УДК 004.415

ОНТОЛОГИЧЕСКИЙ АСПЕКТ МОДЕЛИ ВЫЧИСЛЕНИЙ ГРАФОСИМВОЛИЧЕСКОГО ПРОГРАММИРОВАНИЯ

А.Н. Коварцев

Самарский государственный аэрокосмический университет им. академика С.П. Королева (национальный исследовательский университет)
kovr_ssau@mail.ru

Аннотация

В статье рассматривается модель вычислений графосимволического программирования (ГСП), реализующего визуальный стиль разработки программ. Визуальное программирование повышает наглядность представления моделей алгоритмов, существенно уменьшает число ошибок, допускаемых на этапе проектирования и кодирования программ, и тем самым повышает надежность разрабатываемых программ. Последнее достигается во многом благодаря схожести методов структуризации предметной области, принятых в ГСП, и онтологического подхода к описанию окружающего мира. В данной работе обсуждаются онтологические аспекты технологии ГСП.

Ключевые слова: *модель вычислений, графосимволическое программирование, визуальное программирование, онтологии, 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) [1], реализующей *визуальный* стиль программирования. В ГСП алгоритм программы представляется графом управления, где в вершинах расположены вычислительные модули, а дугам приписаны логические функции.

Данная технология долгие годы успешно использовалась на кафедре при разработке различных программных приложений и в учебном процессе. Практическая эксплуатация данной технологии выявило ряд её полезных качеств. Комфортные условия, которые обретает пользователь технологии ГСП при разработке моделей алгоритмов программных приложений, а также снижение числа ошибок можно легко объяснить визуальной формой программирования и принятыми стандартами структуризации информации. Однако наиболее важное свойство, связанное с утвердившимся в ГСП стилем программирования, объяснить сложно. Дело в том, что в технологии ГСП разработка программного приложения происходит, как правило, эволюционно без использования заранее разработанной схемы (алгоритма) решения поставленной задачи. Точнее разработка алгоритма и построение программного приложения в ГСП в определенном смысле «сливаются». Этот феномен уже невозможно объяснить ни визуальным стилем программирования, ни структуризацией вводимой информации.

Именно это обстоятельство побуждает исследовать вычислительную модель, язык и технологию ГСП с позиций онтологических подхода. Тем более что в последнее время нарастает интерес к проблематике связей алгоритмов решения задач в предметной области с описы-

вающими предметную область формальными онтологиями (см., например [2-4]). Началу такого исследования и посвящена данная статья.

1 Модель вычислений технологии ГСП

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

Данная технология программирования исповедует два основополагающих принципа:

- использование визуальной, графической формы представления алгоритмов программ и других компонентов их спецификации;
- принцип структурированного процедурного программирования.

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

Под *предметной областью программирования* будем понимать некоторую среду программирования, имеющую общую цель - разработку программного обеспечения автоматизации расчетов в некоторой области практических интересов (авиационные двигатели, бизнес, медицинские приборы и т.д.), общую область данных и общую область знаний.

В качестве методологической основы для представления алгоритмов в ГСП используется модель объекта с дискретными состояниями. Основу такой модели составляет предположение, 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*, используемого в работе, определимся с принятой в технологии ГСП модели вычислений и концепцией алгоритма [1]. Можно выделить следующие три основных типа универсальных алгоритмических моделей:

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

Второй - основан на представлении об алгоритме как о некотором детерминированном процессе, реализуемом устройством, способным выполнять в каждый отдельный момент 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 ГСП используется первый тип формализации процесса вычислений, когда программа *A* интерпретируется как некоторая вычислимая функция:

$$f : \boldsymbol{in}(D) \rightarrow \boldsymbol{out}(D),$$

где D – структуры данных, представляющих ПрОП, $\boldsymbol{in}(D)$ - множество входных данных функции f, $\boldsymbol{out}(D)$ - множество выходных (вычисляемых) данных функции f.

Далее для простоты не будем разделять множество программных модулей $\{A_i\}$ и приписанных им вычислимых функций $\{f_i\}$. Предположим, что каждый модуль является вычислимой функцией и $A_i \in F$, где F - множество вычислимых функций.

Более строго данная концепция представления вычислительного процесса исследована для модели вычислений, названной *машиной Колмогорова* [5]. 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 ГСП 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 $D^2 = f_2(D^1) = f_2(f_1(D^0))$ 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 исполнительным объектом O 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 O, 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 O 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 дальнейший 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 O.

Для реализации *событийного управления* на графе состояний G введем множество предикативных функций $P = \{P_1, P_2, \ldots, P_l\}$. 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 $\rightarrow S_j$ на графе состояний G, инициируется, если модель объекта O 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 (без принятия дополнительных соглашений) допускает одновременное наступление нескольких событий, когда несколько предикатов, помечающих дуги (исходящих из одной вершины), приняли значение истинности. Возникает вопрос: на какое из наступивших событий объект программирования должен отреагировать в первую очередь?

Традиционное решение этой проблемы связано с использованием механизма приоритетов. В связи с чем, все дуги, исходящие из одной вершины, помечаются различными натуральными числами, определяющие их приоритеты. Отметим, что принятое уточнение обусловлено ресурсными ограничениями, свойственными однопроцессорной ЭВМ.

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 - выполняе-

мой 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 *M* технологии ГСП четверкой

(1) $$M = < D, F, P, G >,$$

где *D* - множество данных (ансамбль структур данных) некоторой предметной области программирования; *F* - множество вычислимых функций; *P* - множество предикатов, действующих над структурами данных предметной области *D*; *G* - граф состояний объекта *O*.

Представленная модель является универсальной, поскольку допускает описание любых алгоритмов.

Граф модели заменяет текстовую (вербальную) форму описания алгоритма программы, при этом:

1) Реализуется главная цель - представление алгоритма в визуальной (графосимволической) форме.
2) Происходит декомпозиционное расслоение основных компонент описания алгоритма программного продукта. Так структура алгоритма представляется графом *G*, элементы управления собраны во множестве предикатов *P* и, как правило, значимы не только для объекта *O*, но и для всей предметной области. Спецификация структур данных, а также установка межмодульного информационного интерфейса по данным «пространственно» отделена от описания структуры алгоритма и элементов управления.

Предложенная модель $M = < D, F, P, G >$ описывает, в конечном счете, некоторую вычислимую функцию $f_G(D)$ и в этом смысле может служить «исходным материалом» для построения алгоритмических моделей других программ. Последнее означает, что технология ГСП допускает построение иерархических алгоритмических моделей. Уровень вложенности граф-моделей в ГСП не ограничен.

Структура построенной модели во многом зависит от выбранного способа декомпозиционного расслоения объекта программирования на множество состояний *S* и множество событий, определяемых предикативными функциями *P*. В каждой конкретной предметной области эта задача решается индивидуально и, как правило, не вызывает затруднений.

2 Место онтологии в технологии ГСП

Кратко изложив методологические основы технологии ГСП, вернемся к основному вопросу данной статьи, связанному с определением «места» онтологических концепций в технологии ГСП.

Основоположник использования онтологий в области информационных технологий Томас Груббер [6] 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 ПрОП и решаемой задачи.

Под формальной моделью онтологии ***O*** часто понимают [7] упорядоченную тройку вида

(2) $$\boldsymbol{O} = < \boldsymbol{C}, \boldsymbol{R}, \boldsymbol{F} >,$$

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 (2). Множество понятий (концептов) предметной области ***C*** онтологии ***O*** можно связать с множеством состояний *S* модели *M* объекта *O*. 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 *F* и предикатов *P*. Фактически модель алгоритма (1) *содержит описание онтологии* разрабатываемого объекта *O* в данной предметной области. А 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. Кроме удобной для программиста визуальной формы описания модели алгоритма (данные аспекты были уже описаны выше) оказалось, что сформированная средствами ГСП предметная область содержит всю необходимую информацию для автоматической генерации отчетов разрабатываемых программных приложений. Но главной особенность данной технологии является возможность нового способа построения модели 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.

Отметим ещё одно обстоятельство. Обычно онтологии используются в так называемых интеллектуальных системах в интересах обеспечения их взаимодействия между собой и с человеком. Здесь онтологии рассматриваются как интерфейсы интеллектуальных систем. И как следствие, возникают задачи разработки стандартов представления данных, процедурных и декларативных знаний, языков описания знаний, а также методов и средств слияния онтологий. С точки зрения программиста, онтология напоминает систему управления базой данных, с той лишь разницей, что хранятся не данные, а знания. За построение онтологии отвечают одни программные средства, использование этих знаний - привилегия программных приложений. Одни программы хранят знания, другие содержат «движок», позволяющий использовать эти знания, например, делать выводы в экспертных системах, реализовывать интеллектуальный поиск необходимой информации на Web-ресурсах и т.п.

Однако существуют задачи искусственного интеллекта, для которых не всегда удобно производить подобное разделение программного приложения на хранилище знаний и исполнительную часть. К ним относятся, например, так называемые «*творческие задачи*» [8]. Для подобных задач онтологию предметной области сложно отделить от исполнительной части, реализующей её решение.

Рассмотрим в этом смысле возможности технологии ГСП на примере решения известной головоломки о «Ханойской башне».

3 Модель алгоритма решения задачи «Ханойские башни»

Задача о «Ханойских башнях» - одна из наиболее известных головоломок, для которой разработано большое количество алгоритмов решения. Суть задачи состоит в следующем.

Пусть имеются три стержня *X, Y,* и *Z*. На стержень *X* надето *N* дисков разного диаметра, упорядоченных 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.

Наиболее известным решением данной задачи, вошедшим во многие учебные пособия, является рекурсивный алгоритм [8, 9]. В работе [10] приводится параметрическое решение задачи о «Ханойских башнях», основанное на математическом подходе. Автоматный подход к решению задачи о «Ханойских башнях», позволивший устранить рекурсию, предложен в работе [11]. Однако все эти решения, так или иначе, основаны на идеях рекурсивного алгоритма, и при составлении программных реализаций решения алгоритм был известен заранее.

Попытаемся написать программу для задачи о «Ханойских башнях», не имея перед собой рекурсивного алгоритма её 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) в 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*.

Словарь данных предметной области, необходимых для решения задачи, приведен в таблице 1. Числом 777 обозначено основание стержней.

Таблица 1 – Словарь данных предметной области

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*	1	Размер верхнего стержня на диске *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.

Исходным «строительным» материалом для технологии ГСП являются *базовые модули* (функции интерпретации), составленные в кодах базового языка (в нашем случае С) и реализующие функциональные преобразования одних типов данных в другие. Фактически они выполняют математические преобразования одних формальных параметров программного модуля в другие. Семантический смысл базовые модули приобретают после выполнения операции привязки формальных параметров модуля к данным предметной области, т.е. после выполнения операции «паспортизации» модуля. В результате порождаются один или несколько *акторов* в технологии ГСП. Если базовые модули описывают абстрактные математические отношения над типами данных, то акторы выполняют вполне определенные преоб-

разования данных предметной области и с точки зрения её онтологии формируют множество функций интерпретации ***R***. На графе эти функции «привязываются» к его вершинам, образующим в совокупности множество понятий, входящих в онтологию ПрОП.

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 также 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 графического режима: «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 множество функций интерпретации ***R*** онтологии предметной области.

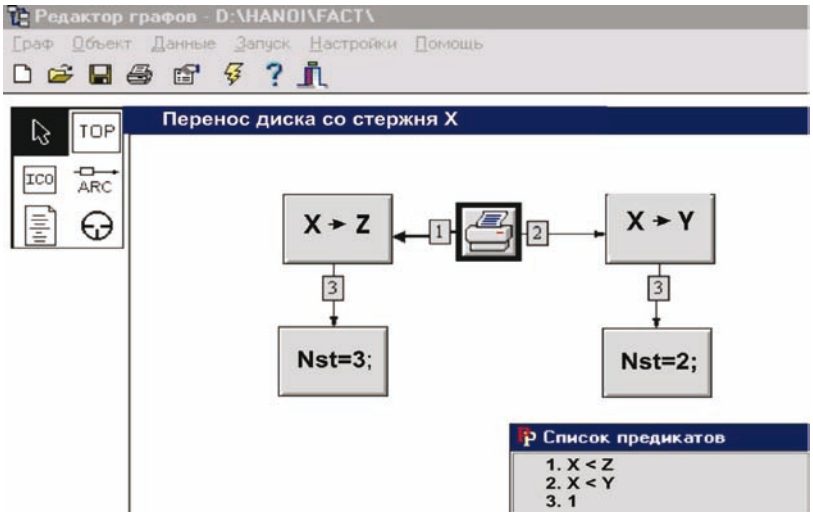

Рисунок 1 – Агрегат «Перенос диска со стержня *X*»

На рисунке 1 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.

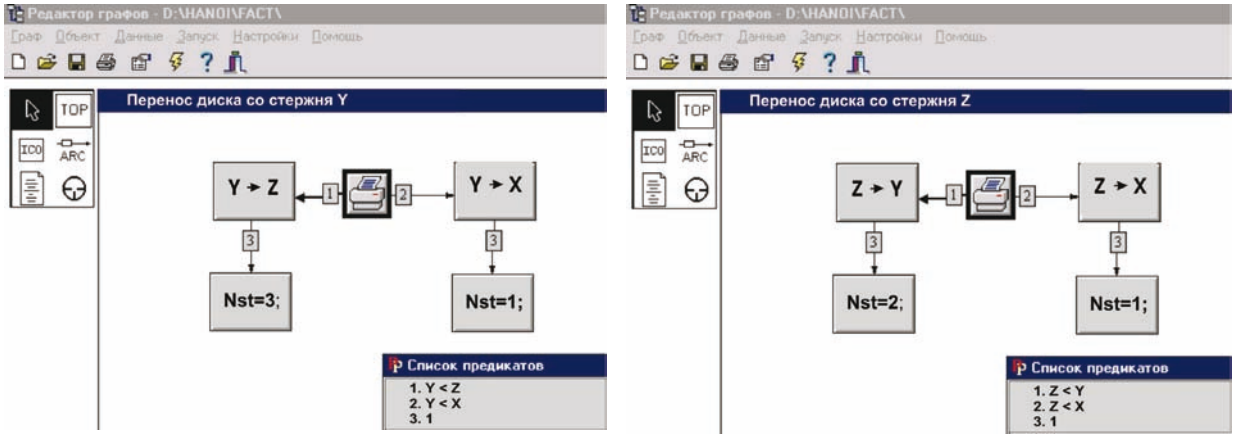

Рисунок 2 – Агрегаты «Перенос диска со стержня *Y*» и «Перенос диска со стержня *Z*»

Аналогично строятся агрегаты: «Перенос диска со стержня *Y*», «Перенос диска со стержня *Z* » (см. рисунок 2).

Теперь построим основную граф-программу решения задачи о «Ханойских башнях».

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 3. Здесь вершина 1 «Инициализация графики» устанавливает графический режим отображения информации.

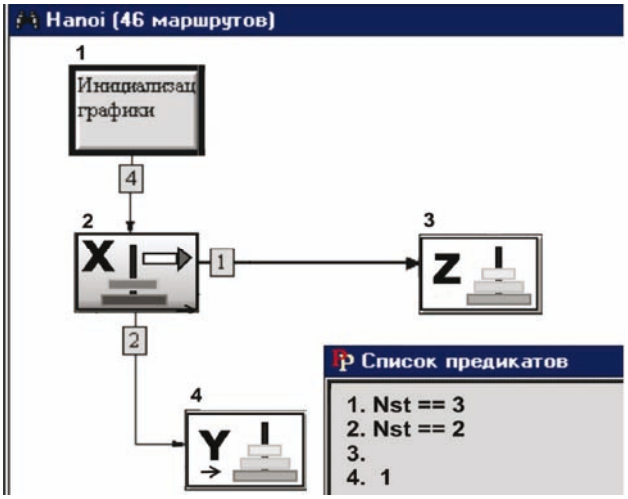

Рисунок 3 – Первый этап построения агрегата «Ханойские башни»

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 переноса 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 4.

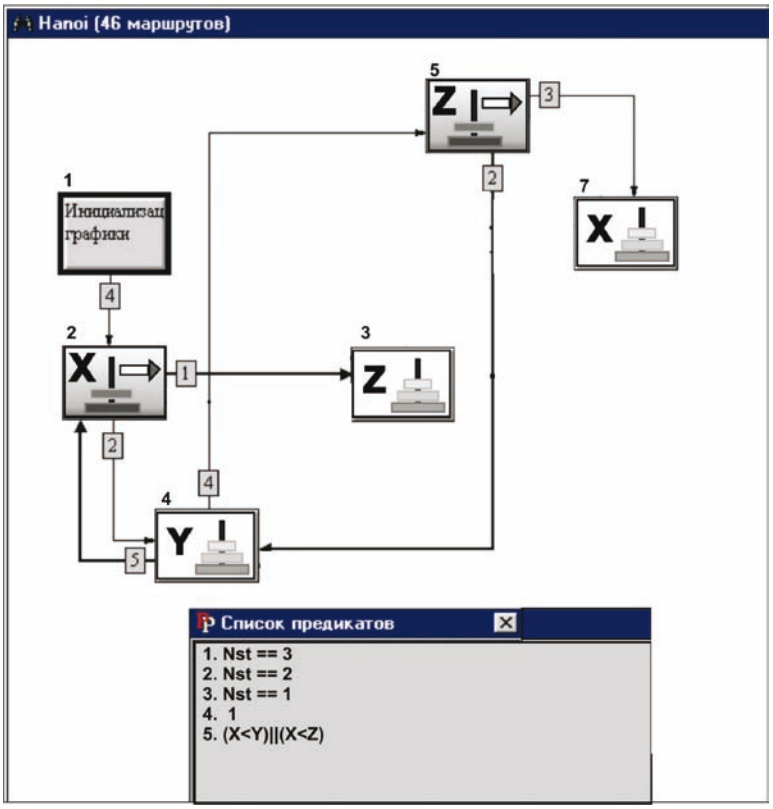

Рисунок 4 – Второй этап построения агрегата «Ханойские башни»

Как видно из рисунка 4 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 5).

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, работая над построением модели алгоритма решения задачи о «Ханойских башнях» мы фактически построили онтологию этой предметной области, основываясь на правилах принятых в рассматриваемой игре и применяя разумные эвристики. Не менее важным обстоятельством является то, что на основе построенного алгоритма решения игровой задачи в технологии ГСП автоматически генерируется исполнимый код программы.

Заключение

В работе представлены 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. Последнее во многом достигается благодаря присутствию в ней аспектов онтологических систем. Как было показано на примере решения «творческой» задачи, построение модели алгоритма её решения по существу формирует онтологию в данной предметной области.

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

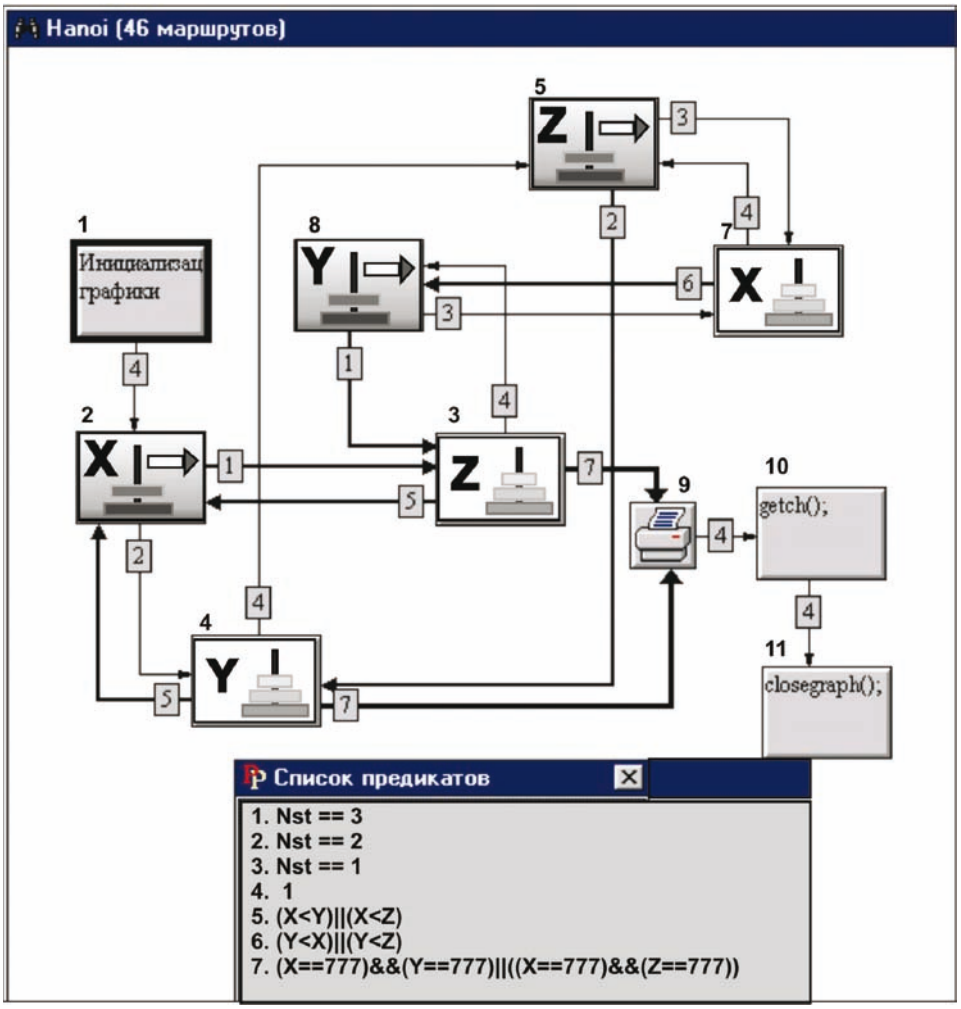

Рисунок 5 – Агрегат «Ханойские башни»

Список источников

[1] Коварцев А.Н. Автоматизация разработки и тестирования программных средств. – Самара: СГАУ, 1999. – 148 с.

[2] Евгенев Г.Б. Модели вместо алгоритмов. Смена парадигмы разработки прикладных систем // Информационные технологии. 1999. №3. – С. 38-44.

[3] Клещев А.С. Роль онтологий в программировании. Часть 1. Аналитика // Информационные технологии. 2008. №10. – С. 42-46.

[4] Смирнов С.В. Разработка пакетов прикладных программ и использующих их приложений на основе онтологического подхода // Перспективные информационные технологии для авиации и космоса (ПИТ-2010): Избранные труды Международной конф. с элементами научной школы для молодежи (29 сентября – 1 октября 2010 г., Самара, Россия). – Самара: СГАУ, 2010. - С. 235-238.

[5] Успенский В.А., Семенов А.Л. Теория алгоритмов: основные открытия и приложения. - М.: Наука, 1987. - 288 с.
[6] Gruber T.R. The role of common ontology in achieving sharable, reusable knowledge bases // Proc. of the 2 International Conf. 1991. - P. 601-602.
[7] Скобелев П.О. Онтологии деятельности для ситуационного управления предприятием в реальном времени // Онтология проектирования. 2012. №1(3). - С.6–39.
[8] Адаменко А.Н., Кучуков А.М. Логическое программирование. – СПб.: БХБ-Петербург, 2003. - 992 с.
[9] Мейер Б., Болдуэн К. Методы программирования. Т.2. – М.: Мир, 1982. - 368 с.
[10] Легалов А.И. В лабиринтах Ханойских башен // Программист. 2002. №11.
[11] Шалыто А.А., Туккель Н.И. Ханойские башни и автоматы // Программист. 2002. №8. - С. 82-90.

Сведения об авторе

Коварцев Александр Николаевич, 1952 г. рождения. В 1975 году окончил Куйбышевский авиационный институт (ныне – Самарский государственный аэрокосмический университет имени академика С.П. Королева) по специальности «Прикладная математика». Доктор технических наук (2000), профессор, заведующий кафедрой программных систем СГАУ. Директор школы информатики СГАУ. Коварцев А.Н. - специалист в области автоматизации проектирования, разработки и тестирования сложных программных средств; надежности программного обеспечения; моделирования параллельных вычислений и глобальной оптимизации. Имеет более 80 научных публикаций.

Alexander Kovartsev (b. 1952) graduated with honours (1975) from the S. P. Korolyov Kuibyshev Aviation Institute (presently, S. P. Korolyov Samara State Aerospace University (SSAU)), majoring in Applied Mathematics. He received his Doctor of Engineering (2000) degrees from Samara State Aerospace University. Professor, chair of SSAU's sub-department software systems SSAU. Director of the School of Computer SSAU. His current research interests include computer-aided design, development and testing of complex software; simulation of parallel computing and global optimization. He has more than 80 scientific publications.