписанных им вычислимых функций $\{f_j\}$. Предположим, что каждый модуль является вычислимой функцией и $A_i \in \mathfrak{F}$, где $\mathfrak{F}$ – множество вычислимых функций.

Более строго эта концепция исследована для вычислительной модели, названной *машиной Колмогорова* [Успенский, 1987]. Там же можно найти едва ли не единственное формальное определение понятия *состояния* вычислительного процесса. По Колмогорову, *состояния* - суть конструктивные объекты, под которыми в случае технологии графосимволического программ-мирования можно понимать ансамбли конкретизаций структур данных (входных или вычисляемых), используемых в алгоритме. На каждом шаге итерации реализуется переработка текущего состояния структур данных D в новое состояние D^* с помощью некоторой локальной функции $D^* = f_k(D)$. Процесс переработки $D^0 = D$ в $D^1 = f_1(D^0)$, D^1 в

$D^2 = f_2(D^1) = f_1(f_2(D))$ и т.д. продолжается до тех пор, пока не появится сигнал о получении решения.

Определим *граф состояний* G как ориентированный помеченный граф, вершины которого - суть состояния, а дугами отмечаются переходы системы из состояния в состояние.

Формально, *состояние* - это достижение объектом ℗ определенной конкретизации структур данных, требующее выполнения определенных действий над данными предметной области посредством вычислимой функции A_k. Поэтому каждая вершина графа помечается соответствующей локальной вычислимой функцией A_k. При этом следует помнить, что состояние графа S_j - это некое понятие («концепт»), связанное с описанием объекта ℗, а вычислимая функция A_k, приписанная состоянию, - функциональные действия, которые необходимо выполнить при переходе объекта в это состояние.

Одна из вершин графа, соответствующая начальному состоянию, объявляется начальной вершиной и, таким образом, граф оказывается *инициальным*.

Дуги графа проще всего интерпретировать как *события*. С позиций предлагаемого подхода, *событие* - это изменение *состояния* объекта ℗ предметной области программирования, которое влияет на развитие вычислительного процесса.

На каждом конкретном шаге работы алгоритма в случае возникновения коллизии, когда из одной вершины исходят несколько дуг, соответствующее *событие* определяет дальнейшей ход развития вычислительного процесса алгоритма. Активизация того или иного события так или иначе зависит от состояния объекта, которое в свою очередь определяется достигнутой конкретизацией структур данных D объекта ℗.

Для реализации *событийного управления* на графе состояний G введем множество предикативных функций $\boldsymbol{P} = \{ \boldsymbol{P_1}, \boldsymbol{P_2}, ..., \boldsymbol{P_l} \}$. Под *предикатом* будем понимать логическую функцию $P_i(D)$, которая в зависимости от значений данных D принимает значение равное 0 или 1.

Дугам графа G поставим в соответствие предикативные функции. Событие, реализующее переход $S_i \to S_j$ на графе состояний G, инициируется, если модель объекта ℗ на текущем шаге работы алгоритма находится в состоянии S_i и соответствующий предикат $P_{ij}(D)$ (помечающий данный переход) истинен.

В общем случае предложенная концепция допускает одновременное наступление нескольких событий, в том случае, когда несколько предикатов, помечающих дуги (исходящих из одной вершины), приняли значение истинности. В ГСП этот конфликт разрешается назначением приоритетов дугам графа.

Существенно, что изображение программ в виде ориентированного помеченного графа естественно для восприятия человеком. Направленная дуга служит очевидным изображением перехода из одного состояния вычислительного процесса в другое, вершина - выполняемой вычислительной функции, а в целом ориентированный граф наглядно представляет все пути, по которым может развиваться вычислительный процесс. В этом случае логические особенности разрабатываемого программного модуля проявляются в характерной для него топологии графа. Можно сказать, что графическое представление программ позволяет задействовать непосредственное образное восприятие, расширяя возможности человека при разработке и анализе сложных программ.

Определим универсальную алгоритмическую модель технологии ГСП четверкой

$M=<D, \mathfrak{F}, P, G>$, (1)

где D - множество данных (ансамбль структур данных) некоторой предметной области программирования; $\mathfrak{F}$ - множество вычислимых функций; P - множество предикатов, действующих над структурами данных предметной области D; G - граф состояний объекта ℗.

Представленная алгоритмическая модель является универсальной, поскольку допускает описание любых алгоритмов.

2. Типы данных

В понятии *типа данных* Т сигнатуры Σ обычно выделяют:

- *спецификацию типа* данных сигнатуры Σ;
- и соответствующую ей *реализацию типа* данных [Замулин, 1990].

Здесь под сигнатурой понимается пара $\Sigma = \langle B, \Omega \rangle$, где B - множество имен-основ (имена базовых типов базового языка программирования или производных от этих типов данных), а $\Omega = \left(B^* \times B\right)$ - индексированное семейство множеств имен операций, B^* - множество всех цепочек элементов множества B.

Например, $B = \{\text{int}, double, real, \ldots\}$,

$\Omega = \{+(double, double \to double),$

$+(double, real \to doubl\ e), \ldots\}.$

В таком определении типа данных отражаются два аспекта: пользовательский, когда программист, составляя свою программу, видит тип как спецификацию; и машинный, связанный со

способом реализации в ЭВМ обозначенного типа данных.

Типизация данных полезна не только в гносеологическом смысле, как инструмент изучения теории программирования, но и в чисто прикладном аспекте, позволяя избежать большого количества ошибок, поскольку заставляет программиста точно определять все используемые им объекты и лучше контролировать свою программу. В этом смысле использование чисто синтаксического аспекта спецификации типа данных можно считать недостаточным.

В технологии ГСП понятие типа данных расширено понятием *интерпретации* данного. Действительно, во многих предметных областях чисто "языковое" представление типа данного является неполным. Например, в физике в формуле $F = am$ все три компоненты формулы *F, a, m* имеют естественный языковый тип double с определенными на этом типе операциями (+, -, *, /). Однако каждое из перечисленных данных имеет самостоятельную смысловую интерпретацию (для физических параметров она определяется его размерностью): *F* - сила [н], *a* - ускорение [м/сек^2], *m* - масса [*кг*].

Очевидно, что передача в подпрограмму вместо ускорения *a* (тип double) скорости *v* (тип double), не вызовет синтаксической ошибки, но приведет к неправильному исполнению программы. Следует отметить, что такого рода ошибки обычно очень сложно распознаются. В качестве типа *интерпретации* данного в ГСП предлагается использовать теорию размерности.

Определим множество основ типа интерпретации данных S_{int} как множество образующих типов интерпретации и их производных. Например, для физических задач это множество имеет вид:

$$S_{\text{int}} = \{[\text{м}], [\text{кг}], [\text{сек}], [\text{м/сек}], \ldots\}.$$

Множество операций в простейшем случае может быть представлено естественными операциями над типами интерпретации $\Omega_{\text{int}} = \{+, -, *, /\}$. При описании типов интерпретации данных базовых модулей необходимо ввести множество аксиом A_{int}, действующих над типами интерпретаций. Множество A_{int} состоит из замкнутых σ-формул. Например, для формулы $F = am$ аксиомой типообразования служит формула: $T_1 = T_2 * T_3$.

Тип *интерпретации данного* определим как $T_{int} = T_{int} = \langle \Sigma_{int}, A_{int} \rangle$, где $\Sigma_{int} = \langle S_{int}, \Omega_{int} \rangle$. Таким образом, под **типом данного** в технологии ГСП понимается пара $T = \langle \Sigma, \Sigma_{int} \rangle$.

Если определение некоторого типа данного не расширено семантическим аспектом - описанием типа интерпретации, то считается, что данный тип имеет "пустую" размерность «[ ]».

При верификации используемых типов данных в ГСП первоначально проверяется синтаксическая составляющая описания типа, а затем сравниваются типы интерпретации.

Спецификация данных в технологии ГСП реализована в виде совокупности таблиц информационного фонда системы. В отдельную таблицу, называемую *архивом типов данных*, сведены описания всех необходимых атрибутов типов данных, начиная с имени типа, описания родового языкового типа или его редукции, описания типа интерпретации и заканчивая описанием области возможных значений (доменов).

3. Базовые модули

В качестве исходного «строительного материала» в технологии ГСП выступают две категории: базовые модули (подпрограммы) и типы данных. Базовые модули представляют собой перечень локальных вычислимых функций, на основе которых в конечном итоге порождаются все объекты технологии ГСП. Типы данных описывают синтаксический и семантический аспекты строения данных, используемых в базовых функциях, а также и в объектах технологии ГСП.

Порождение первоначального множества вычислимых функций (базовых модулей) производится на любом из существующих языков программирования, например, на языке C++.

В технологии ГСП под **базовым модулем** будем понимать независимую программную единицу, реализующую определенную функцию в процессе преобразования некоторого агрегата данных.

4. Типы модулей

Введем понятие ***типа модуля*** *(вычислимой функции)* как обобщение понятия типа операции, которое определим как множество отображений из области определения функции в область результата.

Причем областью определения функции считается декартово произведение множеств значений нескольких типов данных (типов формальных параметров), а областью результатов - множество значений некоторого одного типа данных. При этом тип функции изображается $T_{i_1}, T_{i_2}, \ldots, T_{i_n} \to T_{j_1}, T_{j_2}, \ldots, T_{j_m}$, где $T_{i_1}, T_{i_2}, \ldots, T_{i_n}, T_{j_1}, T_{j_2}, \ldots, T_{j_m}$ - типы формальных параметров (исходных и результатов вычислений) рассматриваемой функции.

С каждым типом функции (подпрограммы) связывают две операции: создания функции и аппликации функций к своим аргументам. Первая операция в языках программирования связана с изображением функции в виде: заголовок функции плюс тело функции. Вторая операция - в виде обращения к функции.

В теории программирования подпрограммы это одно из фундаментальных средств абстрагирования и искусственного расширения возможностей языка, однако, правила реализации первой из перечисленных выше операций - составления тела подпрограммы, обычно синтаксически неотличимы от правил кодирования основного текста программы. Последнее достигается за счет введения понятия формальных параметров. В то время как между основной программой и подпрограммами имеет место существенная разница.

При разработке основной программы преследуется конкретная практическая цель, например, произвести расчеты некоторого технического устройства, физического явления, математических формул и т.д. При этом данным, используемым в программе "придается" вполне конкретный смысл (давление, скорость, пропускная способность и т.п.). В подпрограммах формальные параметры лишены конкретной смысловой нагрузки.

Действительно, в подпрограммах важен лишь тип параметра и порядок его использования, а "осмысление" назначения параметров возникает только после их аппликации к фактическим параметрам. Например, процедура, реализующая формулу A = B*C, в одной интерпретации типов данных вычисляет силу F по заданным ускорению *a* и массе *m* материальной точки (F = *a m)*, в другой - путь S, по заданной скорости V и времени t (S = V t).

Таким образом, *тип базового модуля* определим как отображение типов данных из области определения на область их значений $B:T_{i_1},T_{i_2},...,T_{i_n} \to T_{j_1},T_{j_2},...,T_{j_m}$, которые реализуются в соответствии с принятыми в *B* операциями над типами данных.

5. Объекты технологии ГСП

В технологии ГСП в качестве программных единиц рассматриваются ***объекты.*** По способу порождения и функциональному назначению различают три типа объектов: ***акторы, агрегаты и предикаты***. Все они имеют конкретный содержательный смысл и действуют в рамках предметной области программирования.

В предметной области, как правило, заранее уже определен терминологический словарь данных (параметров, переменных или констант). Поэтому программирование в рамках технологии ГСП начинается с формирования ***словаря данных*** ПрОП, представляющего собой таблицу, в которой каждому данному присвоено уникальное имя, задан тип, начальное значение данного и краткий комментарий его назначения в ПрОП.

Кроме словаря данных и каталога типов данных, информационную среду определяют объекты ГСП. Под объектом понимается специальным образом построенный в рамках технологии ГСП программный модуль, выполняющий определенные действия над данными ПрОП.

5.1. Акторы

Одним из объектов технологии ГСП является ***актор***. Актор формируется из базового модуля путем привязки абстрактных типов данных базового модуля к данным предметной области. По сути дела *актор* порождается в результате аппликации базового модуля к своим аргументам.

Актор производит те же действия, что и породивший его базовый модуль, но над конкретными данными ПрОП. В отличие от базовых модулей, каждый актор является содержательным программным модулем, который выполняет понятные функции в рамках заданной предметной области. Акторы в технологии ГСП реализуют отображение над множеством данных предметной области:

$$A_k : D_k^{in} \to D_k^{out}, \qquad (2)$$

где $D_k^{in} = \{d_1^{in}, d_2^{in},..., d_n^{in}\}$ - множество входных данных актора A_k, $D_k^{out} = \{d_1^{out}, d_2^{out},..., d_n^{out}\}$ - множество выходных данных актора A_k.

Между базовым модулем и актором осуществляется односторонняя связь типа "один ко многим". Каждый актор имеет свой прототип в виде базового модуля, а на основании каждого базового модуля можно построить один или несколько акторов. Это свойство полиморфизма объектов позволяет избежать избыточности при порождении новых акторов, которые различаются между собой только привязкой к данным. Другими словами, на основе одного отлаженного и оттестированного базового модуля за счет механизма автоматизированной привязки по данным можно построить несколько корректных акторов, что позволяет значительно повысить уровень надежности порождаемых программных кодов.

Порождение актора производится путем формирования, так называемого, ***паспорта*** объекта. Процедура паспортизации базового модуля заключается в установке соответствия между списком типов данных базового модуля и данными предметной области, таким образом, что каждому формальному параметру (типу данных) ставится в соответствие конкретное данное ПрОП.

Соответствие между базовым модулем B_i и актором A_j порождает соответствие между подмножеством типов T_i данных и подмножеством самих данных D_j предметной области:

$$\begin{cases} B_i(T_i^{in}, T_i^{out}) \to A_j(D_j^{in}, D_j^{out}) \\ T_i = (T_i^{in}, T_i^{out}) \to D_j = (D_j^{in}, D_j^{out}) \end{cases}$$

Для стандартизованной формы базового модуля операция конкретизации типов данных сводится к процедуре формирования списка фактических

параметров, который определяется паспортом порождаемого актора. В этом смысле паспорт актора и базовый модуль полностью определяют актор ПРОП.

5.2. Предикаты

В процессе реализации алгоритма в рамках технологии ГСП передача управления между объектами осуществляется с помощью управляющих объектов-предикатов. Формально предикат представляет собой отображение из множества данных предметной области на множество логических значений “истина” или “ложь”: $P_k : (d_1, d_2, ..., d_m) \to \{0,1\}$.

5.3. Агрегаты

Объекты (акторы или предикаты) являются исходным материалом для визуального программирования. Результатом визуального программирования являются агрегаты. Агрегат создается в форме графа, в котором объекты ПрОП играют роль вершин и дуг. Дуги - предикаты, а вершины - акторы или агрегаты. Дуги графа определяют передачу управления от одной вершины к другой.

Формально агрегат представляет собой помеченный ориентированный граф с одной входной и несколькими выходными (концевыми) вершинами:

$G = \{ \mathfrak{F}, P \}$,

где $\mathfrak{F}=\{ A_1, A_2, ..., A_n \}$ - множество акторов, которые являются вершинами графа, $P = \{ P_1, P_2, ..., P_m \}$ - множество предикатов, которые представляют дуги графа.

Развитие вычислительного процесса в агрегате происходит путем передачи управления из одной вершины в другую, начиная с начальной. Этот процесс может быть завершен по двум причинам: либо достигнута концевая вершина графа, из которой нет исходящих дуг, либо из текущей вершины отсутствуют разрешенные другими предикатами переходы в другие вершины. Если в процессе передачи управления сложилась ситуация, когда истинными одновременно являются несколько предикатов, то управление будет передано по предикату, имеющему наибольший приоритет.

При таком подходе, между агрегатом в технологии ГСП и блок-схемой алгоритма существуют аналогии. Отличие заключается в том, что агрегат не имеет специальных управляющих блоков (условие и выбор), и передача управления всегда осуществляется посредством проверки предиката, который в частном случае может быть тождественно истинным. Это упрощает визуальный анализ алгоритма, за счет чего можно сократить число структурных ошибок. Например, ошибок, связанных с переусложненной структурой (неправильно вложенные циклы, неверная передача управления и т.п.), либо ошибок вызванных противоречиями в самом графе (непредусмотренные циклы).

В отличие от акторов и предикатов, которые полностью определяются своими паспортами, при порождении агрегата с помощью специального компилятора формируется текст нового объекта ПрОП, который после трансляции заносится в библиотеку объектных модулей ПрОП.

6. Модель межмодульного интерфейса

Проблема передачи информации от одной программы к другой традиционно представляет собой одну из наименее популярных проблем в среде программистов и одну из проблем, которая служит источником наибольшего количества ошибок в разрабатываемом программном обеспечении.

В технологии ГСП введён стандарт на организацию межмодульного информационного интерфейса. Стандарт обеспечивается выполнением пяти основных правил:

1). Вводится единое для всей предметной области хранилище данных, актуальных для ПрОП (общая память). Полное описание данных размещено в словаре данных ПрОП. Любые переменные, не описанные в *словаре* данных, считаются локальными данными для тех объектов ГСП, где они используются.

2). В пределах ГСП описание типов данных размещается централизовано в *архиве типов* данных.

3). В базовых модулях в качестве механизма доступа к данным допускается только передача параметров *по адресам* данных.

4). Привязка данных объектов ПрОП реализована в паспортах объектов ПрОП.

5). В технологии ГСП не рекомендуется использовать иные способы организации межпрограммных связей по данным.

Предложенный стандарт позволяет полностью отделить задачу построения межмодульного информационного интерфейса от кодирования процедурной части программы, а также частично автоматизировать процессы построения информационного интерфейса.

С информационной точки зрения каждый объект ГСП f_i (2) представляет собой функциональное отображение области определения объекта D_i^{in} на область значений D_i^{out}.

В общем случае $D_i^{in} \bigcap D_i^{out} \neq \varnothing$ (в объекте могут быть модифицируемые данные) и $D_i^{in}, D_i^{out} \in D$, где D - полная область данных ПрОП.

Формально, сущность проблемы организации передачи данных между объектами в рамках некоторого модуля-агрегата f_Σ можно определить как задачу построения области данных агрегата f_Σ - $D_\Sigma = D_\Sigma^{in} \bigcup D_\Sigma^{out}$ и установления соответствий между данными $D_\Sigma = \{d_1, d_2, ...d_{n_\Sigma}\}$ и данными $D_i = \{d_1^i, d_2^i, ..., d_{n_i}^i\}$ объектов $f_1, f_2, ..., f_m$, из которых составлен агрегат f_Σ.

В традиционном программировании любое, даже незначительное изменение структур данных в модулях вызывает необходимость "ручной" переделки соответствующих информационных связей ρ_j. В технологии ГСП этот процесс автоматизирован за счет использования отношений ρ_j, описанных и хранящихся отдельно от программной реализации объекта в «паспортах» модулей.

Отношение ρ_j в ГСП формируется "паспортизацией" типов данных базовых модулей, т.е. за счет "опредмечивания" формальных параметров базовых модулей. В этом смысле отношение ρ_j является по сути "паспортом" модуля и вместе с базовым модулем определяют понятие актора или предиката.

7. Управление вычислительным процессом

В технологии ГСП для объектов - агрегатов в зависимости от стиля программирования (последовательного или параллельного) используются различные схемы управления вычислительным процессом. Каждому стилю программирования соответствует собственная схема управления в агрегате при незначительных изменениях в синтаксисе изображения агрегата.

Объекты, порожденные из базовых модулей (акторы), управляются в соответствии с правилами, принятыми в базовом языке программирования. Синтез агрегата происходит на основе его графического изображения, представленного соответствующими структурами данных, которые хранятся в информационном фонде технологии ГСП.

В ГСП агрегат фактически состоит из двух компонент:

- универсальной управляющей программы граф-машины (GM), которая в соответствии со структурой графа управления каждого из агрегатов, управляет развитием вычислительного процесса на агрегатах ПрОП;
- структур данных описания графа управления каждого из агрегатов.

Централизация функций управления в рамках одной программы (граф-машины) на самом деле очень удобное решение, поскольку позволяет:

1. контролировать вычислительный процесс в целом, и в случае нештатных ситуаций, принимать системные решения;
2. реализовать сбор статистической информации о характеристиках надежности каждого из модулей; вычислительной сложности модулей; маршрутах развития вычислительного процесса и т.д.

8. Онтологический аспект технологии ГСП

Основоположник использования онтологий в области информационных технологий Томас Груббер [Gruber, 1991] определяет онтологию как «спецификацию концептуализации». При этом под «концептуализацией» можно понимать спецификацию знаний об окружающем мире, т.е. описание структуры Бытия безотносительно к какой-либо инженерной задаче. Для программистов естественнее и ближе понимание концептуализации как построение модели решаемой задачи, т.е. её концептуальной схемы.

Под формальной моделью онтологии ***O*** часто понимают [Скобелев, 2011] упорядоченную тройку вида

$$O = < C, R, F >, \quad (3)$$

где C – конечное множество понятий (концептов) предметной области, R – конечное множество отношений между понятиями предметной области, F – конечное множество функций интерпретации, заданных на понятиях и/или отношениях онтологии ***O***.

Сравним между собой концептуальные схемы (1) и (3). Множество понятий (концептов) предметной области C онтологии ***O*** можно связать с множеством состояний S модели M объекта ⓞ. Конечное множество отношений R онтологии ***O*** в технологии ГСП описываются графами состояний G_i. И, наконец, конечное множество функций интерпретации F – это множество вычислимых функций , $\mathcal{F}$ и предикатов P. Фактически модель алгоритма (1) является *описанием онтологии* разрабатываемого объекта ⓞ в данной предметной области. В совокупности, все модели объектов предметной области формируют описание её онтологии. Как правило, множество предикатов и, в меньшей степени, множество вычислимых функций общезначимы для всей предметной области и повторно используются в разных объектах ПрОП.

Данное обстоятельство наделяет технологию ГСП рядом полезных свойств. Удобной для программиста визуальной формы описания модели алгоритма. Сформированная средствами ГСП предметная область содержит информацию, необходимую для автоматической генерации

отчетов для разрабатываемых программных приложений. Но главной особенностью данной технологии является возможность построения модели алгоритма без использования заранее сформированного «плана» алгоритма. В процедурных языках программирования для написания программы решения некоторой практической задачи необходимо первоначально решить задачу, т.е. разработать алгоритм, а затем реализовать его в программных кодах. В технологии ГСП для разработки алгоритма, а, следовательно, и кодов программы, достаточно иметь лишь идею решения задачи.

Последнее связано, по-видимому, с присутствием онтологических аспектов в технологии ГСП. При разработке программных приложений в технологии ГСП пользователь описывает онтологию предметной области, вводя для неё новые данные, новые понятия и функциональные отношения по мере изучения объекта программирования, в итоге параллельно формируется и модель алгоритма программы, связывающая все эти понятия.

9. Пример. Задача «Ханойские башни»

Задача о «Ханойских башнях» - одна из наиболее известных головоломок, для которой разработано большое количество алгоритмов её решения [Легалов, 2002]. Суть задачи заключается в следующем.

Пусть имеются три стержня X, Y, и Z. На стержень X надето N дисков разного диаметра, упорядоченные в направлении убывания диаметров от основания стержня. Цель игры заключается в переносе всех дисков со стержня X на стержень Z (по одному диску за раз), используя при этом промежуточный стержень Y, Причем ни один диск большего диаметра нельзя ставить на диск меньшего диаметра.

Проиллюстрируем возможности технологии ГСП на примере решения задачи о ханойских башнях.

При разработке алгоритма решения будем руководствоваться следующими простыми принципами, входящими в условия задачи:

1. При перекладывании дисков всегда применяется правило о недопустимости установки диска большего диаметра на диск меньшего диаметра.
2. Соблюдать правило приоритетов для операций переноса дисков. Установить приоритеты операциям переноска дисков (в прядке убывания приоритетов) следующем порядке: $(X \to Z)$, $(X \to Y)$, $(Y \to Z)$, $(Y \to X)$, $(Z \to Y)$, $(Z \to X)$
3. По возможности желательно в первую очередь снимать диски со стержня X и переносить их на стержень Z.

Сформируем словарь данных предметной области. Данные, необходимые для решения задачи, приведены в таблице 1. Числом 777 обозначено основание стержней.

Таблица 1. Словарь данных

Имя данного	Тип	Нач. значение	Комментарий
Mx	STERJN	{1,2,3,5,6,777}	Массив дисков стержня X
My	STERJN	{777,0,0,0,0,0}	Массив дисков стержня Y
Mz	STERJN	{777,0,0,0,0,0}	Массив дисков стержня Z
X	int	777	Размер верхнего диска на стержне X
Y	int	777	Размер верхнего диска на стержне Y
Z	int	777	Размер верхнего диска на стержне Z
Nst	int	1	Текущий номер стержня

В дальнейшем нам потребуется программа выполняющая операцию переноса диска с одного стержня на другой.

Рассмотрим базовый модуль *perA_B(int A, int B, STERJN Ma, STERJN Mb)*, реализующий перенос верхнего диска с абстрактного стержня A на абстрактный стержень B. Сама по себе программа базового модуля достаточно проста: первый элемент массива Ma переносится на первое место массива Mb (при этом реализуются все необходимые перемещения остальных элементов этих массивов) и переменным A и B присваиваются размеры первых элементов массивов Ma и Mb.

На основе базового модуля *perA_B* путём привязки формальных параметров A, B, Ma, Mb к данным ПрОП X, Y, Mx, My порождаются акторы, связанные с понятиями «Перенос диска со стержня X на стержень Y» (это понятие условно обозначим «X->Y»). Соответствующую функциональность реализует актор: *perA_B(X, Y, Mx,My)*. Точно так же порождаются понятия «X->Z», «Y->Z», …., «Z->X» с помощью акторов: *perA_B(X, Z, Mx,Mz)*, *perA_B(Y, Z, My,Mz)*,…, *perA_B(Z, X, Mz,Mx)*.

Аналогичным образом вводятся понятие «Визуализация перемещений дисков» и актор, отображающий на экране дисплея положение дисков на стержнях: *move(Mx, My, Mz)*, а также понятия, связанные с управлением графическим режимом: «Инициализация графики», «Закрытие графического режима» и т.д.

Отметим, что многие понятия на схемах имеют свои графические образы, что упрощает восприятие граф-программ человеком.

Представим себе ситуацию, что уже выбран стержень, с которого будет сниматься диск. Необходимо построить алгоритм, определяющий на

какой стержень его необходимо переместить? Его несложно реализовать с помощью граф-программы «Перенос диска со стержня X», представленной на рисунке 1. В технологии ГСП такие объекты называются *агрегатами* и пополняют список функций интерпретации *R* онтологии предметной области.

Рисунок 1 – Агрегат «Перенос диска со стержня X»

На рисунке корневая вершина (обведена жирной рамкой) выводит на экран дисплея текущее состояние стержней. Вершины «X->Z» и «X->Y» связаны с понятиями переноса диска со стержня X на стержни Z и Y. Причем переход в вершину «X->Z» происходит при выполнении условия X<Z, а в вершину «X->Y» - при истинности выражения X<Y. На графе более приоритетные дуги обозначены «жирными стрелками». Безусловный переход обозначен символом «1». Переменная *Nst* определяет номер стержня, на который реализован перенос диска. На графе этот факт отражен в вершинах *Nst=3;* и *Nst=2;* Стержням X, Y, Z присвоены номера 1, 2, 3.

Аналогично строятся агрегаты: «Перенос диска со стержня Y», «Перенос диска со стержня Z ».

Первоначально все диски находятся на стержне X. С диска X, в общем случае, мы можем переместить верхний диск, либо на стержень Z, либо на стержень Y. Начальные действия построения агрегата «Ханойские башни» показаны на рисунке 2. Здесь вершина 1 «Инициализация графики» - устанавливает графический режим отображения информации

Вершина 2 привязана к агрегату «Перенос диска со стержня X», представленного на рисунке 1. Вершины 3 и 4 формально фиксируют факт переноса диска на стержни Y или Z (управляются предикатами *Nst==3* и *Nst==2*). Эти вершины не имеют привязок к каким-либо акторам (т.е. являются «пустыми»), но необходимы для исключения циклических переносов дисков между двумя стержнями. Иными словами, если, например, диск был перенесен на стержень Y, то выбор стержня для выполнения следующей операции должен быть реализован между стержнями X и Y. Это обстоятельство приводит к развитию алгоритма решаемой задачи, представленному на рисунке 3.

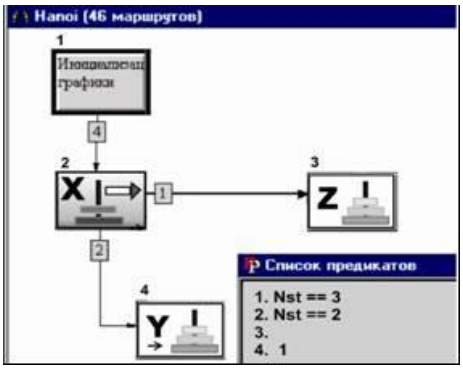

Рисунок 2 – Первый этап решения задачи

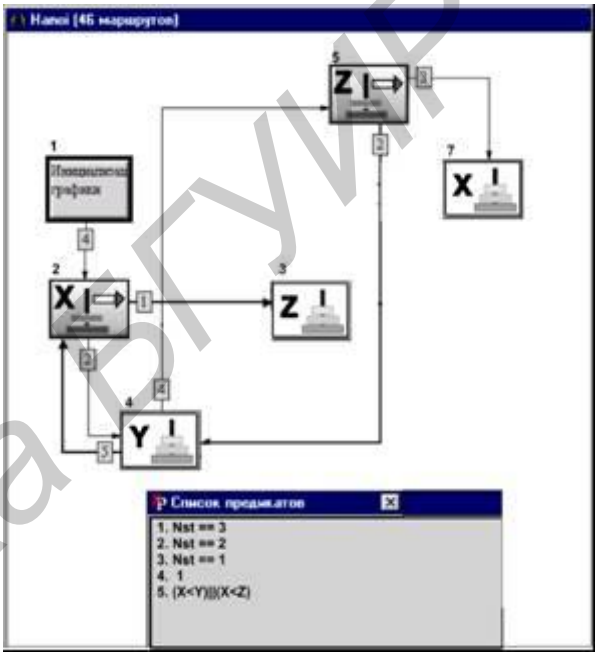

Рисунок 3 – Второй этап решения задачи

Как видно из рисунка, вершина 4 связана с вершинами 5 и 2, т.е. после переноса диска на стержень Y рассматриваются варианты возможности «снятия» дисков со стержней X или Z. В частности, переход в вершину 2 возможен в случае, если со стержня X имеется возможность переноса дисков на другие стержни, что управляется предикатом 5: (X<Y)||(X<Z). Иначе, следующий диск «снимается» со стержня Z.

Очевидно, что, если мы будем снимать диск со стержня Z (вершина 5), то в качестве «операционных» стержней следует рассматривать стержни X и Y (вершины 4 и 7). Продолжая в том же духе, получим окончательный вариант агрегата «Ханойские башни», обеспечивающий решение поставленной задачи (см. рисунок 4).

Технологические акторы 9, 10 и 11 необходимы для отображения результата работы алгоритма и закрытия графического режима работы монитора. Предикат №7 обеспечивает остановку работы алгоритма.

Таким образом, формируя агрегат решения задачи о «Ханойских башнях», мы эволюционно разработали алгоритм её решения, опираясь

фактически только на правила, принятые в рассматриваемой игре, и применяя разумные эвристики.

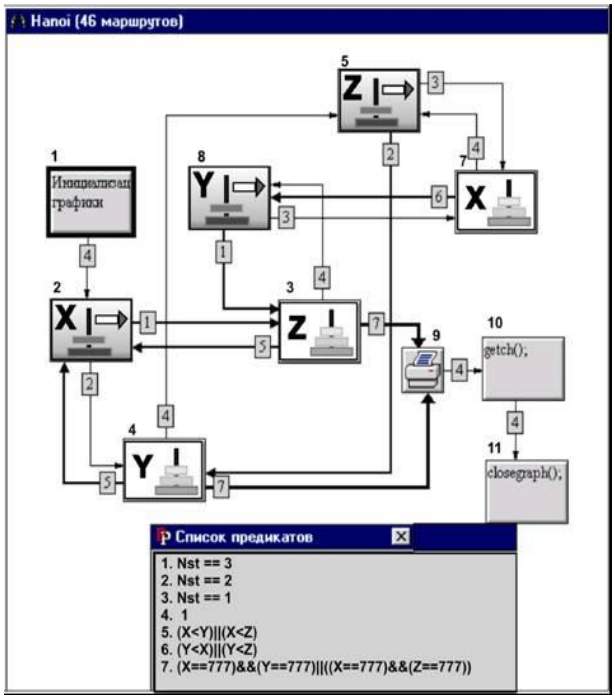

Рисунок 4 – Агрегат алгоритма решения задачи о ханойских башнях

Заключение

В работе были рассмотрены основные принципы построения технологии визуального программирования ГСП. Визуальное программирование повышает наглядность, представляемых кодов, существенно уменьшает число ошибок, допускаемых на этапе проектирования и кодирования программ, и тем самым повышает надежность кодов разрабатываемых программ.

На примере решения «творческой» задачи о ханойских башнях продемонстрированы возможности технологии ГСП с точки зрения применения визуальных форм представления модели алгоритма программы. Предлагаемый подход к разработке ПО, по существу, формирует онтологию в данной предметной области.

Рассмотрение технологии ГСП с этих позиций создает новые теоретические основы для дальнейшего развития технологии графосимволического программирования

Библиографический список

[Замулин, 1990] Замулин А.В. Системы программирования баз данных и знаний. - Новосибирск: Наука, 1990

[Коварцев, 1999] Коварцев А.Н. Автоматизация разра-ботки и тестирования программных средств. - Самарский государственный аэрокосмический университет, Самара, 1999. 148 с.

[Легалов, 2002] Легалов А.И. В лабиринтах Ханойских башен. «Программист», №11, 2002

[Мейер, 1982] Мейер Б., Бодуэн К. Методы программирования: В 2-х томах. Т.2.- М.: Мир, 1982

[Скобелев, 2011] Скобелев П.О. Онтологии деятельности для ситуационного управления предприятием в реальном времени. «Онтология проектирования», №1(3), 2011. С.6 – 39

[Успенский, 1987] Успенский В.А., Семенов А.Л. Теория алгоритмов: основные открытия и приложения. - М.: Наука, 1987. 288 с.

[Gruber, 1991] Gruber T.R. The role of common ontology in achieving sharable, reusable knowledge bases // Proc. of the 2 International Conf. – 1991. P 601-602

THE BASICS OF GRAPH-SYMBOLIC PROGRAMMING TECHNOLOGY

Kovartsev A.N., Zhidchenko V.V.,
Popova-Kovartseva D.A., Abolmasov P.V.

Samara State Aerospace University named after academician S.P. Korolyov (National Research University), Samara, Russian Federation

kovr_ssau@mail.ru

The article considers Graph-Symbolic Programming technology (GSP-technology), which implements visual programming style. Visual programming makes it easier to represent models of algorithms, significantly reduces the number of errors made during design and writing of programs, and thus improves the reliability of the developed programs.

Introduction

Modern software is complex. To reduce the complexity of software development and shorten the software development life-cycle the Graph-Symbolic Programming technology provides the means for visual description of algorithms and automatic synthesis of programs based on that description.

Main Part

The article introduces major components of GSP-technology:

- basic concepts;
- datatypes, modules and objects;
- interface between objects;
- calculation management;
- ontological aspects of GSP-technology
- example of visual programming in GSP-technology.

Conclusion

GSP-technology facilitates the development of numerical software. It provides visual environment not only for algorithm description but also for algorithm construction. In GSP it is possible to construct the algorithm “on the fly” using earlier created objects and creating new ones. The program is synthesized automatically on the basis of the algorithm. As a result development speed and program reliability improve.