ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 5
1. МЕТОД СИНТЕЗА ПАРАЛЛЕЛЬНЫХ ПРОГРАММ НА ВЫЧИСЛИТЕЛЬНЫХ МОДЕЛЯХ 8
1.1 Простые вычислительные модели (ПВМ) 8
1.1.1 Определение ПВМ 8
1.1.2. Синтез программ на ПВМ 15
1.1.3 Расширение описания ПВМ 18
1.1.3.1 Условные операции 18
1.1.3.2 Структурированные операции 19
1.2 Вычислительные модели с массивами 21
1.3 Постановка задачи 29
1.4 Родственные работы, относящиеся к проблеме 32
1.4.1 Система ПРИЗ 32
1.4.2 Т-система 33
1.4.3 Система Норма 34
2. ЯЗЫК ОПИСАНИЯ ПРЕДМЕТНОЙ ОБЛАСТИ 36
2.1 Требования к языку 36
2.1.1 Описание простых вычислительных моделей 36
2.1.1.1 Переменные 36
2.1.1.2 Использование подмоделей 37
2.1.1.2 Операции 41
2.1.2 Описание вычислительных моделей с массивами 44
2.1.3 Средства помощи планированию и генерации 45
2.2 Ограничения, принятые в реализации языка 46
2.3 Описание языка 49
2.3.1 Синтаксис и семантика основных конструкций языка 49
2.3.1.1 Лексика и общие правила описания 49
2.3.1.2 Вычислительные модели и переменные модели 50
2.3.1.3 Описание операций 55

2.3.1.3 Описание макроопределений ... 58
2.3.2 Описание моделей с массивами ... 59
2.3.3 Свойства операций и отображение массивов ... 63
3. РЕАЛИЗАЦИЯ ТРАНСЛЯТОРА ... 67
3.1 Общее описание программы-транслятора ... 67
3.2 Структура и описание работы транслятора ... 69
3.2.1 Разбор текста описания ... 69
3.2.2 Формирование табличного представления задачи ... 74
3.3 Внутреннее представление данных транслятора ... 75
3.4 Результаты работы транслятора ... 78
ЗАКЛЮЧЕНИЕ ... 80
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ ... 81

ВВЕДЕНИЕ

Параллельные вычисления сейчас являются одним из ведущих направлений в области информационных технологий. Вычислительная техника смогла добиться того уровня производительности, который мы имеем сегодня, только благодаря параллельному исполнению. Этот факт наиболее очевиден на примере современных суперкомпьютеров, которые имеют большое число процессоров, работающих одновременно. Кроме того, наиболее перспективные направления в разработке универсальных микропроцессоров (архитектура EPIC, многопотоковые архитектуры, SMP архитектуры) представляют собой различные варианты реализации параллелизма в пределах одного или нескольких процессоров.

Для эффективного исполнения программы на параллельной вычислительной системе очень важно, чтобы в программе можно было легко выделить независимые части вычислений для их одновременного выполнения на различных исполнительных устройствах. Известно, что средства автоматического распараллеливания в общем случае не позволяют создавать программы такого же качества, как параллельные программы, написанные вручную. Параллельное программирование по своей сути сложнее последовательного. Это в большой степени ограничивает круг знатоков параллельного программирования, а значит и число программ, эффективно работающих на современных компьютерах. Автоматическая генерация параллельных программ является одним из способов повышения качества программирования при существенном сокращении затрат на разработку программ.

Любая система конструирования программ работает с определенным набором знаний, на основании которых может быть сгенерирована программа. Представление знаний в памяти компьютера должно отвечать двум требованиям. С одной стороны оно должно быть удобным для пользователя системы, т.е., понятия, используемые в представлении, должны

быть близки понятиям предметной области, в которой работает пользователь. С другой стороны, необходимо, чтобы на основании принятого представления знаний была построена достаточно эффективная программа. Применительно к параллельному программированию эффективность определяется возможностью выделения независимых фрагментов вычислений. Сами фрагменты могут быть последовательны по своей структуре, что даже является более удобным с точки зрения создания и отладки. Система конструирования параллельных программ в этом случае может представлять собой систему сборки программ из готовых фрагментов. Тогда представление знаний должно определять связи между отдельными фрагментами вычислений. К преимуществам такого подхода можно отнести возможность эффективного применения большого числа разработанных ранее последовательных программ.

В работе рассмотрено представление знаний в виде вычислительных моделей. Оно отвечает требованиям удобства и эффективности, так как позволяет формировать систему знаний в виде совокупности абстрактных объектов, соответствующих объектам предметной области, и не привносит «лишних» зависимостей между объектами. Таким образом, оно сохраняет весь внутренний параллелизм содержащихся в нем алгоритмов.

Основное направление использования параллельных вычислительных систем – это решение так называемых «больших задач», т.е. задач, требующих использования большого объема данных и вычислений. Наличие в таких задачах массовых данных и массовых вычислений является естественным. Тем не менее, в существующих системах, построенных на основе использования вычислительных моделей, отсутствует встроенная поддержка массовых вычислений. С другой стороны, существуют системы, эффективно решающие большие задачи из относительно узкой предметной области, например, решение задач математической физики сеточными методами.

Цели работы:

- разработка языка описания предметной области в виде совокупности вычислительных моделей с массивами,
- реализация транслятора, преобразующего текстовое описание задачи на вычислительных моделях во внутреннее представление, удобное для ее дальнейшей обработки системой конструирования параллельных программ.

Разработанный язык позволяет формировать описание предметной области в виде совокупности вычислительных моделей с массивами. Он содержит набор вспомогательных средств описания для помощи пользователю в формировании системы знаний. Также средствами языка пользователь имеет возможность предоставить системе конструирования дополнительную информацию по обработке описания задачи.

Разработанный транслятор выполняет проверку правильности текстового описания задачи и переводит его во внутреннее табличное представление, удобное для его дальнейшей обработки (планирования алгоритма решения задачи и генерации программы).

1. МЕТОД СИНТЕЗА ПАРАЛЛЕЛЬНЫХ ПРОГРАММ НА ВЫЧИСЛИТЕЛЬНЫХ МОДЕЛЯХ

1.1 Простые вычислительные модели (ПВМ)

1.1.1 Определение ПВМ

При решении задач из различных предметных областей часто приходится сталкиваться с проблемой представления знаний о данной предметной области в форме, удобной для машинной обработки. Представление знаний в виде совокупности объектов и их свойств удобно прежде всего тем, что таким образом строится вся система знаний человека. В памяти ЭВМ любой объект представляется абстрактной моделью, содержащей необходимый набор активных (пригодных для машинной обработки) знаний об объекте. Именно таким средством являются вычислительные модели. Понятие «вычислительная модель» (как семантическая сеть) было впервые предложено Э. Х. Тыугу в конце 60-х годов.

Как в реальном мире объект определяется своими свойствами, характеристиками, так и в памяти ЭВМ модель объекта определяется набором числовых характеристик, значения которых могут быть вычислены функциональными преобразователями (операциями). Таким образом, вычислительная модель составляют два множества:

- множество переменных – некоторых характеристик описываемого объекта;
- множество отношений, определяющих возможность и способы вычисления значений одних переменных при известных значениях других. Эти отношения будем называть операциями.

Естественно, с помощью этих двух множеств можно построить только очень простую систему знаний. Для описания более сложных объектов и

систем может понадобиться расширение понятия «вычислительная модель», введение в нее каких-то дополнительных элементов и связей.

Самая элементарная из всех модификаций вычислительных моделей – это простая вычислительная модель. Ее структура представляет собой конечную совокупность простых, т.е. неиндексированных переменных, связанных конечной совокупностью операций. Определим простую вычислительную модель более формально. Пусть заданы:

1. конечное множество $\mathbf{X}=\{x, y, z, \ldots\}$ переменных для представления вычисляемых и измеряемых величин;
2. конечное множество $\mathbf{F}=\{a, b, c, \ldots\}$ символов операций арности $m\times n$, $m\geq 0$, $n\geq 0$; m и n различные для различных операционных символов;
3. с каждым символом операции a арности m×n связан набор $in(a)=(x_1, \ldots, x_m)$ входных и набор $out(a)=(y_1,\ldots,y_n)$ выходных переменных, при этом $i\neq j \rightarrow y_i\neq y_j$.

Пара $\mathbf{M}=(\mathbf{X}, \mathbf{F})$ называется простой вычислительной моделью (ПВМ). Операция $a\in\mathbf{F}$ описывает возможность вычисления переменных $out(a)$ из переменных $in(a)$ с помощью некоторой процедуры. Таким образом, в простой вычислительной модели отсутствуют более специфические объекты и механизмы описания: массивы, счетчики, предикаты, неявные функции, рекурсия и т.д. Графически простую вычислительную модель удобно представлять в виде двудольного ориентированного графа с вершинами двух типов: переменные и операции. Дуги графа показывают возможные направления вычислений. Дуга графа, ведущая из переменной x в операцию a, показывает, что переменная x является входной для операции a. Дуга графа, ведущая из операции a в переменную y, показывает, что переменная y является выходной для операции a (операция a вычисляет переменную y). Вычислительная модель отражает принципиальную возможность вычисления одних параметров описываемого объекта через другие.

Пример простой вычислительной модели

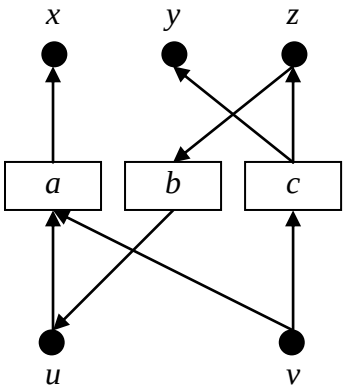


Рис. 1.1

Рассмотрим простую вычислительную модель, представленную на рисунке 1.1. Здесь x, y, z, u, v – переменные, a, b, c – операции. Пусть необходимо вычислить переменную x. Анализируя структуру связей модели, можно увидеть, что переменная x может быть вычислена операцией a, входными переменными которой являются u и v. Таким образом, алгоритм вычисления переменной x можно представить термом: $t_1=a(u, v)$. Дальше, рассматривая модель, замечаем, что переменная u может быть вычислена с помощью операции b, требующей на входе переменную z. Переменная z, в свою очередь, вычисляется операцией c, имеющей входную переменную v. Получаем, что порядок вычисления переменной x может быть представлен следующими термами:

$t_1=a(u, v)$; $t_2=a(b(z), v)$; $t_3=a(b(c(v)), v)$.

Пусть $X \subseteq \mathbf{X}$, $F \subseteq \mathbf{F}$. Введем определение множества термов $T(X,F)$, порожденных X и F:

1. Если $x \in X$, то $x \in T(X,F)$.
2. Пусть термы $t^1,\ldots,t^s$ входят в $T(X,F)$. Пусть $a \in F$, $in(a)=(x_1,\ldots,x_s)$. Тогда терм $t=a(t^1,\ldots,t^s)$ включается во множество $T(X,F)$ тогда и только тогда, когда $\forall i(x_i \in out(t^i))$.

Графически терм можно представить в виде дерева. В его корне находится операция, которая непосредственно вычисляет целевую переменную. Дуги, ведущие в корень дерева, берут свое начало от корней поддеревьев, которые описывают термы, вычисляющие входные переменные операции в корне дерева. Например, терм $t_3=a(b(c(v)), v)$ из предыдущего примера будет иметь следующий вид:

Графическое представление терма

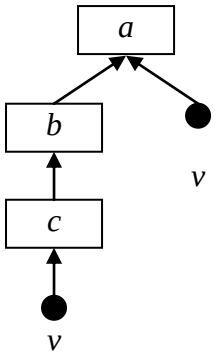


Рис. 1.2

Это обычное представление терма показывает только входные переменные и операции, которые участвуют в вычислении. В дальнейшем более полезным будет расширенное представление, показывающее также все используемые промежуточные переменные. Тогда терм t_3 будет иметь вид:

Расширенное графическое представление терма

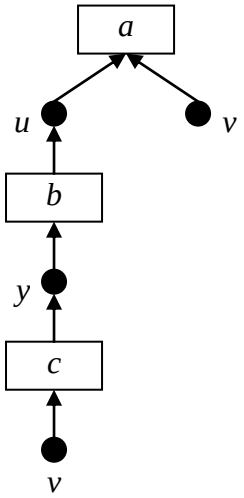


Рис. 1.3

Можно сказать, что терм представляет развернутый путь движения по графу модели от входных переменных до целевой переменной. Множество *T(X,F)* может быть бесконечным, если в графическом представлении части вычислительной модели, порожденной множествами *X* и *F*, имеет место цикл.

Пусть на простой вычислительной модели заданы два множества переменных: $V \subseteq \mathbf{X}$ и $W \subseteq \mathbf{X}$, $W \neq \varnothing$. Тогда можно сформулировать задачу: по заданным значениям переменных из *V* найти значения переменных из *W*. Это стандартная постановка задачи на вычислительной модели [lit1]. Первым шагом ее решения будет построение совокупности термов, описывающих все возможные вычисления из переменных *V* переменных *W*.

Рассмотрим множество $T(V,\mathbf{F})$. Оно представляет все различные способы проведения вычислений из переменных множества *V*, осуществляемых операциями множества **F**. Среди термов множества $T(V,\mathbf{F})$ есть такие, которые не вычисляют значения переменных *W*. Введем определение двойственного множества $\tilde{T}(X,F)$ термов, которые вычисляют множество переменных *X*, используя операции из множества *F*:

1. Если $x \in X$, то $x \in \tilde{T}(X,F)$.
2. Пусть $t \in \tilde{T}(X,F)$, $x \in in(t)$; $a \in F$, $in(a)=(x_1,\ldots,x_m)$; $x \in out(a)$. Тогда терм $S^{x}_{a(x_1,\ldots,x_m)}(t)$ включается во множество $\tilde{T}(X,F)$. Символ $S^{X}_{t'}(t)$ означает замещение некоторых вхождений *X* в терме *t* термом *t'*.

Пересечение $T=T(V,\mathbf{F}) \cap \tilde{T}(W,\mathbf{F})$ выражает множество всех способов вычисления *V* из *W*. Причем, оно не содержит «лишних» операций: любой терм из *T* вычисляется на основе переменных из *V* и участвует в вычислении какой-либо переменной из *W*. Множество *T* содержит все возможные способы решения поставленной задачи. Для конкретной задачи, как правило, совсем не обязательно знать все возможные пути ее решения. Более того, в нетривиальных задачах множество *T* чаще всего бесконечно. Поэтому достаточно иметь одно или несколько хороших решений, которые будут

описываться некоторым подмножеством $R \subseteq T$. Любое подмножество $R \subseteq T$ такое, что оно содержит термы для вычисления всех переменных из W на основе переменных из V, будем называть (V,W)-планом вычислений или просто планом.

Пусть $V \subseteq \mathbf{X}$ – множество переменных. V-интерпретация I, далее просто интерпретация, есть тройка:

$$I=(\Delta, \delta_0, \{f_a: a \in \mathbf{F}\}).$$

Здесь Δ – множество значений, которые могут принимать переменные из V, δ_0 – функция, сопоставляющая каждой переменной из V конкретное значение из Δ (δ_0: $V \to \Delta$), f_a – интерпретация операционного символа a, т.е. функция $f_a=I(a)$: $\Delta^m \to \Delta^n$, где m=|in(a)| – число входных, n=|out(a)| – число выходных переменных оператора a. Отображение $f_a=I(a)$ естественным образом распространяется на множество термов $T(V,F)$: каждому терму сопоставляется суперпозиция преобразований, соответствующих операциям в его записи:

$$I(a(t_1,\ldots,t_m))=f_a(I(t_1),\ldots,I(t_m)).$$

Пусть $t=a(x_1,\ldots,x_m)$, $(x_1,\ldots,x_m)=\mathrm{in}(a)$, $\{x_1,\ldots,x_m\} \subseteq V$, $\mathrm{out}(a)=(y_1,\ldots,y_n)$. Интерпретация I естественным образом сопоставляет набору переменных out(a) вектор-значение

$$\mathrm{val}(t) = (d_1,\ldots,d_n) = f_a(\delta_0(x_1),\ldots, \delta_0(x_m)).$$

Пусть теперь $t=a(t_1,\ldots t_m)$ – произвольный терм, такой что $\mathrm{in}(a)=(x_1,\ldots,x_m)$, $\mathrm{out}(a)=(y_1,\ldots,y_n)$. Тогда набору переменных out(a) сопоставляется вектор-значение

$$\mathrm{val}(t)=(d_1,\ldots,d_n)=f_a(\mathrm{val}_{x_1}(t_1),\ldots,\mathrm{val}_{x_m}(t_m)),$$

где $\mathrm{val}_z(\tau)$ обозначает значение компоненты вектора val(τ), соответствующий переменной z.

Если имеется два различных вычисления из одних и тех же начальных данных V каких-либо двух значений переменной y, то, по смыслу модели, эти значения должны быть эквивалентными по некоторому отношению

эквивалентности, в простейшем случае равны. Далее будем рассматривать только корректные интерпретации, т.е. такие, при которых для любой переменной *y* и любых термах *t* и *t'*, вычисляющих *y*, значения $val_y(t)$ и $val_y(t')$ совпадают.

Пусть имеется набор вычислительных модулей, выполняющих различные вычисления в некоторой предметной области. В процессе разработки вычислительной модели для предметной области, ее закономерности выражаются в синтаксисе модели, т.е. характере связей между модулями, и в семантике модели, т.е. специфике этих модулей. Семантика определяет интерпретацию операций модели. Однако когда необходимо чтобы найти цепочку для вычисления одной переменой из другой, семантические особенности модулей уже не важны. В процессе планирования достаточно пользоваться синтаксисом моделей, абстрагируясь от специфики модулей. Таким образом, получаем, что вычислительная модель является неинтерпретированным объектом. В принципе, одна и та же вычислительная модель может соответствовать различным предметным областям. Именно абстрагирование от конкретной интерпретации позволяет с помощью анализа лишь структурной составляющей создавать универсальные алгоритмы синтеза с реальными оценками сложности.

Простая вычислительная модель **M**=(**X**, **F**) описывает множество термов *T(***X**,**F***)*, порожденных **X** и **F**, т.е. все возможные варианты проведения вычислений из переменных множества **X** с использованием операций из множества **F**. Такое представление знаний о вычислениях является максимально непроцедурным, т.е. допускает наибольшее число реализаций (способов вычисления). Это имеет большое значение для параллельного исполнения. Сейчас существует большое количество параллельных вычислительных систем, отличающихся по своим параметрам. На каждой из них будет оптимальным свой вариант исполнения. Таким образом,

представление знаний в виде простых вычислительных моделей позволяет абстрагироваться от конкретной реализации вычислительной системы.

1.1.2. Синтез программ на ПВМ

Построив простую вычислительную модель, мы можем сформулировать на ней стандартную задачу вычисления множества переменных V из W. Из всех возможных вариантов решения задачи, которые предоставляет вычислительная модель, необходимо выделить некоторое подмножество, т.е. выбрать (V,W)-план $R \subseteq T=T(V,\mathbf{F}) \cap \tilde{T}(W,\mathbf{F})$.

Одним из параметров, которые нужно учитывать при выборе подмножества R, является надежность плана. Вследствие разных причин операции в составе модели, а, следовательно, и содержащие их термы, могут иметь различную надежность вычислений. Надежность повышается путем увеличением избыточности вычислений и использования более «надежных» термов. Обычно ставится задача получения минимального плана R с заданной степенью надежности, в том смысле, что любое подмножество $L \supset R$ уже не обеспечивает заданной надежности.

Чтобы ограничить числа термов в плане до конечного числа, в [lit1] вводится понятие функции надежности терма, основанное на том, что надежность терма не больше надежности любого его подтерма. Это условие позволяет сократить все множество термов T, и оставить в нем только бесповторные термы. Терм t называется бесповторным, если в графическом представлении t на любой его ветви каждая операция встречается не более одного раза для вычисления любой из своих выходных переменных. Т.е., если операция a, $out(a)=\{x,y\}$, используется в бесповторном терме t, то в некоторой ветви t могут быть два фрагмента вида $...a \rightarrow x...$ и $...a \rightarrow y...$, но не может быть двух фрагментов вида $...a \rightarrow x...$ и $...a \rightarrow x...$.

Таким образом, задача планирования (построения плана) заключается в построении множества T_V^W бесповторных термов. Пользователя, как правило, интересует наиболее оптимальный план, т.е. отвечающей определенному

критерию или набору критериев оптимальности. Существует много критериев выбора (V,W)-плана, каждый из которых будет наиболее подходящим для какого-то конкретного случая, например:

- минимальное число термов в плане – для сокращения числа ветвей вычислений.
- минимальное число переменных, от которых зависят термы в составе плана («входных» переменных) – можно использовать для сокращения медленных операций ввода данных.
- наименьшая максимальная глубина самого глубокого терма, где глубина терма определяется числом операций в самой длинной цепочке между какой-либо входной переменной и вершиной терма – критерий сокращает длину последовательной части вычислений, которая определяет время всего вычисления при его параллельной реализации.

Пусть с каждой операцией связан некоторый вычислительный модуль, который реализует соответствующую процедуру вычисления. Таким образом, вычислительная модель позволяет решать не только те задачи, решение которых было явно реализовано в виде модулей, но и синтезировать новые решения. Естественно, можно так построить вычислительную модель, что различные пути вычисления одной переменной будут давать различные результаты. Модели с такой некорректной интерпретацией мы рассматривать не будем, т.е. любой алгоритм вычисления какой-либо переменной дает один и тот же результат.

Конструирование программы состоит из построения (V,W)-плана (выбор алгоритма) и записи на некотором алгоритмическом языке программы $П$, реализующей построенный план. Выбор плана и генерация программы должны быть сделаны таким образом, чтобы функционал $Ф$, который определяет качество программы $П$ и задан на множестве всех программ, реализующих всевозможные (V,W)-планы, достигал экстремума на $П$. Лишь для некоторых простых функционалов (например, глубина термов

(V,W)-плана) задача конструирования оптимальной программы может быть решена за полиномиальное время. Для большинства практически важных функционалов (например, время выполнения программы) решение этой задачи имеет экспоненциальную сложность. Оно зависит от свойств вычислительной системы, от свойств функционала Φ, от свойств выбранного алгоритма решения задачи. Более того, при различных входных данных могут быть оптимальны различные алгоритмы и, соответственно, реализующие их программы.

В силу названных причин в большинстве случаев не удается получить оптимальную программу автоматически. Поэтому на этапе планирования и генерации должны применяться человеко-машинные алгоритмы. Пользователь должен иметь возможность скорректировать процесс планирования и генерации, например, путем задания простого функционала, а также путем указания переменных и операций, использование которых недопустимо (нежелательно, желательно, обязательно). Качество полученной программы пользователь может оценить с помощью тестовых испытаний. Если качество программы его не устраивает, пользователь должен иметь возможность проведения оптимизирующих преобразований плана и соответствующей программы, вплоть до прямых указаний о распределении ресурсов и о структуре программы. Конечно, полученная таким образом программа не будет являться оптимальной в строгом смысле, поэтому здесь можно говорить только о псевдо-оптмальности.

При отсутствии указаний пользователя система конструирования программ также должна построить какую-то программу, решающую поставленную задачу. При этом она может на основе имеющейся информации исключить заведомо плохие решения.

Генерация программы по готовому алгоритму является последним этапом работы системы конструирования. В [lit1] рассматриваются

следующие три варианта конструирования параллельной программы на основе (V,W)-плана:

1. Программа представляет собой совокупность параллельных присваиваний переменным из W вырабатывающих их термов (V, W)-плана.
2. Программа – последовательно-параллельное задание вычислений с помощью операторов ветвления и слияния fork, join. При этом она имеет структуру, подобную структуре терма. В начале вычисления каждого терма выполняется ветвление (fork) для параллельного вычисления его подтермов, после чего выполняется слияние (join) и вычисление операции-вершины терма.
3. Асинхронная программа (А-схема), задающая вычисление с помощью спусковых функций. Она состоит из А-блоков – блоков, определяющих атомарное вычисление. Структура связей А-блоков повторяет структуру фрагмента вычислительной модели, который состоит из переменных и операций, входящих в алгоритм решения задачи. Каждый А-блок соответствует одной операции модели и содержит спусковую функцию, которая определяет наличие входных переменных операции. Как только все входные переменные операции получили значение, соответствующий А-блок срабатывает, и его операция вычисляет значения своих выходных переменных. Программа, сформированная таким образом, обладает наибольшей асинхронностью.

1.1.3 Расширение описания ПВМ

1.1.3.1 Условные операции

Простые вычислительные модели обладают существенным недостатком: они не содержат другой информации о применимости отношений, кроме той, которая определяется указанием входных и выходных

переменных операций. Поэтому на простых вычислительных моделях могут строиться только линейные участки вычислений. Это ограничение снимается с введением условных операций.

Условной операцией будем называть такую операцию, которая может исполняться только при выполнении некоторого условия, т.е. при определенном состоянии описываемого объекта. Предусловием условной операции будем называть логическое выражение, зависящее от переменных модели, при истинности которого данная операция может быть применена. Таким образом, вводится множество **P** предикатных символов. Каждой операции $a \in \mathbf{F}$ соответствует предикатный символ (предусловие) $p_a \in \mathbf{P}$. С каждым p_a связан конечный набор переменных $\mathrm{in}(p_a)=\mathrm{in}(a)$. Операции, для которых предусловие является тождественно истинным, будем называть безусловными.

Полезность введения в ПВМ условных операций очевидна. Часто некоторая операция имеет ограниченную область определения, и при каких-то значениях переменных модели не может применяться. При использовании условных операций появляется возможность включения такой операции в модель без каких-либо недопустимых последствий. Кроме того, на различных входных данных могут быть эффективны различные операции, правила выбора которых также можно определить с помощью предусловий.

1.1.3.2 Структурированные операции

Имея описания некоторой вычислительной модели, мы можем влиять на формирование плана вычислений только путем «внешних» модификаций, например, запрещать либо требовать использование определенных переменных и операций. При этом изменять реализацию операций в составе вычислительной модели мы не имеем права. Тем не менее, существует множество ситуаций, когда в различных случаях планирования вычислений на одной и той же модели необходимо использовать различные реализации одной операции. Это, например, возможно при решении задач на различных

вычислительных системах. Возможным решением проблемы будет включение в модель всех возможных операций, эквивалентных по входным и выходным данным, но с разными предусловиями, чтобы затем можно было выбрать подходящую к каждому конкретному случаю. Но такой вариант не самый лучший, так как существует опасность «засорения» вычислительной модели множествами эквивалентных операций на все случаи жизни. При этом даже самая тривиальная модель может разрастись до невероятных размеров, что затруднит ее понимание и использование программистом.

Введение структурной интерпретации простой вычислительной модели позволяет снять указанную проблему. Пусть G – конечное множество ПВМ, в каждой ПВМ $\mathbf{B}=(\mathbf{X_B},\mathbf{F_B})$, $\mathbf{B}\in G$, существует $(V_\mathbf{B},W_\mathbf{B})$-план $T_\mathbf{B}$. Структурной интерпретацией ПВМ $\mathbf{C}=(\mathbf{X_C},\mathbf{F_C})$ называется функция SI, которая сопоставляет:

1) каждой операции b, $b\in \mathbf{F_C}$, вычислительную модель $\mathbf{B}=SI(b)$, $\mathbf{B}\in G$;
2) каждой переменной $x\in \mathrm{in}(b)$ переменную $v=SI(x)$, $v\in V_\mathbf{B}$; каждой переменной $y\in \mathrm{out}(b)$ переменную $w=SI(y)$, $w\in W_\mathbf{B}$, и, кроме того,

$$\forall\, u(u\in V_\mathbf{B} \vee W_\mathbf{B})\rightarrow \exists\, x(x\in \mathrm{in}(a) \vee \mathrm{out}(a) \wedge u=SI(x))).$$

Операция b называется структурированной операцией и говорится, что операция b раскрывается ПВМ $\mathbf{B}$. Предполагается также, что в $\mathbf{B}$ нет операции b, и при раскрытии структурированных операций $\mathbf{B}$ никогда не встретится операция b. Интерпретация I, которую в отличие от структурной будем называть содержательной, может быть естественным образом доопределена. Таким образом, структурная интерпретация $SI(b)=\mathbf{B}$ задает внутреннюю структуру операции b, а в качестве алгоритма вычисления функции f_b может быть взят любой $(V_\mathbf{B},W_\mathbf{B})$-план, построенный на ПВМ $\mathbf{B}$.

С помощью одной структурированной операции пользователь, формирующий описание предметной области, в состоянии охватить все множество существующих методов решения некоторой подзадачи. Он также имеет возможность выбрать нужный уровень детализации операций, т.е.

использовать тот уровень элементарности операций, который присущ описываемому объекту.

Выделение в описании модели структурированных операций позволяет разбить процесс планирования на этапы. Для больших вычислительных моделей со многими компонентами и операциями это имеет существенное значение. Вместо того, чтобы учитывать все переменные и операции, имеющие отношения к задаче, планировщик берет во внимание только отдельный уровень описания. Выполнив планирование на этом уровне, планировщик рассматривает попавшие в план операции, и, если среди них оказались структурированные, выполняет планирование на них. Вместе с тем, ничто не мешает планировщику при необходимости использовать обычный подход, просто включив вычислительную модель вместо структурированной операции, произведя необходимые отождествления переменных.

1.2 Вычислительные модели с массивами

Практически все вычислительные задачи требуют массового применения операций. Как только появляется необходимость обрабатывать множество однородных данных или многократно выполнять какое-то действие, без массовых операций уже не обойтись. Синтез программ на простых вычислительных моделях не позволяет конструировать эффективные параллельные программы, так как в них нельзя задавать массовое применение операций. Это приводит к необходимости помещать все массовые вычисления внутрь модулей. В большинстве задач на массовые вычисления приходится основное время выполнения параллельной программы, поэтому их необходимо распараллеливать. С использованием механизма простых вычислительных моделей сделать это нельзя, так как элементарные операции, предназначенные для массового применения, не могут самостоятельно входить в простую вычислительную модель. Поэтому

чрезвычайно важно ввести в вычислительные модели возможность описания массовых вычислений.

Понятие вычислительной модели с массивами вводится в два этапа. Введем сначала понятие упрощенной вычислительной модели с массивами. Пусть заданы:

1. Множество переменных $\mathbf{X}=X \cup Y$, где $X=\{x, y,\ldots,z\}$ – конечное множество, элементы X называются простыми переменными. Множество Y – счетное либо пустое; непустое множество Y разбито на конечное число счетных, непересекающихся, линейно упорядоченных подмножеств, которые называются массивами. Элементы массива $\bar{x}=\{x_1,\ldots,x_n,\ldots\}$ называются компонентами $\bar{x}$; компонента $\bar{x}_n$ обозначается $\bar{x}[n]$; множество всех массивов обозначается $\bar{Y}=\{\bar{x}, \bar{y},\ldots,\bar{z}\}$, $X \cap Y=\varnothing$.
2. Множество $\mathbf{F} \subseteq F_1 \times N$, где N – множество натуральных чисел; $F_1=\{a, b,\ldots,c\}$ – конечное множество функциональных символов (операций). Элемент $(a, i) \in \mathbf{F}$ обозначается a^i и называется i-м экземпляром a. Операция а называется простой, если $N_a=\{n \in N|(a, n) \in \mathbf{F}\}$ – одноэлементное множество, и массовой, если N_a содержит более одного элемента. Функция **In** сопоставляет каждому экземпляру a^i единственное конечное множество входных $\mathrm{in}(a^i)=\mathbf{In}(a, i)$, а функция **Out** – единственное конечное множество выходных $\mathrm{out}(a^i)=\mathbf{Out}(a, i)$ переменных, $\mathrm{in}(a^i) \cup \mathrm{out}(a^i) \subseteq \mathbf{X}$, $\mathrm{in}(a^i) \neq \varnothing$, $\mathrm{out}(a^i) \neq \varnothing$.

Если некоторое свойство выполняется для всех экземпляров операции a, то будем говорить, что это свойство выполняется для операции a. На множества $\mathrm{in}(a^i)$ и $\mathrm{out}(a^i)$ накладываются обычные ограничения, связанные с тем, что все a^i реализуются одним программным модулем:

1) $\mathrm{in}(a^i) \neq \mathrm{in}(a^j)$;
2) $|\mathrm{in}(a^i)|=|\mathrm{in}(a^j)|$ и $|\mathrm{out}(a^i)|=|\mathrm{out}(a^j)|$;

3) множества in(a^i) и in(a^j), а также out(a^i) и out(a^j) могут различаться только компонентами одноименных массивов, т.е. не допускается, чтобы, например, in(a^1)={$\bar{x}$[1], $\bar{x}$[3]}, а in(a^2)={$\bar{x}$[2], $\bar{y}$[3]}.

Набор **C**=(**X**, **F**, **In**, **Out**) называется упрощенной вычислительной моделью с массивами (УВММ). На **C** обычным образом определяются понятия терма, множества термов T_V^W, которое в общем случае может быть бесконечным, (V, W)-плана вычислений. При определении T_V^W вместо операции следует использовать понятие экземпляра операции. Множества V и W могут быть бесконечными, например, V может содержать все компоненты массива $\bar{x}$ с четными номерами, т.е. $\{\bar{x}[2], \bar{x}[4],\ldots\} \subseteq V$.

Пример упрощенной вычислительной модели с массивами показан на рисунке 1.4.

Пример упрощенной вычислительной модели с массивами

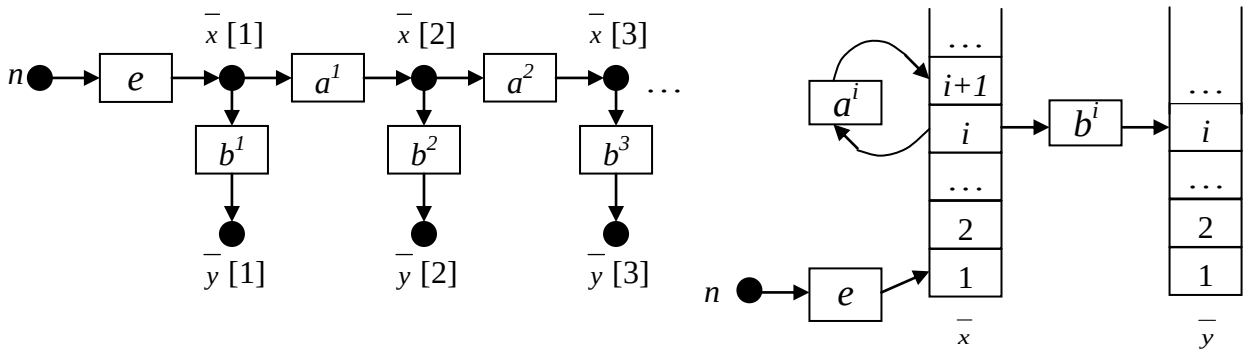


Рис. 1.4

Если в УВММ общее число экземпляров операций конечное, то, соответственно, число используемых переменных тоже будет конечным, и на УВММ можно планировать вычисления так же, как и на простой вычислительной модели. В общем случае в УВММ экземпляров операций бесконечное счетное множество, поэтому необходимы дополнительные средства, чтобы ограничивать перебор операций при планировании. Такие средства вводятся в вычислительных моделях с массивами. Пусть заданы:

1. Множество переменных $\mathbf{X}=X \cup Y \cup Z$, где X и Y – те же множества простых переменных и компонентов массивов, что и в УВММ, а Z – конечное множество переменных, называемых счетчиками, $Z \cap X=\varnothing$, $Z \cap Y=\varnothing$; каждому массиву $\bar{x} \in \bar{Y}$ соответствует счетчик из Z, который обозначается $n_{\bar{x}}$.
2. Множество $\mathbf{F} \subseteq F_1 \times N$ экземпляров операций определяется так же, как и в УВММ, с тем дополнением, что их входные и выходные наборы могут содержать счетчики.

Пример вычислительной модели с массивами

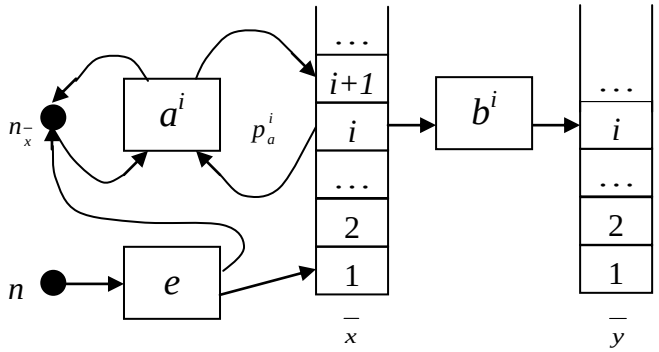


Рис. 1.5

Счетчики являются особыми переменными, которым может сопоставляться счетное множество значений. Рассмотрим массовую операцию a, изображенную на рисунке 1.5.

$$\mathrm{in}(a^i)=\{n_{\bar{x}}, \bar{x}[\mathrm{i}]\},\ \mathrm{out}(a^i)=\{n_{\bar{x}}, \bar{x}[\mathrm{i+1}]\}$$

При выполнении различных экземпляров операции a счетчик $n_{\bar{x}}$ будет получать разные значения. Для вычисления переменной $\bar{x}[5]$ с помощью экземпляра операции a^4 нужно значение счетчика $n_{\bar{x}}$, которое вычисляет a^3. Но синтаксически правильными будут не только термы вида $t=a^4(a^3(t_1,t_2),t_3)$, но и любой терм вида $t=a^4(a^j(t_1^j,t_2^j),t_3^j)$. Необходимо средство, чтобы ограничить порядок выполнения операций, т.е. указать, какие экземпляры

операций могут вычислять значение счетчика, которое a^4 может использовать в качестве входного. Для этой цели вводится функция **Θ**. Она каждой паре ($n_{\bar{x}}$,a^i), $n_{\bar{x}} \in \mathrm{in}(a^i)$ сопоставляет конечное множество экземпляров операций, таких, что $n_{\bar{x}}$ является их выходной переменной, и только они могут использоваться для вычисления счетчика $n_{\bar{x}}$ для экземпляра a^i.

3. Множество $\mathbf{P} \subseteq P_1 \times N$, где P_1 – конечное множество предикатных символов. Каждой операции $a \in F_1$ соответствует предикатный символ (предусловие) $p_a \in P_1$, Каждому экземпляру $a^i \in \mathbf{F}$ – элемент (p_a, i), обозначаемый p_a^i. С каждым p_a^i связан конечный набор переменных $\mathrm{in}(p_a^i)=\mathrm{in}(a^i)$.

Набор **M**=(**X**, **F**, **In**, **Out**, **P**, **Θ**) называется вычислительной моделью с массивами (ВММ). Пусть $V \subseteq \mathbf{X}$, $W \subseteq \mathbf{X}$, $V \cap W = \varnothing$. Предполагается, что если какие-то компоненты массива $\bar{x}$ входят в V, то и $n_{\bar{x}} \in V$, это же справедливо и для W. Терм $t \in T_V^W$ тогда и только тогда, когда выполняются условия:

1. t является безповторным термом;
2. в каждом подтерме t вида $f(t_1,\ldots,t_n)$, $n_{\bar{x}} \in \mathrm{in}(f)$, счетчик $n_{\bar{x}}$ вычисляется либо термом $b(t_1',\ldots,t_m')$, $b \in \mathbf{\Theta}(n_{\bar{x}},f)$, либо термом $t''=n_{\bar{x}}$, $n_{\bar{x}} \in V$.
3. Если $n_{\bar{x}} \in \mathrm{in}(f)$, то любой терм $b(t_1',\ldots,t_m')$, $b \in \mathbf{\Theta}(n_{\bar{x}},f)$ вычисляет одно и то же значение $n_{\bar{x}}$.

Для того, чтобы на ВММ можно было описывать итеративные процессы, необходимо зафиксировать направление изменения значений счетчика: если какой-то терм вычисляет значение счетчика $n_{\bar{x}}=i$, то любой его подтерм может вычислять только значения $n_{\bar{x}}=j<i$. Таким образом, теперь на ВММ можно описать правило перебора экземпляров массовой операции, чтобы построить какое-то итеративное вычисление. Предикатные символы здесь необходимы для ограничения числа итераций: итеративный

процесс может продолжаться, пока могут выполняться входящие в его состав операции.

Рассмотрим УВММ, представленную на рисунке 1.4. Пусть поставлена задача: вычислить $\bar{y}$ из n, т.е. $V=\{n\}$, $W=\bar{y}=\{y[1], y[2], \ldots\}$. Алгоритм решения задается потенциально бесконечным множеством термов T_V^W :

$t_1=b^1(e(n))$;

$t_2=b^2((a^1(e(n))$;

…

$t_n=b^n(a^{n-1}(\ldots a^1(e(n))\ldots))$;

…

Так как множество T_V^W потенциально бесконечно, оно должно представляться алгоритмом, который перечисляет термы из T_V^W в некотором порядке. Для конструирования этого алгоритма на основе УВММ формируется простая вычислительная модель (рисунок 1.6), которая определяет экземпляры всех операций, используемых в термах множества T_V^W .

ПВМ, полученная из УВММ после выделения ψ-операции

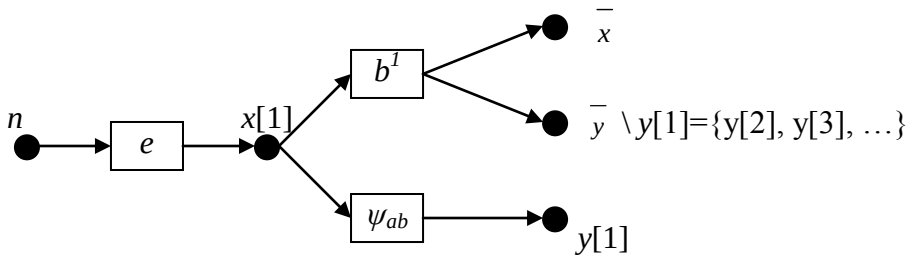


Рис. 1.6

Операция ψ_{ab} задается параметрическим термом $\bar{x}[i]\rightarrow a\rightarrow \bar{x}[j]\rightarrow b\rightarrow \bar{y}[j]$. Здесь параметрами являются значения счетчиков i и j с областью применимости M={(i, j): $i\in$ N, $j=i+1$}, где N – множество натуральных чисел. На области применимости M можно легко задать

отношение порядка, в соответствии с которым можно автоматически построить алгоритм перечисления элементов M: (1, 2), (2, 3), …. Каждому элементу M соответствует конкретный терм: $\bar{x}[1] \rightarrow a \rightarrow \bar{x}[2] \rightarrow b \rightarrow \bar{y}[2]$, $\bar{x}[2] \rightarrow a \rightarrow \bar{x}[3] \rightarrow b \rightarrow \bar{y}[3]$, …. Множество этих термов определяет множество всех используемых экземпляров операций a и b. Объединив их множества входных переменных, получим множество $\{\bar{x}[1], \bar{x}[2], \bar{x}[3], \ldots\}$, но отношение порядка позволяет вычислить элементы $\{\bar{x}[1], \bar{x}[2], \bar{x}[3], \ldots, \bar{y}[2], \bar{y}[3], \ldots\}$ только из $\bar{x}[1]$. Поэтому $in(\psi_{ab})=\{\bar{x}[1]\}$, $out(\psi_{ab})=\{\bar{x}, \bar{y}[2], \bar{y}[3], \ldots\}$. Если сравнивать с языками программирования, то параметрический терм задает тело цикла, а область применимости с отношением порядка на нем – заголовок цикла.

В УВММ нет средств для завершения работы цикла. В ВММ циклы должны завершаться либо когда предусловие некоторой операции в теле цикла окажется ложным (цикл типа while), либо номер элемента массива превзойдет текущее значение счетчика (цикл типа for).

Операция ψ_{ab} задает алгоритм вычисления $out(\psi_{ab})$ из $in(\psi_{ab})$. Такие операции называются ψ-операциями [lit1]. Бесконечное множество термов T_V^W, построенного по ВММ (рисунок 1.5), можно заменить конечным множеством $T_{V'}^{W'}$, сформированным на основе простой вычислительной модели (рисунок 1.6). Построение плана тогда происходит как на обычной ПВМ, с той разницей, что требуется сформировать необходимые ψ-операции. Алгоритм планирования на ВММ (включая формирование ψ-операций) подробно приведен в [lit1].

Огромное число задач требуют использования массивов с размерностью больше одной. Поэтому следующим расширением вычислительных моделей будет логично ввести вычислительные модели с многомерными массивами. Вычислительная модель с многомерными массивами определяется аналогично ВММ с некоторыми необходимыми дополнениями:

- Каждому массиву $\bar{x} \in Y$ соответствует число $m_{\bar{x}}$ – число размерностей, и конечный набор счетчиков $n_{\bar{x}}^{1}$, $n_{\bar{x}}^{2}$, …, $n_{\bar{x}}^{m_{\bar{x}}}$.
- Каждой операции a соответствует число m_a – число счетчиков операции. При этом множество экземпляров операции a определяется набором (a, i_1, i_2, …, i_{m_a}), $i_k \in$ N, который обозначается $a^{i_1 i_2 \ldots i_{m_a}}$. Каждому счетчику $n_{\bar{x}}^{j}$, такому, что хотя бы один элемент массива $\bar{x}$ является параметром операции a, ставится в соответствие счетчик операции a. Разделение счетчиков переменных и операций стало необходимо потому, что возможно различное число счетчиков для операции и ее параметров. Например: in(a^i)={ $x[i][i]$ } – здесь $m_{\bar{x}}$=2, m_a =1.
- Для каждого экземпляра операции $a^{i_1 i_2 \ldots i_{m_a}}$ также вводится соответствующий предикат $p_a^{i_1 i_2 \ldots i_{m_a}}$ с теми же свойствами, что и в ВММ.
- Функция **Θ** также сопоставляет каждому набору ($a^{i \ldots j}$, $n_{\bar{x}}^{k}$) множество экземпляров операций, таких, что $n_{\bar{x}}^{k}$ является их выходной переменной, и только они могут использоваться для вычисления счетчика $n_{\bar{x}}^{k}$ для экземпляра $a^{i \ldots j}$.

Кроме того, необходимо уточнить направление вычисления значений счетчиков в термах: если какой-то терм вычисляет значения счетчиков $\{n_{\bar{x}}^{1}$, $n_{\bar{x}}^{2}$, …, $n_{\bar{x}}^{m_{\bar{x}}}\}$, то любой его подтерм может вычислять только значения $\{n_{\bar{x}}^{1}$', $n_{\bar{x}}^{2}$', …, $n_{\bar{x}}^{m_{\bar{x}}}$'$\}$, причем $\forall j \;\; n_{\bar{x}}^{j} \geq n_{\bar{x}}^{j}$', и хотя бы один $n_{\bar{x}}^{j} > n_{\bar{x}}^{j}$'.

Теперь есть возможность формирования ψ-операций и преобразования вычислительной модели с многомерными массивами в ПВМ. Алгоритм формирования ψ-операций на многомерных массивах является более сложным, чем алгоритм формирования ψ-операций на ВММ, и в настоящее время в литературе не обнаружен. Возможным способом решения этой проблему может быть явное задание пользователем ψ-операции, т.е. задание параметрического терма и области его применимости.

Далее везде под вычислительными моделями с массивами (ВММ) будем понимать именно вычислительные модели с многомерными массивами.

1.3 Постановка задачи

На кафедре ПВТ разрабатывается система конструирования параллельных программ на вычислительных моделях с массивами (далее просто «система»). Система должна будет осуществлять весь комплекс работ по сборке параллельных программ из готовых фрагментов, начиная от описания предметной области в виде совокупности вычислительных моделей с массивами и постановки прикладной задачи и заканчивая генерацией кода программы. Также задачей этой системы является накопление и хранение информации о предметных областях в библиотеке с целью последующего ее использования. Разрабатываемая система ориентирована на решение задач из области численных методов. Работу системы можно показать в виде схемы (рисунок 1.7).

Основными компонентами системы конструирования программ являются:

1. Транслятор с языка описания вычислительных моделей.
2. Планировщик алгоритма.
3. Генератор программы.

На вход системы поступает формулировка задачи, т.е. текстовое описание предметной области в виде совокупности вычислительных моделей и оператор постановки задачи. Транслятор проверяет синтаксическую правильность описания и переводит его во внутреннее представление, удобное для длительного хранения и дальнейшего использования (в частности, для обработки планировщиком). Задача планировщика – по внутреннему описанию вычислительной модели сформировать алгоритм решения поставленной задачи. Генератор на основе алгоритма, полученного от планировщика, формирует программу на некотором языке, содержащую

вызовы процедур из библиотеки программных модулей. Язык, используемый при формировании выходной программы, будем называть объектным.

Схема работы системы конструирования программ на вычислительных моделях

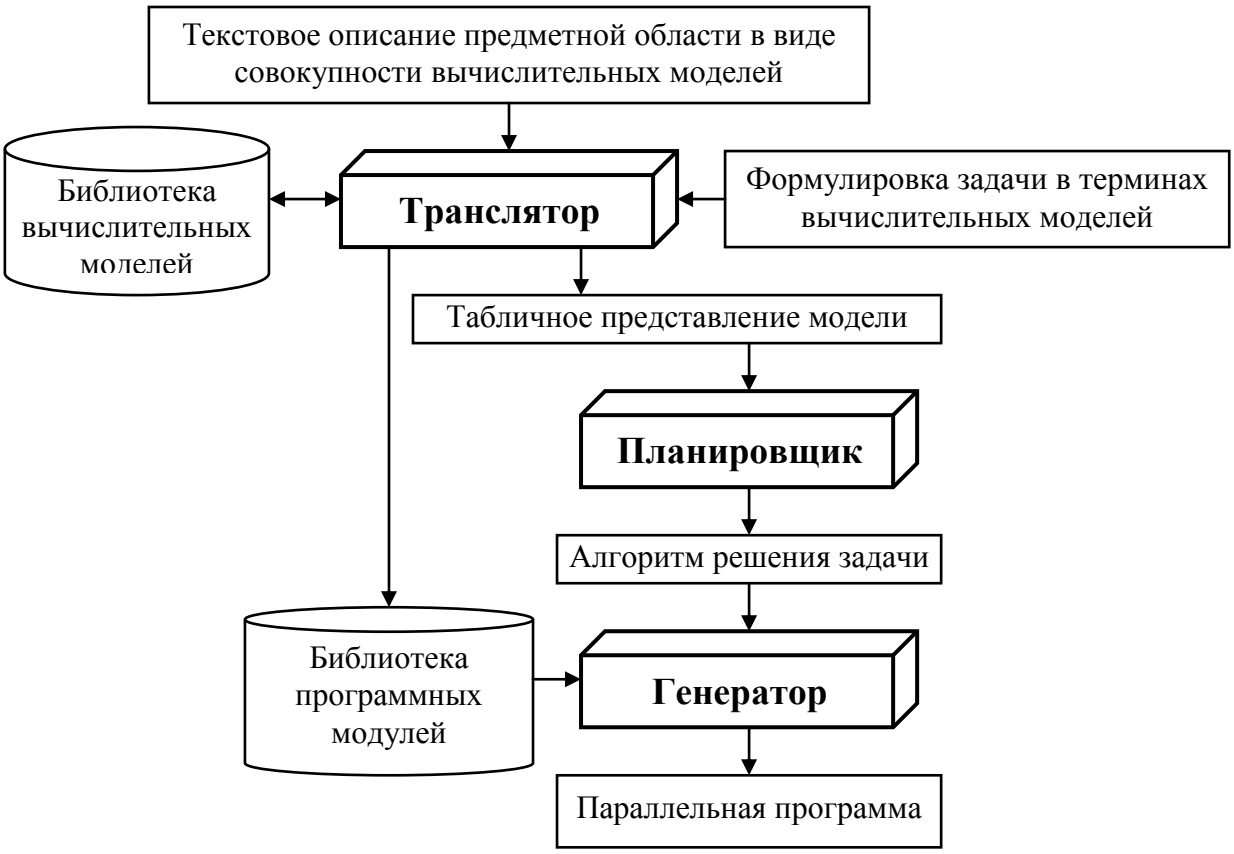


Рис. 1.7

Представление знаний, удобное для человека, как правило, не является подходящим для его непосредственной обработки на ЭВМ. Для удобства формирования описания в формальные языки обычно вводится масса дополнительных конструкций, механизмов, облегчающих использование языка. Но наличие таких конструкций требует дополнительной работы по приведению описания к форме, пригодной для обработки на компьютере. В частности, для работы планировщика наиболее подходящим является представление вычислительной модели в виде совокупности таблиц

переменных и операций, имеющих глобальную нумерацию внутри модели. Например, такие объекты, как структурированные переменные, должны быть раскрыты, т.е. заменены совокупностью обычных переменных, входящих в их состав.

Работа является частью проекта создания системы конструирования параллельных программ на вычислительных моделях. Она ставит перед собой следующие цели:

1. Разработка языка описания вычислительных моделей с массивами, обеспечивающего необходимый набор средств для описания модели и конструирования параллельных программ решения задач из области численного моделирования. Среди необходимых характеристик языка следует отметить:
   - наличие структурированных операций,
   - описание массовых вычислений.
   - наличие средств помощи системе конструирования в обработке задачи.
2. Разработка алгоритмов компиляции и реализация компилятора, проверяющего правильность текстового описания множества вычислительных моделей на разработанном языке и формирующего описание в табличном представлении, удобном для дальнейшей обработки и анализа планировщиком. Компилятор должен быть пошаговым и допускать возможность отката трансляции, что позволит облегчить процесс написания и отладки текстового описания предметной области.

1.4 Родственные работы, относящиеся к проблеме

1.4.1 Система ПРИЗ

В начале 70-х годов была разработана система ПРИЗ (см. [2], [3]), в которой использовался язык концептуального программирования Утопист. Концептуальным программированием называют такой подход к программированию, при котором ЭВМ применяется уже на этапе постановки задачи, а не только на этапе кодирования программы. Язык Утопист позволял задавать условия задачи в виде совокупности описаний абстрактных объектов, представленных в ЭВМ вычислительными моделями либо переменными, имеющими один из базовых типов. Отношения между переменными в формулировке условия задачи задавались уравнениями, из которых выводились правила вычисления переменных. Также имелась возможность задавать программные отношения, т.е. отношения, заданные во внешнем модуле или библиотеке макроопределений. Кроме того, язык Утопист имел большой набор управляющих структур, таких как указание порядка выполнения (последовательное, параллельное), циклы, условные предложения. Задача на языке Утопист формулируется следующим образом: формируется условие задачи в виде совокупности описаний абстрактных объектов и определяются два множества: множество входных переменных, значения которых заданы, и множество выходных переменных, значения которых необходимо вычислить. При формировании алгоритма решения задачи используется механизм логического вывода, а также реализован автоматический синтез программ.

Отличительной чертой системы Приз является ее ориентация на очень широкий круг задач. Будучи расширением операционной системы, она рассматривалась для использования в качестве системы построения трансляторов, системы построения пакетов программ (в частности СУБД), системы искусственного интеллекта и т.д. В то же время она плохо подходит для вычислительных задач, прежде всего потому, что в ее языке отсутствует

такой тип данных как массив. Есть возможность определить массив как расширение языка, но его эффективность будет заметно ниже. Также в языке отсутствует понятие структурированной операции. Решение задачи в системе ПРИЗ происходит в последовательном исполнении.

Существовали реализации системы Приз для различных типов ЭВМ (БЭСМ-6, СМ-4, 16-битных микроЭВМ, ЕС ЭВМ). В настоящее время система Приз не используется.

1.4.2 Т-система

Разработка Т-системы была начата в Институте программных систем РАН (г. Переславль-Залесский) в конце 80-х годов XX века [4]. Т-система предлагает подход к динамическому распараллеливанию программ основанный на использовании парадигмы функционального программирования и механизма параллельной редукции графов, который представляет собой один из самых эффективных подходов к реализации чисто функциональных программ.

Т-система ориентирована на класс задач с динамическим (скрытым до момента запуска программ) параллелизмом. Программа для Т-системы представляет собой набор чистых (без побочных эффектов) функций. Каждая функция может иметь несколько аргументов и несколько результатов. В то же время тела функций могут быть описаны в императивном стиле (на языках типа FORTRAN, Си и т. п.) Важно только, чтобы:

- всю информацию извне функция получала только через свои аргументы;
- вся информация из функции передавалась только через явно описанные результаты.

Явные параллельные конструкции в языке отсутствуют. Система в процессе выполнения программы сама заботится о распределении вычислений по имеющимся вычислительным ресурсам. Вызов функции G, производимый в процессе вычисления функции F выполняется

нетрадиционным способом (т.н. сетевой вызов функции). При этом порождается новый процесс с несколькими входами (в соответствии с числом аргументов функции G) и несколькими выходами (в соответствии с числом результатов функции G). Выходы нового процесса связываются с соответствующими переменными процесса F отношением "поставщик - потребитель", и тем самым, переменные-потребители принимают неготовые (не вычисленные) значения. Порожденный процесс G должен вычислить функцию G и заменить неготовые значения у всех своих потребителей на соответствующие результаты функции G.

Т-система предлагает подход к автоматическому распараллеливанию программ, но не предоставляет дополнительных средств описания задач, кроме расширения императивных языков типа языка С. Пользователь Т-системы должен быть хорошим программистом.

В настоящее время Т-система продолжает успешно развиваться [5].

1.4.3 Система Норма

Язык Норма (Непроцедурное Описание Разностных Моделей Алгоритмов) является специализированным непроцедурным языком, предназначенным для спецификации задач вычислительного характера, в частности, решения задач математической физики сеточными методами [4].

Основное назначение языка состоит в автоматизации процесса разработки программ. Входной язык – язык вычислительной математики. Язык Норма дает разработчику прикладных программ возможность абстрагироваться от особенностей конкретных компьютеров и сформулировать свою задачу в привычных для него терминах. Все конструкции языка носят декларативный характер и описывают правила вычисления значений. В записи программы не требуется никакой информации о порядке выполнения операций. Порядок предложений языка может быть произвольным. Язык позволяет формулировать запрос на вычисления, не уточняя, каким именно образом вычисления следует

организовывать. Все информационные связи выявляются и учитываются транслятором-синтезатором на этапах анализа исходной программы и синтеза выходного текста. На трансляторе лежит и выбор конкретного способа организации вычислений с учетом архитектуры ЭВМ (возможностей параллельной, векторной обработки и т.п.).

Язык Норма является языком с однократным присваиванием: каждая переменная в программе может принимать значение только один раз, а индексы переменных имеют вид $i \pm D$, $i \in N$, D – целая константа. Эти ограничения (и некоторые другие) связаны с необходимостью решать в процессе трансляции задачу синтеза выходной программы. В языке Норма отсутствует понятие памяти, а также многие обычные элементы традиционных языков программирования (управление, циклы, переприсваивание и т.п.).

Система Норма является примером использования существующих алгоритмов планирования и организации массовых вычислений в «хорошей» предметной области, т.е. в той, где они эффективно работают. Сейчас система успешно развивается. С ее помощью решаются многие задачи математической физики.

2. ЯЗЫК ОПИСАНИЯ ПРЕДМЕТНОЙ ОБЛАСТИ

2.1 Требования к языку

2.1.1 Описание простых вычислительных моделей

Необходимо разработать язык описания вычислительных моделей, обеспечивающий необходимый набор средств для описания задач из области численного моделирования. Результирующая программа должна собираться из готовых фрагментов вычислений, находящихся в библиотеке процедур или макроопределений.

Исходя из поставленной задачи, выделим моменты, которые должны быть отражены в языке. Прежде всего, разрабатываемый язык предназначен для описания объектов, называемых вычислительными моделями. Вычислительная модель есть совокупность двух неупорядоченных множеств переменных и операций. Следовательно, в основе это должен быть декларативный язык, позволяющий формировать структуру данных «вычислительная модель» как совокупность ее компонентов – переменных и операций – и отношений между ними. Так как предполагается наличие множества вычислительных моделей, необходимо ввести средство их идентификации, т.е. каждая вычислительная модель должна иметь уникальное имя. Так как в результате должна быть получена программа на некотором процедурном языке, целесообразно все непосредственные вычисления переменных описывать с использованием этого языка.

2.1.1.1 Переменные

Рассмотрим более подробно переменные модели. Они составляют неупорядоченное множество, относящееся к конкретной вычислительной модели. Как и практически во всех языках, переменная должна иметь имя и тип. Уникальность имени переменной достаточно ограничить вычислительной моделью, так как область действия любой операции ограничена переменными одной модели. Необходимость введения типов

определяется, прежде всего, тем, что переменные являются параметрами операций. Результирующая программа будет собираться из готовых фрагментов. Эти фрагменты будут написаны на каком-нибудь процедурном строго типизированном языке, например Си или Fortran, которые чаще всего используются для программирования вычислительных задач. Поэтому для корректности вызовов или подстановки этих фрагментов следует хранить информацию о типе переменных. Таким образом, набор типов переменных разрабатываемого языка должен соответствовать набору типов объектного языка.

Вычислительная модель является неинтерпретированным объектом, она только отражает структуру вычислений, которая может соответствовать различным предметным областям. Логичным будет предоставить пользователю возможность отложить описание интерпретации модели и описать только структуру вычислений. Это потребует введения в язык переменных неопределенного типа, т.е. таких, для которых интерпретация будет указана позднее.

2.1.1.2 Использование подмоделей

При описании сложных объектов и систем «плоское» представление вычислительной модели в виде множества одинаковых переменных и операций оказывается крайне неудобным. Как правило, в таких случаях используют метод «от простого к сложному», т.е. сперва формируется описание простых объекты, а затем на их основе – более сложных. Такой способ описания затрудняет его обработку на компьютере, но необходим пользователю, так как естественным образом отражает общепринятые представления об устройстве окружающего мира.

Введение в модель возможности включения готовых вычислительных моделей как части описания новых более высокого уровня позволяет строить описания сложных объектов постепенно. При этом все содержимое вычислительной модели более низкого уровня копируется в новую модель.

Такую вычислительную модель, которая является частью вычислительной модели более высокого уровня, будем называть подмоделью. Например, вычислительная модель, описывающая треугольник, может включать в себя три различных подмодели «Отрезок», каждый из которых имеет свой набор характеристик, соответствующих отрезку.

Введение подмоделей заставляет расширить понятие «имя переменной», чтобы избежать дублирования имен при включении в модель нескольких одинаковых подмоделей. Сопоставим каждой подмодели уникальное имя – идентификатор. Имя переменной, находящейся в некоторой подмодели, будет состоять из имени этой подмодели и имени переменной внутри этой подмодели. Такое составное имя переменной позволит избежать коллизии.

В процессе описания вычислительной модели нередкой бывает ситуация, когда несколько переменных должны иметь одно и то же значение. Она, как правило, проявляется при сборке модели из подмоделей. Пусть, например, имеем описание вычислительной модели «Точка на плоскости», представленная своими координатами (рисунок 2.1). С помощью нее можно построить вычислительную модель «Отрезок».

Вычислительные модели «Точка» и «Отрезок»

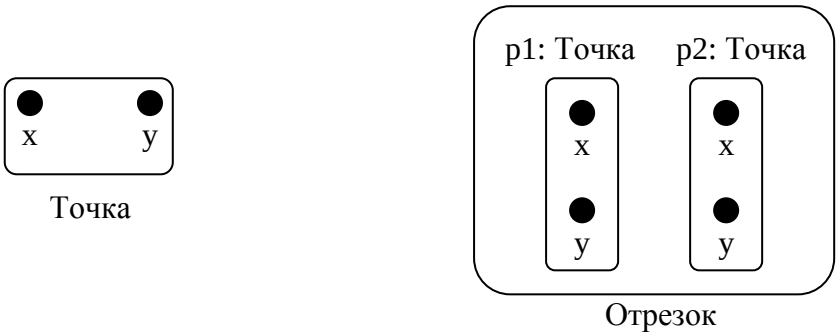
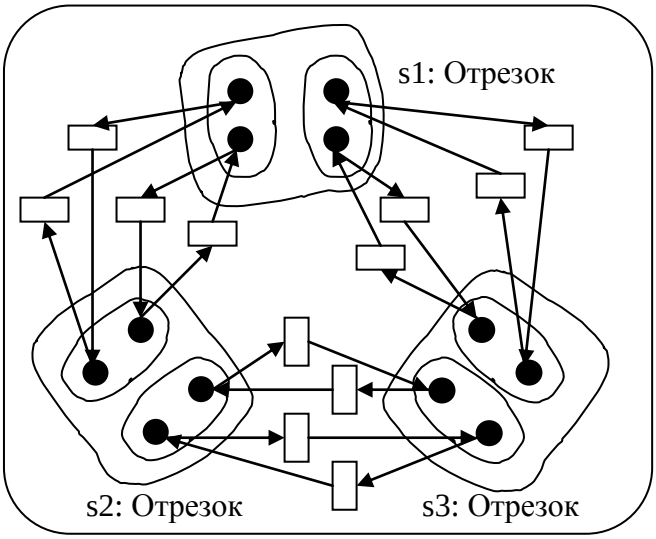


Рис. 2.1

Строим теперь вычислительную модель «Треугольник» – собираем ее из трех отрезков s1, s2 и s3. Чтобы обеспечить равенство компонентов точек соседних отрезков, можно вести соответствующие операции в двух направлениях, вычисляющие компоненты точки одного отрезка из компонент точки другого отрезка (рисунок 2.2).

Структура вычислительной модели «Треугольник»



Треугольник

Рис. 2.2

Если несколько все компоненты подмоделей «Треугольник» в составе какой-то большей модели должны быть равны между собой, то число операций еще возрастет. Причем, этим мы только описываем структуру объектов, а не связанные с ним вычисления. Чтобы упростить описание таких ситуаций, введем специальное отношение равенства объектов. Более того, нет необходимости иметь несколько разных переменных с одним значением – каждый набор равных переменных можно представить одной переменной. Такая операция, при которой набору переменных соответствуют тождественно равные значения и одна область памяти в выходной программе, будем называть унификацией.

Как один из вариантов унификации можно рассмотреть унификацию переменных с константой. Тогда мы можем, например, описать модель,

которая определяет множество всех треугольников с вершиной в начале координат (все координаты одной из точек равны нулю).

Естественным будет расширение операции унификации и на эквивалентные по своей структуре подмодели. При этом будем считать, что унифицируются все переменные соответствующих подмоделей с одинаковыми именами. Тогда та же вычислительная модель «Треугольник» примет более простой вид (рисунок 2.3):

Структура вычислительной модели «Треугольник» с использованием унификации

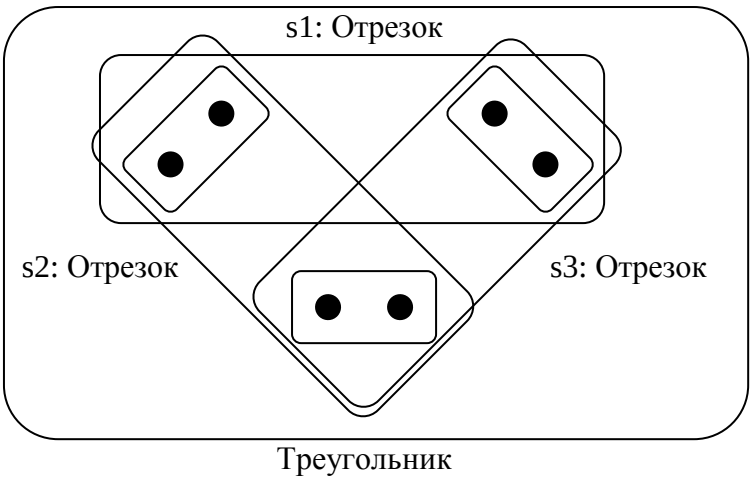


Рис. 2.3

Наличие подмоделей позволяет использовать готовые описания моделей для создания новых. Довольно часто встречается ситуация, когда необходимо описать новый объект, несколько изменив уже существующий. Но после нескольких таких шагов получится, что модель содержит длинный ряд последовательно вложенных подмоделей, что может быть не очень удобно для ее использования и обработки. Эту ситуацию можно улучшить введением специального механизма, который назовем «упрощенным наследованием»: при этом все содержимое наследуемой вычислительной модели будем просто копировать в новую модель. Тем не менее, не будем допускать множественного наследования, как это реализуется в некоторых

объектно-ориентированных языках, чтобы избежать возможных коллизий имен компонентов модели.

2.1.1.2 Операции

Рассмотрим операции модели. Пусть каждой операции соответствует имя, уникальное в пределах вычислительной модели – оно позволит идентифицировать конкретную операцию.

Прежде всего, операция связывает два множества переменных – входные и выходные параметры операции. С точки зрения описания модели операция с пустым множеством выходных переменных не имеет смысла, так как ничего не вычисляет. Но на практике такой вид имеет, например, операция вывода, так же, как операция с пустым множеством входных переменных может быть операцией ввода. Отсюда, ограничением на множества входных и выходных параметров отдельной операции будет только то, что их объединение должно быть не пусто. Кроме того, можно еще рассмотреть пересечение этих множеств, потому как возникает естественный вопрос о целесообразности вычисления того, что уже вычислено. Любое перевычисление переменной повышает надежность ее значения. Так как вычислительная модель организована таким образом, что допускает избыточность вычислений (в этом одно из ее преимуществ), она может использоваться для организации вычислений с различной степенью надежности. Поскольку в работе такая цель не ставится, то разрешать повторное вычисление сейчас не имеет смысла. Тем не менее, повторное вычисление не оказывает никакого отрицательного влияния на решение задачи в целом (при планировании такие операции могут просто не включаться в алгоритм), поэтому оставим эту возможность во имя дальнейшего расширения системы. В частности, будем допускать пересечение входных и выходных параметров операции.

Операция показывает возможность выполнения некоторого вычисления. Она должна быть связана с конкретным объектом, который

реализует эту операцию. В простейшем случае ее реализацией является фрагмент программы на процедурном языке, который будет подставлен в результирующую программу. Этот фрагмент, возможно, содержит свои переменные, участвующие в вычислении. Часть переменных являются для фрагмента входными, а часть выходными. Нужен механизм сопоставления параметров операции (переменных модели) и переменных фрагмента программы.

Более сложным случаем реализации операции является процедура во внешнем объектном модуле. В этом случае требуется сформировать корректное обращение к модулю и правильный вызов нужной процедуры. Для правильного вызова, возможно, понадобится выполнить определенные действия, например, сформировать специальную структуру данных до вызова и уничтожить ее после вызова. Причем, эти действия должны быть запрограммированы с помощью какого-то процедурного языка. Как и в предыдущем случае, получаем фрагмент программы на процедурном языке, с теми же требованиями по организации. Отличием является то, что появляется необходимость указать модуль, в котором содержится вызываемая процедура. В общем случае это даже множество модулей, так как один фрагмент вычислений может использовать много разных внешних процедур.

Еще одним вариантом реализации операции является постановка задачи. Это так называемая структурированная операция, рассмотренная в первой главе. Постановка задачи включает указание вычислительной модели и множеств входных и выходных переменных. Для описания структурированной операции в модели необходим механизм установления соответствия между переменными этой модели – параметрами операции и переменными модели, на которой ставится задача.

Пользователь также может отложить интерпретацию операции, и не указывать ее реализацию сразу после описания.

Введение условных операций в язык требует возможности сопоставить каждой операции некоторое предусловие – логическое выражение, содержащее переменные модели. Включение в язык логических выражений автоматически приводит к необходимости включения арифметических выражений. Далее здесь возможны два пути. Первый – это следовать утверждению, что все вычисления в модели должны быть скрыты внутри операций, и ввести специальные логические операции, которые на входе имеют список переменных модели, а на выходе – логическое значение. Второй – это разрешить прямо в предусловии использовать элементарные арифметические и логические операции над переменными.

Первый подход означает, что для переменных x, y, …, z будут вводиться дополнительные операции, имеющие реализацию вида: x>0, y>0, …, z>0. Такое множество дополнительных операций, имеющих косвенное отношение к алгоритму, ясности модели не прибавит. Кроме того, уже на этапе постановки задачи значения некоторых переменных могут быть известны, поэтому желательно иметь возможность распознавать выражение, чтобы при планировании вычислить все предусловия, какие возможно, и отбросить заведомо недопустимые пути вычислений. Итак, было решено выбрать второй вариант – явное задание арифметико-логических выражений в предусловии.

На этом пути существует другая проблема: ранее было решено, что все вычисления должны быть записаны с использованием объектного языка. Например, запись логических операций на языке Fortran отличаются от их записи на языке Си. Желательно, чтобы описание модели не было привязано к конкретному языку. Тогда было принято решение ввести обозначение элементарных арифметико-логических операций, характерное только для языка описания моделей. При трансляции такое описание может быть распознано, при планировании – частично вычислено после подстановки

известных значений переменных, а при генерации – сформировано логическое выражение на требуемом объектном языке.

2.1.2 Описание вычислительных моделей с массивами

С появлением в модели индексированных компонентов появляется необходимость введения дополнительных средств для работы с ними. Индексированный объект определяется не только именем, но и значением индекса (или набора индексов при многомерной индексации).

Прежде всего, при работе с индексированными объектами может потребоваться указание множества значений индексов. Как правило, в численных методах они имеют регулярную структуру, т.е. их можно задать путем указания правила перебора. Перебор является последовательной процедурой, но это только способ описания множества независимых значений, и он не ограничивает их каким-либо порядком. Тем не менее, можно оставить в языке возможность явного указания, необходимо ли всегда следовать заданному порядку перебора индексов.

Для выделения отдельных элементов множества значений индексов необходимо ввести в модель специальные переменные – счетчики. В отличие от обычных переменных модели счетчики могут принимать любые значения из заданного множества значений. При описании модели может потребоваться указание объектов с индексами, зависящими от значений счетчиков или даже переменных модели, поэтому логичным будет допустить использование арифметических выражений в качестве значений индексов. Естественным ограничением на индексные арифметические выражения будет разрешение использовать в них только счетчики, целочисленные переменные модели и целочисленные константы.

Как известно из первой части работы, планирование на вычислительных моделях с массивами происходит следующим образом: на основе индексированных операций, задающих массовое вычисление, формируются ψ-операции, после чего выполняется планирование на простой

вычислительной модели. Входные и выходные параметры ψ-операций могут содержать не только простые переменные и отдельные элементы массивов, но и множества элементов, например, «все нечетные начиная с 5-го», так как они используются либо являются результатом массового вычисления.

Ранее было упомянуто, что пока нет алгоритма формирования ψ-операций на многомерных массивах. Даже когда он будет разработан, автоматически сформированная ψ-операция может не удовлетворять пользователя с точки зрения эффективности. Поэтому необходима возможность самостоятельно формировать ψ-операцию путем явного указания лежащего в ее основе параметрического терма и области применимости счетчиков – параметров терма. На основе ψ-операции должен быть сформирован цикл, после выполнения которого будут вычислены переменные модели, указанные в качестве выходных. Множество значений счетчика задается правилом перебора индексов – на его основе будет сформирован заголовок цикла. Параметрический терм представляет собой совокупность экземпляров массовых операций с конкретными значениями индексов, которые должны быть выполнены на каждой итерации цикла. Порядок выполнения экземпляров операций в общем случае определяется зависимостями по данным. Параметрический терм представляет собой простую вычислительную модель, собранную из указанных экземпляров операций и переменных, являющихся для них входными или выходными параметрами. На этой модели также необходимо провести планирование.

2.1.3 Средства помощи планированию и генерации

При планировании и генерации система конструирования параллельных программ должна решать проблему нахождения наиболее оптимального алгоритма решения поставленной задачи для данной вычислительной системы и распределения данных и вычислений по имеющимся ресурсам.

Для помощи системе в выборе пути при процессе планировании можно ввести в язык понятие «свойство операции» – некоторый дополнительный параметр, характеризующий операцию. Пользователь при описании модели определяет свойства каждой операции, а при описании задачи накладывает ограничения на эти свойства, в результате чего планировщик имеет возможность сделать выбор.

Распределение данных в памяти компьютера является задачей генератора. В случае простых вычислительных моделей здесь нет проблемы: простые переменные занимают немного места в памяти, их распределением относительно легко управлять. В вычислительных моделях с массивами ситуация усложняется: один массив может занять всю имеющуюся память, или даже превысить ее размер. Кроме того, так как вычислительная модель может определять потенциально бесконечные итерационные вычисления, требуется возможность использования массивов с потенциально бесконечным множеством элементов. Возникает проблема отображения части массива на физическую память. Причем, это отображение может меняться с ходом вычислений. Существуют алгоритмы автоматического решения этой задачи, но, опять же, автоматически полученное решение может не удовлетворять пользователя. Поэтому требуются средства явного задания правил отображения логического массива на физическую память.

2.2 Ограничения, принятые в реализации языка

В процессе работы системы описание вычислительной модели подвергается различного рода анализу и обработке. Возможности алгоритмов обработки зачастую имеют ограничения, которые необходимо учесть при разработке языка описания. Основная часть времени работы вычислительной программы приходится на массовую обработку данных, поэтому именно ее стремятся как можно сильнее ускорить, применяя различные алгоритмы оптимизации. Отсюда, большинство ограничений, как правило, связано с описанием массовых вычислений.

Самым элементарным фрагментом массового вычисления на вычислительных моделях с массивами является ψ-операция. В результирующей программе ей в простейшем случае будет соответствовать некоторый цикл. Сложность описания ψ-операции определяет сложность всего алгоритма. ψ-операция задается:

- параметрическим термом с параметрами-счетчиками,
- областью применимости – множеством допустимых значений параметров-счетчиков.

Параметрический терм описывает атомарное вычисление, проводимое как часть массового. Максимальная разность значений индексов, которая имеется в данном измерении у элементов массива, используемых в параметрическом терме, определяет число элементов массива по данному измерению, которые должны одновременно присутствовать в памяти. Естественно, что чем меньше эта разность, тем лучше, т.к. меньше памяти требуется для выполнения терма на каждом шаге итерации. Для планирования массовых вычислений необходима возможность проведение различных преобразований индексных выражений (в том числе оценка указанной разности), поэтому имеет смысл сократить свободу описания индексных выражений до аддитивных, т.е. использующих операции сложения и вычитания.

Область применимости ψ-операции определяется множеством индексов. Именно оно охватывает все массовое вычисление в целом. Кроме того, множества индексов могут встречаться не только в ψ-операциях, но и в других элементах описания модели. Все конструкции, определяющие множества индексов, должны быть таковы, что по ним могут быть автоматически построены процедуры, перечисляющие элементы описанных множеств. Итак, множество значений индексов будем определять с помощью следующих параметров:

- начальное значение,

- шаг изменения,
- конечное значение,
- тип перебора: последовательный или произвольный.

Первые три параметра являются целочисленными значениями, и в общем случае могут быть представлены арифметическими выражениями, содержащими константы, счетчики и переменные модели. Тип перебора необходим для обеспечения большей непроцедурности описания, т.е. чтобы не ограничиваться рамками строго заданного порядка перебора там, где это не необходимо.

Основные операции, выполняемые над множествами индексов при планировании – это проверка вложенности одного множества в другое и объединение двух множеств. Эти операции могут быть выполнены над двумя множествами, только если их соответствующие параметры отличаются на константу. Чтобы можно было выполнить эти операции для любых двух множеств индексов из описания, к моменту планирования все множества должны иметь константные параметры (конечное значение может быть неограниченным). Т.е. не должно оставаться параметрического множества индексов, зависящего от какого-либо счетчика или переменной.

В настоящей реализации системы в качестве объектного языка был выбран язык Си. В связи с этим, синтаксические конструкции, которые далее вводятся для обеспечения формирования корректной программы в результате, учитывают только использование языка Си. При расширении системы нетрудно будет дополнить язык описания конструкциями, учитывающими особенности формирования программы на другом объектном языке.

2.3 Описание языка

2.3.1 Синтаксис и семантика основных конструкций языка

С учетом рассмотренных выше требований был разработан язык описания вычислительных моделей с массивами. В этом параграфе будут представлены основные конструкции языка описания простых вычислительных моделей. Описание и использование массивов и подробно рассматривается в следующем параграфе.

2.3.1.1 Лексика и общие правила описания

На лексическом уровне в языке различаются ключевые слова, идентификаторы, константы и строки спецсимволов. Правила формирования идентификаторов соответствуют правилам в большинстве распространенных языков программирования: идентификатор состоит из букв, цифр и символов подчеркивания, первый символ – буква. Константы в языке различаются целые, вещественные, целые шестнадцатеричные и строковые. Правила записи констант аналогичны правилам языка Си. Строки спецсимволов – специальные последовательности символов, используемые в языке, например, символы арифметических операций, скобки, многоточие и т.п. К символам-разделителям относятся пробел, табуляция и перевод строки. В языке не имеет значения способ распределения лексем по стокам и число символов-разделителей между ними.

Одно текстовое описание может располагаться в разных файлах. Включение текста из файла в текущее описание осуществляется с помощью директивы:

```
#include "имя файла"
```

Директива должна располагаться на отдельной строке без предшествующих символов-разделителей.

Правила записи комментариев аналогичны правилам языка C++, например:

```
integer x; // комментарий до конца строки
real /* комментарий произвольной длины */ y;
```

Разработанный язык описания вычислительных моделей является декларативным, он содержит средства для описания набора объектов, их компонентов и отношений между ними. Текст описания рассматривается последовательно, что накладывает некоторые ограничения на порядок следования элементов описания. Работу с любым объектом в языке можно разделить на две части: описание объекта и использование этого объекта в описаниях других объектов и отношений. Главным ограничением на порядок следования конструкций является то, что любой объект может быть использован только после его описания. Порядок следования конструкций языка также может быть важен для некоторых специальных объектов, как, например, переменная неопределенного типа, информация о которых дополняется по мере их использования в различных конструкциях.

В представленных ниже примерах описания объектов и отношений жирным шрифтом выделены ключевые слова языка, в угловых скобках указаны различные синтаксические понятия языка. LL1-грамматика разработанного языка представлена в приложении 1.

2.3.1.2 Вычислительные модели и переменные модели

Основной синтаксической единицей языка является описание вычислительной модели, формирующей описание некоторого объекта. В простейшем случае оно выглядит следующим образом:

```
model <имя модели>
{
  <тело модели>
};
```

Здесь «имя модели» есть идентификатор, а «тело модели» содержит описания всех ее компонентов и отношений. В том случае, если требуется выполнить «упрощенное наследование», т.е. скопировать в новую модель все

компоненты и отношения ранее описанной модели, то описание будет иметь следующий вид:

```
model <имя модели>: <модель>
{
  <тело модели>
};
```

Здесь «модель» – имя ранее описанной модели. Вычислительная модель формируется один раз, и после завершения описания не может быть изменена.

Переменные модели задают разного рада характеристики описываемого объекта. Каждая переменная должна быть описана один раз в своей модели. Основными характеристиками переменой являются ее имя и тип. Описание переменных выглядит следующим образом:

```
<тип> <список имен переменных>;
```

«Список имен переменных» – это список идентификаторов, перечисленных через запятую. «Тип» – синтаксическая конструкция, задающая один из возможных типов переменных. В языке различаются:

- базовые типы – типы объектного языка,
- неопределенный тип.

Базовые типы, должны иметь аналоги в объектном языке. К ним относятся следующие типы: integer – целочисленный и real – вещественный. Для программирования большинства вычислительных задач этих двух типов вполне достаточно. При дальнейшем развитии языка можно предусмотреть использование множества стандартных типов данных, используемых в процедурных языках программирования. В данный момент считается, что тип integer соответствует типу long языка Си, а тип real – типу double языка Си.

Включение подмоделей в описание модели будем производить аналогично описанию переменных. При этом вместо типа указывается имя

включаемой модели. Для указания переменных и в составе подмоделей будем использовать составные имена, элементы которых отделены символом «.» (точка). Для указания подмоделей в составе других подмоделей также будем использовать составные имена.

С введением подмоделей появляется дополнительная классификация переменных. Если вычислительная модель содержит описание некоторой переменной, то такую переменную будем называть собственной переменной данной модели. Если переменная была включена как часть подмодели, то она называется несобственной для данной модели.

Например:

```
model point
{ real x,y;
};
model sector
{ point p1,p2;
  real length;
};
model triangle
{ sector s1,s2,s3;
  real S;
};
```

Модель point:

собственные переменные: x, y.

Модель sector:

собственные переменные: length;

несобственные переменные: p1.x, p1.y, p2.x, p2.y.

Модель sector:

собственные переменные: S;

несобственные переменные: s1.p1.x, s1.p1.y, s1.p2.x, s1.p2.y, s1.length,
s2.p1.x, s2.p1.y, s2.p2.x, s2.p2.y, s2.length,

s3.p1.x, s3.p1.y, s3.p2.x, s3.p2.y, s3.length.

Неопределенный тип указывается с помощью ключевого слова **undefined**. Сразу после объявления переменная неопределенного типа может использоваться в любых конструкциях языка. После конкретизации такая переменная может как оказаться переменной базового типа, так и подмоделью. Многие конструкции накладывают ограничение на интерпретацию переменных, которые используются в них. После использования неопределенной переменной в такой конструкции, информация об ограничении на интерпретацию запоминается и учитывается при последующем ее использовании. Рассмотрим следующий пример:

```
model object0
{ integer n;
  real s;
};
model object1
{ undefined x;  // объявление переменной неопределенного типа
  integer m;  // объявление переменной типа integer
  object0 y;  // объявление подмодели
  x.s=m;      // создание компоненты s типа integer
  x=y;        // изменение типа на object0
};
```

В примере используется новая конструкция, обозначаемая символом «=» (равенство). Она обозначает унификацию переменных либо подмоделей, т.е. их полное отождествление. Например, после конструкции «x.s=m;» обращение к переменной m эквивалентно обращению к переменной x.s. Понятно, что в этом случае их типы должны совпадать. Упоминание в модели переменной x.s говорит о том, что x является подмоделью, и что она имеет компоненту s типа integer. С этого места любое имени x как переменной базового типа или имени x.s как переменной типа, отличного от

integer, является ошибочным. Таким образом, постепенно формируется шаблон вычислительной модели, которой является (или может являться) x. При попытке конкретизации такого неопределенного объекта проверяется соответствие допустимость этой операции:

- для подмодели – соответствие компонентов по именам и типам и отсутствие лишних компонентов (которых нет в соответствующей вычислительной модели);
- для базового типа – отсутствие компонентов.

Если сопоставляются две переменные неопределенного типа (например, при унификации), то проверяется соответствие их шаблонов, после чего формируется общий шаблон как объединение шаблонов переменных. При проверке соответствия шаблонов проверяется соответствие типов переменных с одинаковыми именами.

Использование неопределенной переменной в каком-либо арифметико-логическом выражении указывает на то, что она должна быть базового типа. Использование неопределенной переменной в индексном выражении (при описании массовой обработки данных) указывает на то, что она имеет тип integer.

Следующая параметрическая конструкция позволяет объявлять объекты такого же типа (переменная базового типа, неопределенная переменная, подмодель), как и объект в параметре:

```
typeof(<имя>) <список имен>;
```

Здесь «имя» – это полное имя любой ранее описанного объекта (переменной или подмодели). Так как для переменных неопределенного типа шаблон ограничений зависит от конкретного положения в последовательности описаний внутри модели, то использование данной синтаксической конструкции с неопределенной переменной в параметре будет приводить к различным результатам при расположении ее в различных частях последовательности описаний. Переменные, объявленные таким

образом, будут иметь тот тип (со всеми ограничениями), который имела переменная в параметре в момент этого объявления.

2.3.1.3 Описание операций

Стандартное описание операции выглядит следующим образом:

```
operation <имя операции>={ <реализация> }
  in <список входных переменных>
  out <список выходных переменных>;
```

Здесь «имя операции» – идентификатор, «реализация» – один из возможных вариантов реализации операции, «список входных переменных» и «список выходных переменных» – списки имен переменных (через запятую), которые являются входными или выходными для данной операции. Список входных переменных или список выходных переменных может отсутствовать, тогда не указывается также и соответствующее ключевое слово **in** или **out**.

Если операция является условной, то ее описание выглядит следующим образом:

```
operation <имя операции>=if (<предусловие>)
  { <реализация> }
  in <список входных переменных>
  out <список выходных переменных>;
```

«Предусловие» есть арифметико-логическое выражение, содержащее переменные модели и числовые константы. В нем могут использоваться все стандартные арифметико-логические операции языка Си.

Возможны три вида реализации операции:

- вызов макроопределения,
- постановка задачи,
- ψ-операция.

Описание ψ-операций рассмотрим далее при анализе описания моделей с массивами. Постановка задачи в простейшем случае выглядит следующим образом:

```
compute on <имя модели>
  in <список входных переменных задачи>
  out <список выходных переменных задачи>
```

Здесь «имя модели» есть имя той модели, на которой формулируется задача, «список входных переменных задачи» и «список выходных переменных задачи» – списки имен переменных в этой модели (через запятую). Переменные из списка входных переменных задачи должны по количеству и типам соответствовать переменным из списка входных переменных операции. Соответствие между конкретными переменными устанавливается по их положению в списке. Такое же правило должно выполняться и для списков выходных переменных.

Для постановки задачи есть возможность не использовать готовую вычислительную модель, а сформировать новую. В этом случае используется следующая синтаксическая конструкция:

```
compute on model { <тело модели> }
  in <список входных переменных задачи>
  out <список выходных переменных задачи>
```

«Тело модели» здесь, как и раньше, множество описаний компонентов и отношений. Сформированная таким образом модель не имеет имени и не может использоваться для объявления подмоделей или формулировки других задач, кроме текущей. При необходимости для формирования новой модели можно выполнить «упрощенное наследование»:

```
compute on model: <модель> { <тело модели> }
  in <список входных переменных задачи>
  out <список выходных переменных задачи>
```

Здесь «модель», как и ранее, имя готовой модели, все содержимое которой копируется в новую.

Реализация операции в виде вызова макроопределения имеет следующее описание:

```
mcall <имя макроопределения>(<список параметров>);
```

Правила описания макроопределений будут рассмотрены чуть позже. Здесь же стоит упомянуть, что в данном языке макроопределение определяется уникальным именем (идентификатор) и упорядоченным списком типизированных параметров. Поэтому при описании вызова макроопределения указывается «имя макроопределения» (идентификатор) и «список параметров», содержащий переменные модели и константы, разделенные запятыми. Типы элементов списка фактических параметров должны соответствовать типам формальных параметров макроопределения, заданных при его описании. Переменные, указанные при вызове макроопределения, обязательно должны входить в список входных или выходных переменных описываемой операции.

Одним из требований к языку ранее указывалась возможность отсутствия привязанности модели к конкретной интерпретации. Для переменных это достигается с помощью введения неопределенного типа. Для операций также была введена возможность не указывать реализацию при описании. Соответствующая синтаксическая конструкция выглядит следующим образом:

```
operation <имя операции>
  in <список входных переменных>
  out <список выходных переменных>;
```

или для условных операций:

```
operation <имя операции>=if (<предусловие>)
  in <список входных переменных>
  out <список выходных переменных>;
```

Чтобы такие операции имели практическую значимость, необходимо на некотором этапе описания доопределить их реализацию. Так как уже описанную модель мы не имеем права изменять, единственная возможность – это скопировать ее содержимое в новую модель с помощью упрощенного наследования, и доопределить ее как собственность уже другой модели. Доопределить – значит описать заново с тем же именем и параметрами, но

уже с указанием реализации. Если модель, имеющая операции без реализации, используется для объявления подмоделей в других моделях или для постановки задачи, то такие недоопределенные операции не учитываются при построении программы, так как они фактически не могут реализовать вычисление, которое описывают.

Возможность учета такого предварительного описания операций без реализации полностью зависит от планировщика. Ему таким образом предоставляется возможность провести планирование на одной модели, которая затем используется как основа для формирования множества других с различными реализациями.

2.3.1.3 Описание макроопределений

Макроопределения задают фрагмент вычислений на процедурном языке. В качестве объектного языка системы был выбран язык Си, поэтому он используется для задания фрагмента вычислений. Описание макроопределения должно располагаться за пределами описания какой-либо вычислительной модели. Синтаксическая конструкция, описывающее макроопределение, выглядит следующим образом:

```
macro <имя>(<список параметров>)
{ <фрагмент программы на языке Си> }
```

«Имя» – это идентификатор, задающий уникальное имя макроопределения. «Список параметров» – последовательность объявлений переменных (через запятую) следующего вида:

```
<тип> <имя>
```

Здесь «тип» – один из базовых типов (integer или real), «имя» – идентификатор. «Фрагмент программы на языке Си» есть последовательность операторов в языке Си, использующих переменные, объявленные в списке параметров. Тип переменных real в этом фрагменте соответствует типу языка Си double, а тип integer – типу long. Так как вместо имен переменных – параметров при генерации результирующей программы

могут быть подставлены другие имена, в тексте макроопределения они обозначены предшествующим символом ‘$’, который не используется в языке Си.

Фрагмент программы может содержать различные вызовы функций из внешних библиотек или объектных модулей. Используемые библиотеки должны быть указаны при описании макроопределения следующим образом:

```
macro <имя>(<список параметров>):<список библиотек>
{ <фрагмент программы на языке Си> }
```

Все библиотеки из списка библиотек должны быть описаны ранее с помощью конструкции:

```
module <тип> “<имя файла>” <имя библиотеки>;
```

Здесь «тип» – это один из следующих типов библиотеки:

- C – стандартная библиотека языка Си,
- C_USER – пользовательская библиотека – набор функций на языке Си,
- OBJ – библиотека функций в объектном модуле.

«Имя файла» – имя файла с библиотекой, «имя библиотеки» – идентификатор, по которому данная библиотека может быть идентифицирована в дальнейшем описании. Именно эти имена перечисляются через запятую в конструкции «список библиотек» при описании макроопределения.

2.3.2 Описание моделей с массивами

Массивы вводятся в языке как расширение обычных типов данных. Массив – это индексированное множество переменных, возможно, бесконечное. Массив определяется именем, типом элементов, числом измерений и множеством допустимых значений индексов по каждому измерению. Все эти параметры задаются при описании массива. Например:

```
real x[10][2:9];
```

В данном примере описан двумерный массив с элементами типа real и допустимыми значениями индексов от 0 до 9 по первому измерению и от 2

до 9 по второму измерению. Число измерений массива формально не ограничивается (разве что реализацией компилятора).

В языке также используется понятие массива подмоделей. При этом считается, что объявление массива подмоделей эквивалентно введению дополнительной индексации для всех переменных, входящих в подмодель.

С появлением массовой обработки в языке появляются такие понятия, как счетчики и множества индексов. Счетчик – это специальная целочисленная переменная. В отличие от остальных переменных модели счетчику может соответствовать множество значений. Каждый счетчик имеет имя – идентификатор.

Множество значений индексов – это определенным образом заданное множество целых чисел, которое используется в некоторых конструкциях. Для задания множества значений индексов может использоваться одна из следующих синтаксических конструкций:

```
<начальное значение>:<конечное значение>
<начальное значение>...<конечное значение>
<начальное значение>(<шаг>):<конечное значение>
<начальное значение>(<шаг>)...<конечное значение>
```

Здесь двоеточие указывает на независимость порядка перебора элементов множества, а многоточие устанавливает строгий порядок перебора: последовательно от начального до конечного значения. В различных частях описания модели также бывает необходимо объявить счетчики, принимающие значения из заданного множества индексов. Соответствующая конструкция (назовем ее «объявление счетчиков») выглядит следующим образом:

```
for <множество счетчиков>
```

где «множество счетчиков» – это множество элементов следующего вида (разделенных запятыми):

```
<имя счетчика>=<множество значений индексов>
```

«Имя счетчика» есть идентификатор, «множество индексов» было введено ранее.

Для выполнения массовых вычислений необходимо наличие массовых операций. Рассмотрим пример описания массовой операции:

```
operation a[i][j]={ mcall assign(x[i][j],y[i][j]); }
  in x[i][j] out y[i][j];
```

Здесь отличие от описания обычной операции – это указание счетчиков-параметров массовой операции. Естественно, что параметры – переменные должны зависеть от этих счетчиков. Полное имя переменной теперь может содержать индексные (аддитивные) выражения. Кроме того, так как входными и выходными параметрами ψ-операции могут являться множества элементов массива, то полное имя переменой может содержать и множества индексов. Таким образом, в описании допускаются даже конструкции, подобные этой:

```
x[0][i].y[0:n][1(2):*]
```

Аналогично формируется и имя подмоделей. Если в качестве значения индекса указывается символ ‘*’ (звездочка), то считается, что указано множество значений индексов, заданное при описании соответствующей размерности массива.

На счетчики-параметры массовой операции также может быть наложено ограничение в виде заданного множества индексов, например:

```
<объявление счетчика i>
  operation a[i]={ <реализация> } <параметры>;
```

Теоретически недопустимо унифицировать массовые переменные с чем бы то ни было, так как чаще всего мы не имеем той гибкости в размещении массива в памяти, которая может потребоваться в результате унификации. Но конструкция, задающая унификацию – слишком удобное средство для описания эквивалентных объектов, чтобы от него полностью отказываться. Поэтому можно ввести в язык следующее соглашение: в том месте, где

унификация не может быть произведена, автоматически создаются операции (возможно массовые) для присваивания значений переменных в одну и другую сторону.

Таким образом, в языке допускаются конструкции, следующего вида:

```
<объявление счетчиков> <переменная 1>=<переменная 2>;
```

Например:

```
for i=0:n,j=0:m x[i][j]=y[j][i];
```

Здесь, например, если множество значений счетчиков полностью покрывает один из массивов, то его можно полностью «наложить» на второй, т.е. например, изменить все переменные вида x[i][j] на переменные вида y[j][i]. Это избавит от необходимости отображать в память два массива с одинаковыми элементами.

Синтаксическая конструкция, задающая реализацию ψ-операции, имеет следующий вид:

```
<объявление счетчиков>
  call <список операций>;
```

Здесь «список операций» – это список имен операций, формирующих параметрический терм. Естественно, эти операции должны быть массовыми, и в списке для каждой из них конкретизируются значения индексов, как показано в следующем примере (см. рис. 1.5):

```
operation a[i]={ mcall count1(x[i],x[i+1]); }
  in x[i] out x[i+1];
operation b[i]={ mcall count2(x[i],y[i]); }
  in x[i] out y[i];
operation ab={ for i=1:* call a[i],b[i]; }
  in x[0] out x[1:*],y[*];
```

При использовании циклических конструкций типа while необходим способ получения последнего вычисленного элемента массива. Поэтому в язык было введено специальное обозначение значения индекса последнего вычисленного элемента: x[**last**].

2.3.3 Свойства операций и отображение массивов

Для оказания помощи системе в выборе операций во время планирования было решено ввести понятие «свойство операции». Задача, возлагаемая на свойства операций – это влияние на выбор операции из множества возможных. Значит, свойства операций могут рассматриваться как специальные входные параметры операции. В то же время сам факт использования конкретной операции может повлиять на выбор дальнейшего пути вычислений, значит, свойства могут рассматриваться и как специальные выходные параметры операции.

В разработанном языке описания вычислительных моделей свойства операций – это специальные переменные периода планирования. Они имеют перечисляемый тип, т.е. для каждой из них задается конкретное множество значений-идентификаторов. Объявление свойств операций выполняется за пределами описания вычислительной модели. Например:

```
property algorithm={first,second,third},
         speed={slow,medium,fast},
         precision={low,normal,high};
```

Здесь объявляются два свойства операций: algorithm и speed. Свойство algorithm имеет множество значений first и second, свойство speed – slow, medium и fast. Для каждой операции определяется два множества: множество входных свойств и множество выходных свойств, которые указываются при описании операций следующим образом:

```
operation <имя операции>
  in <список входных переменных>
  out <список выходных переменных>
  property in <список входных свойств>
           out <список выходных свойств>;
```

В списке входных свойств указываются те свойства, на множество значений которых операция накладывает ограничения. С каждым таким свойством указывается множество допустимых значений. В списке выходных

свойств указываются свойства, значение которых операция изменяет после своего выполнения. С каждым таким свойством указывается одно значение. Пример использования свойств операции в описании:

```
operation do_it={ mcall calculate(x,y,z); }
  in x,y out z
  property in precision={normal,high},speed={fast}
           out algorithm=first;
```

Перед планированием пользователь может указать те свойства, которые он хотел бы учитывать при планировании, и задать их начальные значения. Остальные свойства при планировании могут игнорироваться.

Отображение большого многомерного массива, элементы которого постоянно используются в вычислениях, на имеющуюся физическую память может оказаться непосильной задачей для системы конструирования параллельных программ. Поэтому в языке были предусмотрены средства явного задания такого отображения. Для описания отображения необходимо за пределами описания моделей объявить объект «отображение массива» с помощью следующей конструкции:

```
mapping <имя> { <описание отображения> }
```

«Имя» – это идентификатор, позволяющий в дальнейшем использовать это отображение. Конструкции, используемые при описании отображения, аналогичны средствам описания и работы с массивами в вычислительных моделях. Описание содержит объявление двух стандартных массивов:

- array – описание виртуального массива,
- memory – описание массива в памяти.

Далее, после ключевого слова order следует последовательность описаний операций, отражающих порядок обработки массива. Например:

```
mapping array1D_3 // хранится три переменных
{ array[*];            // логический массив
  memory[3];           // физический массив
```

```
  order                // порядок обработки
  { memory[0]=array[0];
    memory[1]=array[1];
    for i=2:*
    { process[i:i+2]; // обработка заданного диапазона
      memory[0]=memory[1];
      memory[1]=memory[2];
      memory[2]=array[i];
    }
  }
}
```

Основными элементами описания обработки являются:

- описание счетчиков,
- оператор присваивания – задает сдвиг элементов логического массива,
- оператор **process** - определяет диапазон обработки элементов массива.

Данный порядок обработки накладывается на план работы с массивом. Если такое отображение оказалось неподходящим для полученного плана (например, параметрический терм в ψ-операции использует четыре компоненты массива или описание переменной array не соответствует описанию массива в модели), то при генерации программы оно во внимание не принимается.

В модели соответствие между массивом и используемым отображением указывается, следующим образом:

```
real x[*]:array1D_3;
```

Необходимым развитием описания отображения массива является параметрическое отображение. При этом описание отображения использует целочисленный параметр, конкретное значение которого будет известно при установлении отображения определенного массива. Например:

```
mapping some(k)
```

```
{ array[*];
  memory[k];
  order { <описание порядка обработки> }
}
model m
{ integer n;
  real x[*]:some(n);
};
```

3. РЕАЛИЗАЦИЯ ТРАНСЛЯТОРА

3.1 Общее описание программы-транслятора

Для обработки текстового описания задачи на языке описания вычислительных моделей был разработан транслятор компилирующего типа. Для его разработки использовался язык программирования Borland C++ Builder 5 с использованием технологии объектно-ориентированного программирования [6], [7]. Транслятор работает под управлением операционной системы Windows. Входными данными транслятора является текстовое описание предметной области в виде совокупности вычислительных моделей, а также формулировка задачи. Транслятор проводит лексический, синтаксический и семантический анализы текста, формирует структуру данных в памяти, отражающую структуру предметной области со всеми ее объектами и связями, и подготавливает описание задач в табличном представлении, удобном для планирования. Выходными данными транслятора является табличное представление предметной области, подготовленное для выполнения планирования.

Транслятор является пошаговым, с возможностью неограниченного отката трансляции. Минимальный шаг трансляции – одна строка. Минимальная единица текста описания, которая может быть обработана транслятором за один шаг – одна лексема.

Программа-транслятор включает в себя многооконный текстовый редактор, построенный на основе модели приложений с множеством документов (MDI). Он позволяет производить набор и элементарное редактирование текста, сохранять текст в файл и загружать из файла, а также просматривать структуру вычислительных моделей в памяти. Внешний вид окна программы представлен на рисунке 3.1.

Процесс работы транслятора можно условно разбить на два этапа:

1. Разбор текста описания и формирование соответствующей структуры в оперативной памяти.

2. Выборка данных из структуры в памяти и формирование табличного описания задачи.

Внешний вид окна программы-транслятора

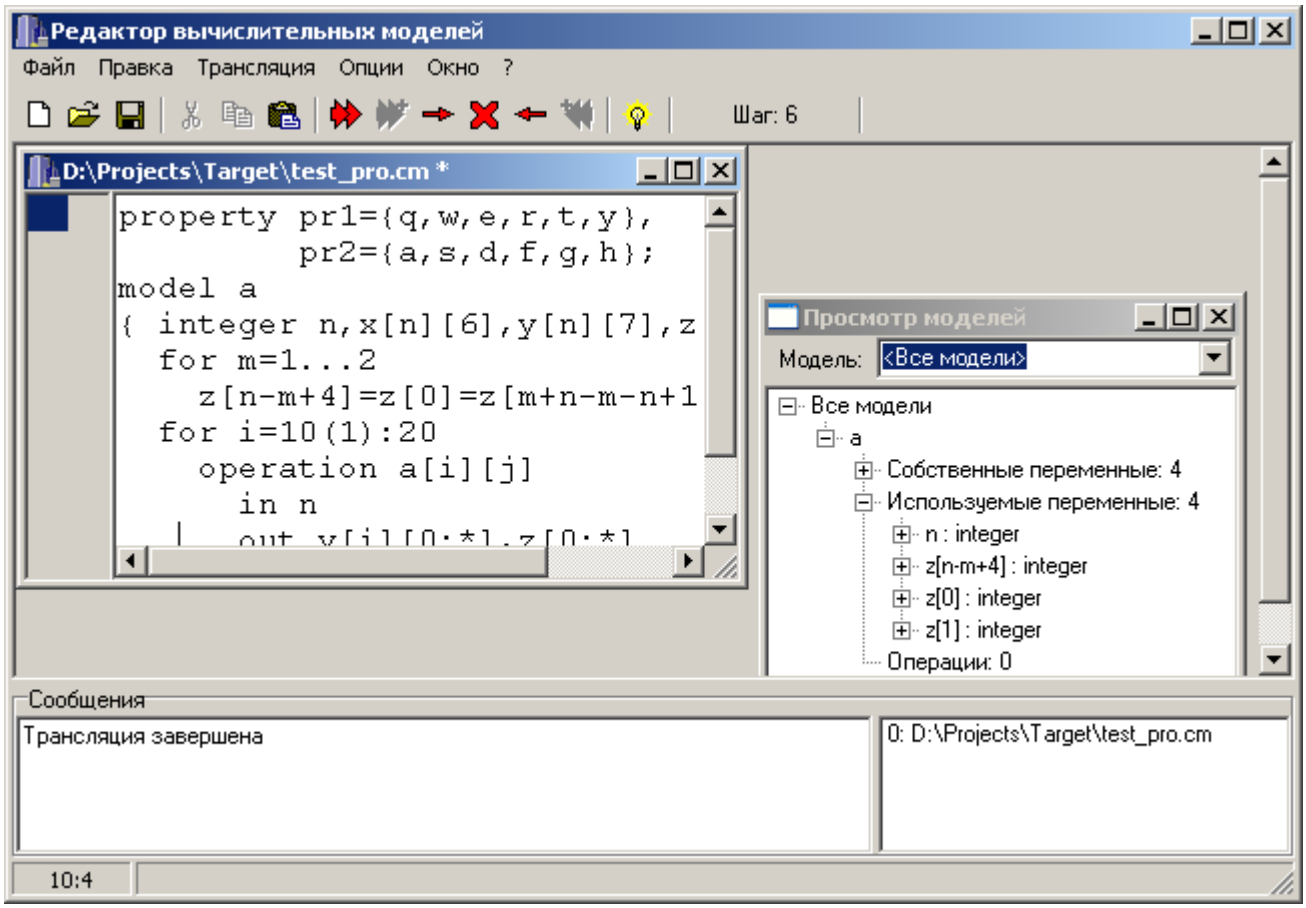


Рис. 3.1

Транслятор является однопроходным. Разбор текста осуществляется последовательно. На каждом шаге разбирается очередная строка, и структура данных в памяти дополняется новой информацией, сохраняя свою целостность, что позволяет выполнить второй этап работы программы, разобрав только часть текстового описания. Табличное описание задачи в этом случае будет формироваться только на основе разобранной части текста.

Транслятор распознает подмножество конструкций, которые были описаны в главе 2, в частности, не распознается описание отображений массивов на физическую память.

3.2 Структура и описание работы транслятора

3.2.1 Разбор текста описания

Первым этапом работы транслятора является разбор текста описания и формирование внутреннего представления предметной области в памяти. Основные классы, участвующие в разборе, представлены в таблице 3.1.

Таблица 3.1

Основные классы, участвующие в разборе текста описания

Название класса	**Основная функция класса**
Translator	Управление процессом трансляции
Source	Управление источниками исходного текста
Scanner	Выполнение лексического анализа
Preprocessor	Выполнение директив препроцессора
Syntax	Выполнение синтаксического анализа
Generator	Выполнение семантического анализа и генерация внутреннего представления предметной области в памяти
Messenger	Вывод сообщений о процессе выполнения и ошибках при трансляции

Класс Translator выполняет общее управление процессом трансляции. Объект класса Translator содержит объекты классов Scanner, Preprocessor, Syntax и Generator, а также связи с объектами классов Source и Messenger. Основными действиями, выполняемыми классом Translator, являются:

- Инициализация процесса трансляции: очистка структур данных и таблиц трансляции, инициализация классов, выполняющих различные виды разбора
- Выполнение очередного шага трансляции: на вход соответствующему методу подается строка символов, полученная от объекта класса Source, после чего последовательно выполняется вызов лексического анализатора, препроцессора, синтаксического анализатора и генератора

структуры в памяти. Также запоминается вся необходимая информация для выполнения отката (позиции в таблицах трансляции, коды выполненных действий и т.п.).

- Откат трансляции на один шаг: вызов генератора структуры в памяти для выполнения необходимых преобразований в структуре, очистка части таблиц трансляции.
- Откат трансляции до заданной позиции в тексте: выполнение отката по одному шагу до достижения заданной позиции.

Класс Source выполняет управление источниками исходного текста: текстовыми файлами и окнами программы-транслятора. Основное назначение класса Source – хранение списка источников текста автоматическое подключение новых источников. Он отслеживает текущую позицию разбора текста и по запросу предоставляет очередную строку для разбора. Основные методы класса предназначены для выполнения следующих функций:

- Регистрация нового источника текста: окна программы или текстового файла.
- Переход текущей позиции текста на новый источник (вызывается объектом класса Translator после распознания препроцессором директивы #include).
- Получение следующей строки текста (вызывается объектом класса Translator).
- Возврат на заданную позицию текста.

Класс Scanner выполняет лексический анализ текста описания. На вход метода класса поступает строка символов, в результате ее обработки происходит заполнение таблиц лексем. Лексический анализатор реализует конечный автомат, работает на основе таблицы перехода в зависимости от типа входного символа, распознает четыре типа лексем:

- идентификаторы – ключевые слова,

- идентификаторы,
- константы (строковые в одинарных и двойных кавычках, целые, вещественные и шестнадцатеричные по стандарту языка Си),
- последовательности символов (не буквенно-цифровых).

При обнаружении недопустимой последовательности символов вызывается метод класса Messenger с кодом ошибки и текстом сообщения. Также при лексическом анализе распознаются и отбрасываются комментарии. Список ключевых слов, допустимых последовательностей символов и символов выделения комментариев задается в текстовом конфигурационном файле.

Класс Preprocessor был реализован для распознавания различных директив в тексте, управляющих процессом трансляции. Единственной директивой, которая нашла применение в трансляторе – это директива #include, указывающая на необходимость включения текста из другого источника в указанное место. При обнаружении данной директивы объект класса Preprocessor передает соответствующее сообщение объекту класса Translator, после чего тот вызывает метод класса Source для использования указанного источника.

Класс Syntax выполняет синтаксический разбор текста методом LL(1). При инициализации объект класса Syntax читает из файла таблицу LL(1)-синтаксического разбора. В процессе разбора синтаксический анализатор выполняет движение по таблице разбора в зависимости от поступаемых на вход лексем. На выходе синтаксического анализатора формируется таблица с кодами действий, которые необходимо далее выполнить транслятору для формирования нужной структуры в памяти. При обнаружении синтаксической ошибки вызывается метод класса Messenger с кодом ошибки и текстом сообщения.

Разработка нового языка, как правило, сопровождается частыми изменениями в его синтаксисе. Построения таблицы синтаксического разбора

вручную является процессом долгим и утомительным. Для облегчения работы с грамматикой языка была написана отдельная программа: редактор грамматики с возможностью автоматического построения таблицы LL(1)-синтаксического разбора. Редактор позволяет формировать грамматику языка в виде множества правил вида:

$$N \rightarrow \alpha,$$

где N – нетерминальный символ,

α – любая последовательность терминальных и нетерминальных символов.

С любой позицией в грамматике имеется возможность связать определенный код действия, который в дальнейшем может быть обработан транслятором при прохождении по таблице разбора. Также редактор позволяет проверить свойство LL(1) языка, сформировать таблицу синтаксического разбора и сохранить ее в файле специального формата, который может быть почитан программой-транслятором. Для одной грамматики редактор позволяет указывать несколько точек входа в грамматику, т.е. нетерминалов, с которых начинается синтаксический разбор. Это позволяет объединить в одной грамматике несколько различных путей разбора. Эта возможность также нашла применение в трансляторе. Внешний вид окна программы – редактора грамматики показан на рисунке 3.2.

Последним этапом работы с текстом является семантический анализ, выполняемый объектом класса Generator. Основной функцией класса является построение в памяти структуры данных, отражающей текстовое описание предметной области. Понятия, используемые при формировании этой структуры, почти полностью соответствуют понятиям, используемым при описании предметной области (вычислительные модели, переменные, операции, типы переменных, …). Используемая структура данных будет подробно рассмотрена далее.

Внешний вид окна редактора грамматики

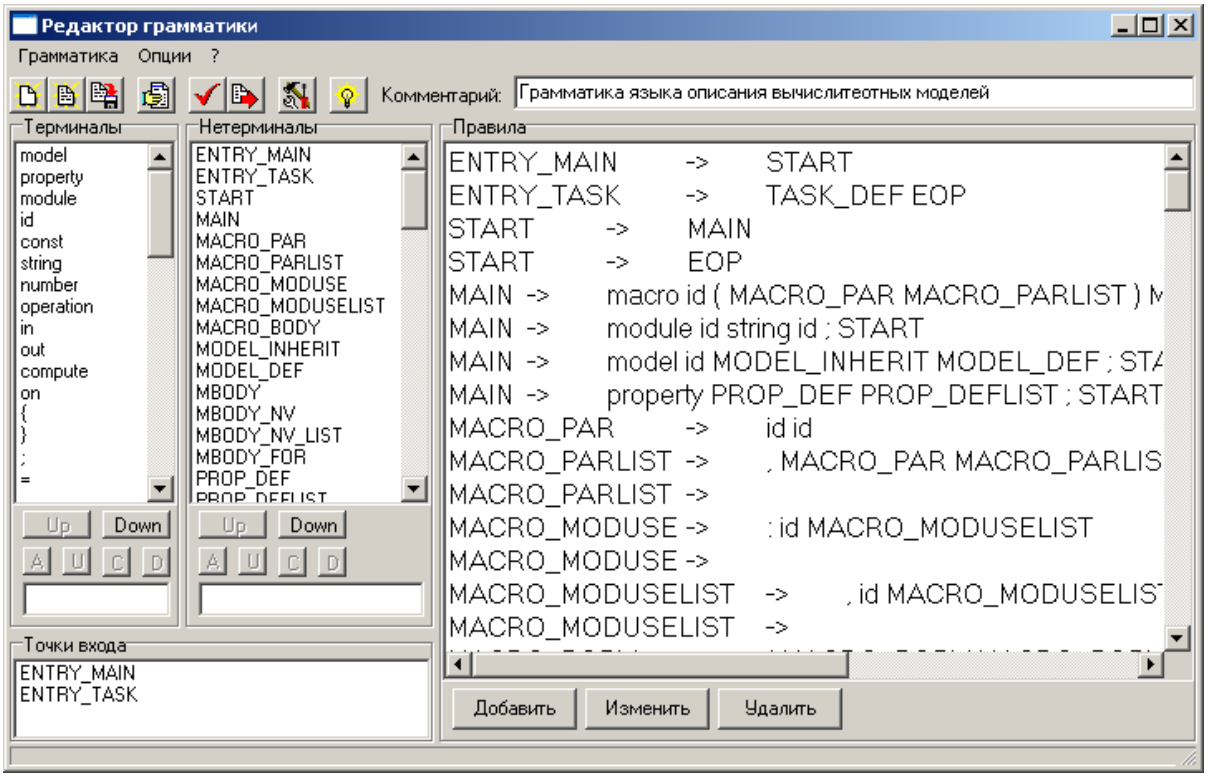


Рис. 3.2.

Основные действия, выполняемые методами класса Generator, это:

- выполнить указанное действие по формированию структуры данных,
- отменить выполненное действие.

Вместе с кодом действия в метод класса в качестве параметра также передается лексема, имеющая отношение к данному действию (например, идентификатор – имя переменной модели).

Реализация класса Generator оказалась наиболее сложной задачей при реализации транслятора. Это вызвано, прежде всего, тем, что описание предметной области в памяти в любой момент может быть использовано для решения задачи, даже если его формирование еще не окончено. Это требует постоянного соблюдения согласованности структуры данных с точки зрения связей между имеющимися объектами. Так что, каждое действие по изменению структуры данных (прямое или обратное) не должно нарушать этой согласованности.

После формирование структуры данных в памяти, отражающей предметную область, на ней можно сформулировать задачу. Формулировка задачи выполняется в специальном окне в текстовом виде аналогично формулировке задачи в теле операции. Для распознавания текста формулировки задачи создается дополнительный объект класса Translator. Для синтаксического разбора этот новый объект использует ту же таблицу разбора, но другую точку входа в таблицу. В результате трансляции формулировки задачи происходит дополнение структуры данных в памяти, после чего выполняются все необходимые действия по работе с задачей (формирование таблиц для планировщика – вторая часть работы транслятора). В конце концов, производится откат трансляции формулировки задачи и возврат к исходному состоянию предметной области в памяти.

Для вывода различных сообщений в программе-трансляторе используется специально созданный для этого класс Messenger. При запуске программы создается один объект этого класса, который связывается с областью вывода сообщений в окне программы. Далее все классы, которые должны выдавать сообщения, могут делать это через объект класса Messenger.

Существует набор стандартных сообщений, которые могут быть выданы пользователю в процессе работы программы. Эти сообщения перечислены в специальном конфигурационном файле, причем каждому из них приписан индивидуальный номер. При запуске программы объект класса Messenger прочитывает этот конфигурационный файл, после чего для вывода определенного сообщения пользователю достаточно указать его номер.

3.2.2 Формирование табличного представления задачи

После завершения разбора текста в оперативной памяти сформировалась структура, отражающая текстовое представление предметной области. Каждая модель, таким образом, содержит набор собственных переменных, часть из которых является структурированными и

тоже содержит переменные и т.д. Такая древовидная структура получилась вследствие предоставления пользователю удобных средств описания объектов. Для выполнения планирования необходимо эту структуру привести к «плоскому» виду, т.е. ввести глобальную нумерацию всех переменных и операций внутри модели.

Задачей формирования таблиц, пригодных для планирования, занимается класс Planner. Он преобразует вычислительные модели из формата, созданного классом Generator, в таблицы, содержащие всю необходимую для планирования информацию. В частности, он создает табличное представление всех моделей, на которых ставились задачи (начальная задача, сформулированная пользователем, или описанная в структурированной операции). В табличном представлении модель к «плоскому» виду, т.е. раскрывает все структурированные переменные. Все переменные и операции из подмоделей помещаются во внешнюю модель. У всех переменных из каждого списка унификации формируется ссылка на одну переменную из этого списка.

3.3 Внутреннее представление данных транслятора

В процессе работы транслятора описание предметной области несколько раз меняет свое представление, начиная от текстового, и заканчивая табличным. Рассмотрим особенности каждого представления.

Перед работой транслятора описание предметной области представлено в виде текста на входном языке, как наиболее близком пользователю. Сейчас пока не рассматриваются такой способ формирования описания предметной области как визуальное программирование, поскольку он может применяться для уже более-менее оформившейся системы описаний. В нашем же случае система описания формировалась параллельно с написанием транслятора. А начальная обработка текста редко различается при распознавании различных формальных языков.

Лексический анализатор (класс Scanner) производит разбиение входного текста на лексемы и частичную классификацию лексем. Поэтому на выходе лексического анализатора формируются набор таблиц с лексемами различных типов, а также таблица со ссылками на эти лексемы, сохраняющая порядок их следования в исходном тексте. В данном трансляторе на выходе лексического анализатора формируется четыре таблицы лексем:

- таблица ключевых слов (специально выделенных идентификаторов),
- таблица идентификаторов,
- таблица констант,
- таблица последовательностей символов.

Таблица ключевых слов и последовательностей символов заполняются из конфигурационного файла, а таблицы идентификаторов и констант – в процессе лексического анализа. Все четыре таблицы лексем имеют хэш-функции для оптимизации доступа.

Синтаксический анализатор (класс Syntax), учитывая классификацию лексем, выполняет проверку синтаксической правильности описания, следуя по таблице синтаксического разбора. По мере прохождения определенных строк таблицы разбора формируется новая таблица, содержащая коды действий, указанные в этих строках. Итак, каждый элемент таблицы действий на выходе синтаксического анализатора содержит код действия и, возможно, ссылку на лексему. Таким образом, в процессе синтаксического анализа убирается весь «синтаксический сахар», и остаются только действительно важные элементы описания.

Генератор структуры (класс Generator) получает на вход последовательность кодов действий, и выполняет указанные действия по формированию необходимой структуры данных. Перечисленные выше представления являются довольно стандартными, и практически всегда используются для задач трансляции. Формируемая же генератором структура данных, как правило, отражает структуру языка, поэтому в различных

трансляторах она обычно различается. После нескольких неудачных попыток в начале разработки сходу сформировать подходящую структуру данных, было решено создать гибкий механизм для формирования различных представлений в памяти, на основе которого можно уже постепенно сформировать конкретное представление знаний. Итак, был создан набор классов для формирования описания различных объектов в памяти.

В основе разработанной структуры лежит базовый класс vObject, имеющий строковое поле Name. Класс vObject, таким образом, описывает какой-либо объект, имеющий имя. От этого класса путем наследования созданы четыре основных класса, из которых строится вся структура данных в памяти:

- vInteger – «Целое число» – класс содержит поле целого типа.
- vString – «Строка» – класс содержит поле строкового типа.
- vSet – «Множество» – класс содержит массив указателей на объекты класса vObject. В дополнение к классу vSet был создан самостоятельный класс vSetItem, содержащий ссылку на любой объект – компонент класса vSet и позволяющий перейти к следующему элементу.
- vReference – «Ссылка» – класс объект класса vSetItem.

Объект любого из четырех классов может «стать частью множества» vSet, т.е. добавить свой указатель в массив объекта vSet. Доступ к элементам множества может быть осуществлен как по имени (у каждого объекта vObject есть имя), так и по номеру. Таким образом, может быть создана сколь угодно сложная древовидная структура, отдельные части которой отражают различные объекты предметной области. Эти объекты могут иметь целочисленные (класс vInteger), строковые (класс VString) или более сложные (класс vSet) поименованные атрибуты, а также просто содержать массив элементов с доступом по индексу. Наличие класса vReference позволяет задавать отношения, отличные от иерархических. Некоторым неудобством является то, что добавление объектов во множество vSet может

происходить только в конец, а удаление только элемента только из конца, из-за особенностей работы лежащей в его основе структуры данных «список массивов». Но этот недостаток не является существенным, так как обычно происходит постепенное заполнение структуры данных, т.е. добавление новых элементов в конец, а при откате – отмена самого последнего действия, т.е. удаление элементов из конца.

С помощью созданного механизма конструирования объектов были разработаны структуры объектов рассматриваемой предметной области (переменные, операции, …). Каждый объект представляется небольшим деревом, в корне которого находится объект класса vSet, содержащий указатели на различные объекты – атрибуты объекта предметной области. Всю структуру каждого объекта предметной области закрепляет соответствующий класс, который выполняет доступ к атрибутам, проходя нужный путь вниз по дереву. Доступ извне к объекту предметной области происходит только через интерфейс соответствующего класса, что позволяет гибко и незаметно для остальных объектов менять внутреннюю структуру объекта. Список объектов, формирующих внутреннее представление предметной области, представлен в приложении 4.

3.4 Результаты работы транслятора

Результатом работы транслятора является табличное представление задачи, готовое для проведения планирования. Основными характеристиками этого представления являются:

- «плоское» представление модели – отсутствие структурированных переменных,
- простое представление множеств индексов в виде логических шаблонов, состоящих из двух частей:
    - нерегулярная часть – логический массив, отмечающий элементы в начале массива,

- регулярная часть – логический массив, отмечающий периодически повторяющиеся элементы.

Формат таблиц представлен в приложении 5.

ЗАКЛЮЧЕНИЕ

Работа рассматривает использование аппарата вычислительных моделей, расширенных введением массовых переменных и операций, для генерации параллельных программ.

На защиту выносятся следующие результаты:

- Проанализировано строение вычислительных моделей с массивами.
- Разработан язык описания задач на вычислительных моделях с массивами.
- Разработано внутреннее представление предметной области, подходящее для использования в пошаговом трансляторе с возможностью отката.
- Реализован транслятор, преобразующий текстовое описание задачи в табличное представление, удобное для планирования алгоритма решения задачи.

Был получен опыт работы с выбранным представлением знаний о вычислениях: рассмотрены различные структуры данных для его реализации и алгоритмы обработки.

В дальнейшей работе над системой конструирования параллельных программ можно выделить следующие направления:

- разработка и использование алгоритмов планирования и генерации программ на вычислительных моделях с массивами;
- разработка библиотеки длительного хранения вычислительных моделей и ее заполнение описанием различных предметных областей;
- создание визуальной среды описания задач на вычислительных моделях;
- разработка технологий конструирования параллельных программ для решения задач численного моделирования на существующих многопроцессорных системах.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Вальковский В. А., Малышкин В. Э. Синтез параллельных программ и систем на вычислительных моделях. – Новосибирск, Наука, 1988.
2. Кахро М.И., Калья А.П., Тыугу Э.Х. Инструментальная система программирования ЕС ЭВМ (ПРИЗ). – 2-е изд., перераб. и доп. – М.: Финансы и статистика, 1988.
3. Тыугу Э. Х. Концептуальное программирование. – М., Наука, 1984.
4. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002.
5. Абрамов С.М., Васенин В.А., Мамчиц Е.Е., Роганов В.А., Слепухин А.Ф. Динамическое распараллеливание программ на базе параллельной редукции графов. Архитектура программного обеспечения новой версии Т-системы. http://cluster.msu.ru\articles\30-10-2000\thesis-html.html
6. Архангельский А.Я. Программирование в C++ Builder 5. – М.: ЗАО «Издательство БИНОМ», 2001.
7. Подбельский В.В. Язык Си++: Учеб. пособие. – 2-е изд., перераб. и доп. – М.: Финансы и статистика, 1996.
8. Немнюгин С. А., Стесик О. Л. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.: БХВ-Петербург, 2002.
9. Осуга С. Обработка знаний: Пер. с япон. – М.: Мир, 1989.
10. Полетаева И. А. Методы трансляции: конспект лекций. – Новосибирск, НГТУ, 1998
11. Серебряков В.А. Лекции по конструированию компиляторов ftp://emerald.hostel.nstu.ru/pub/Text/Documentation/Programming/Translators/conscomp.rar
12. Фомичев В.С. Формальные языки, грамматики и автоматы ftp://emerald.hostel.nstu.ru/pub/Text/Documentation/Programming/Translators/fomichev.rar

13. Мальцев А. Алгоритмы и рекурсивные функции, Наука, 1986.
14. Малышкин В.Э. Основы параллельных вычислений. Учебное пособие. Новосиб. техн. ун-т. Новосибирск, 1997.
15. Малышкин В.Э. ОПАЛ – язык описания параллельных алгоритмов // теоретические вопросы параллельного программирования и многопроцессорные ЭВМ. – Новосибирск: ВЦ СО АН СССР, 1983. с 91-109.
16. Малышкин В. Э. Линеаризация массовых вычислений. Системная информатика № 1, 1991. Под редакцией В. Е. Котова. Наука, Сибирское отделение, Новосибирск. Стр. 229-259.
17. Успенский В.А., Семенов А.Л. Теория алгоритмов: основные открытия и приложения, Наука, 1987, 288 стр.
18. Степин Ю.Г. Генерация вычислительных моделей с помощью графовых грамматик. www.bsu.by/converences/cas99/Abstracts/section4/stepin/stepin.html
19. Гридасов А.Ю. Планирование алгоритмов на структурных вычислительных моделях // Современные проблемы технических наук. Сборник тезисов докладов Новосибирской межвузовской научной студенческой конференции «Интеллектуальный потенциал Сибири». Часть 1.
20. Гридасов А.Ю., Киреев С.Е. Генерация программ обработки массовых данных на основе метода структурного синтеза // Наука. Техника. Инновации // региональная научная конференция студентов, аспирантов и молодых ученых: Тез. докл. в 5-ти частях. Новосибирск: Изд-во НГТУ, 2002. Часть 1.
21. Киреев С.Е, Гридасов А.Ю. Система конструирования параллельных программ и язык описания вычислительных моделей // Наука. Техника. Инновации // региональная научная конференция студентов, аспирантов и

молодых ученых: Тез. докл. в 5-ти частях. Новосибирск: Изд-во НГТУ, 2002. Часть 1.

Приложение 1

LL(1)-грамматика языка описания вычислительных моделей

ENTRY_MAIN	→	START
ENTRY_TASK$_{103}$	→	TASK_DEF$_{1043}$ **EOP**$_{104}$
START	→	MAIN
START	→	**EOP**$_{102}$
MAIN	→	**macro**$_{1100}$ **id**$_{1101}$ **(**$_{1103}$ MACRO_PAR MACRO_PARLIST **)**$_{1106}$ MACRO_MODUSE **{**$_{1107}$ MACRO_BODY E$_{1108}$ START$_{1109}$
MAIN	→	**module**$_{1011}$ **id**$_{1012}$ **string**$_{1013}$ **id**$_{1014}$ **;**$_{1015}$ START
MAIN	→	**model**$_{1001}$ **id**$_{1002}$ MODEL_INHERIT MODEL_DEF **;**$_{1003}$ START
MAIN	→	**property**$_{1004}$ PROP_DEF PROP_DEFLIST **;**$_{1010}$ START
MACRO_PAR	→	**id**$_{1104}$ **id**$_{1105}$
MACRO_PARLIST	→	**,** MACRO_PAR MACRO_PARLIST
MACRO_PARLIST	→	
MACRO_MODUSE	→	**:** **id**$_{1102}$ MACRO_MODUSELIST
MACRO_MODUSE	→	
MACRO_MODUSELIST	→	**,** **id**$_{1102}$ MACRO_MODUSELIST
MACRO_MODUSELIST	→	
MACRO_BODY	→	**{**$_{1114}$ MACRO_BODY MACRO_BODY$_{1114}$
MACRO_BODY	→	**}**
MACRO_BODY	→	**<ANY>**$_{1114}$ MACRO_BODY
MODEL_INHERIT	→	**:** **id**$_{1052}$
MODEL_INHERIT	→	
MODEL_DEF	→	**{**$_{1022}$ MBODY **}**$_{1023}$
MODEL_DEF	→	
MBODY	→	FOR_DEF MBODY_FOR FOR_UP MBODY
MBODY	→	**test** **(** A_EXPR_SIGNED **)**$_{201}$ **;** MBODY
MBODY	→	**typeof** **(**$_{1056}$ CVAR_NAME **)**$_{1057}$ VAR_DEFNAME VAR_DEFLIST **;**$_{1018}$ MBODY
MBODY	→	**undefined**$_{1055}$ VAR_DEFNAME VAR_DEFLIST **;**$_{1018}$ MBODY
MBODY	→	**id**$_{1000}$ SOMEMODELDEF MBODY
MBODY	→	OP_DECLARE MBODY
MBODY	→	
MBODY_NV	→	FOR_DEF MBODY_FOR FOR_UP
MBODY_NV	→	OP_DECLARE
MBODY_NV_LIST	→	MBODY_NV MBODY_NV_LIST
MBODY_NV_LIST	→	
MBODY_FOR	→	MBODY_NV
MBODY_FOR	→	**{** MBODY_NV MBODY_NV_LIST **}**
PROP_DEF	→	**id**$_{1005}$ **=** **{**$_{1006}$ **id**$_{1007}$ PROP_DEFVALLIST **}**$_{1008}$
PROP_DEFLIST	→	**,** PROP_DEF PROP_DEFLIST
PROP_DEFLIST	→	

PROP_DEFVALLIST	→	**,** **id**$_{1007}$ PROP_DEFVALLIST
PROP_DEFVALLIST	→	
SOMEMODELDEF	→	UNI_DECLARE
SOMEMODELDEF$_{1016}$	→	VAR_DEFNAME VAR_DEFLIST **;**$_{1018}$
VAR_DEFNAME	→	**id**$_{1017}$ VAR_DEFARRAY
VAR_DEFARRAY	→	**[**$_{1115}$ INDEX_RANGE **]**$_{1059}$ VAR_DEFARRAY
VAR_DEFARRAY	→	
VAR_DEFLIST$_{1058}$	→	**,** VAR_DEFNAME VAR_DEFLIST
VAR_DEFLIST$_{1058}$	→	
UNI_DECLARE$_{1019}$	→	CVAR_ITEMPOST$_{1020}$ CVAR_NAMEPART **=**$_{1040}$ CVAR_CONST$_{1021}$ UNI_LIST E$_{1026}$ **;**$_{1099}$
UNI_LIST$_{1021}$	→	**=** CVAR_CONST UNI_LIST
UNI_LIST$_{1021}$	→	
CVAR_NAME$_{1019}$	→	CVAR_ITEM CVAR_NAMEPART
CVAR_NAMEPART	→	**.** CVAR_ITEM CVAR_NAMEPART
CVAR_NAMEPART$_{1024}$	→	
CVAR_ITEM	→	**id**$_{1020}$ CVAR_ITEMPOST
CVAR_ITEMPOST	→	**[**$_{1116}$ INDEX_RANGE **]**$_{1053}$ CVAR_ITEMPOST
CVAR_ITEMPOST	→	
CVAR_NAMELIST$_{1021}$	→	**,** CVAR_NAME CVAR_NAMELIST
CVAR_NAMELIST$_{1021}$	→	
CVAR_CONST	→	CVAR_NAME
CVAR_CONST	→	**number**$_{1060}$
CVAR_CONSTLIST$_{1021}$	→	**,** CVAR_CONST CVAR_CONSTLIST
CVAR_CONSTLIST$_{1021}$	→	
OP_DECLARE	→	**operation**$_{1027}$ **id**$_{1028}$ OP_DEFINDEX OP_DEF **;**$_{1031}$
OP_DEFINDEX	→	**[** **id**$_{1009}$ **]** OP_DEFINDEX
OP_DEFINDEX	→	
OP_DEF	→	OP_IS OP_VARIN OP_VAROUT OP_PROP
OP_IS	→	**=** OP_IFRE
OP_IS	→	
OP_IFRE	→	**if** **(** A_EXPR_SIGNED **)**$_{1054}$ OP_REA
OP_IFRE	→	OP_RE
OP_REA	→	OP_RE
OP_REA	→	
OP_RE	→	**{** OP_BODY **}**
OP_VARIN	→	**in**$_{1041}$ CVAR_NAME CVAR_NAMELIST E$_{1029}$
OP_VARIN	→	
OP_VAROUT	→	**out**$_{1042}$ CVAR_NAME CVAR_NAMELIST E$_{1030}$
OP_VAROUT	→	
OP_PROP	→	**property** OP_PROP1
OP_PROP	→	
OP_PROP1	→	OP_PROPOUT
OP_PROP1	→	OP_PROPIN OP_PROP2

OP_PROP2	→	OP_PROPOUT
OP_PROP2	→	
OP_PROPIN	→	**in** OP_PROPINDEF OP_PROPINDEFLIST
OP_PROPINDEF$_{1032}$	→	**id**$_{1033}$ **=** **{** **id**$_{1034}$ OP_PROPINVALLIST **}**
OP_PROPINDEFLIST$_{1035}$	→	**,** OP_PROPINDEF OP_PROPINDEFLIST
OP_PROPINDEFLIST$_{1035}$	→	
OP_PROPINVALLIST	→	**,** **id**$_{1034}$ OP_PROPINVALLIST
OP_PROPINVALLIST	→	
OP_PROPOUT	→	**out**$_{1042}$ OP_PROPOUTDEF OP_PROPOUTDEFLIST
OP_PROPOUTDEF$_{1036}$	→	**id**$_{1037}$ **=** **id**$_{1038}$
OP_PROPOUTDEFLIST$_{1039}$	→	**,** OP_PROPOUTDEF OP_PROPOUTDEFLIST
OP_PROPOUTDEFLIST$_{1039}$	→	
OP_BODY	→	TASK_DEF$_{1046}$ E$_{1051}$
OP_BODY	→	**mcall**$_{1110}$ **id**$_{1111}$ **(**$_{1112}$ CVAR_CONST CVAR_CONSTLIST **)**$_{1113}$ **;**
OP_BODY$_{1093}$	→	FOR_DEF OP_PSIITEM FOR_UP E$_{1095}$
OP_PSIITEM	→	FOR_DEF OP_PSIITEM FOR_UP
OP_PSIITEM	→	**call** COP_NAME COP_NAMELIST **;**
COP_NAME$_{1092}$	→	CVAR_ITEM CVAR_NAMEPART
COP_NAMELIST$_{1094}$	→	**,** COP_NAME COP_NAMELIST
COP_NAMELIST$_{1094}$	→	
FOR_DEF	→	**for**$_{1081}$ FOR_INDEX FOR_INDEXLIST$_{1088}$
FOR_INDEXLIST	→	**,** FOR_INDEX FOR_INDEXLIST$_{1088}$
FOR_INDEXLIST$_{1089}$	→	
FOR_INDEX	→	**id**$_{1082}$ **=** A_EXPR_SIGNED FOR_REST$_{1083}$
FOR_REST	→	FOR_STEP FOR_RANGE
FOR_REST	→	
FOR_STEP	→	**(** A_EXPR_SIGNED **)**$_{1084}$
FOR_STEP$_{1090}$	→	
FOR_RANGE	→	**...**$_{1085}$ FOR_ENDVAL
FOR_RANGE	→	**:**$_{1080}$ FOR_ENDVAL
FOR_ENDVAL	→	*****$_{1087}$
FOR_ENDVAL	→	A_EXPR_SIGNED E$_{1086}$
FOR_UP$_{1091}$	→	
TASK_DEF	→	**compute on** TASK_MODEL TASK_VARIN TASK_VAROUT
TASK_MODEL	→	**model**$_{1001}$ MODEL_INHERIT **{**$_{1022}$ MBODY **}**$_{1023}$ E$_{1045}$
TASK_MODEL	→	**id**$_{1044}$
TASK_VARIN	→	**in**$_{1047}$ CVAR_NAME CVAR_NAMELIST E$_{1048}$
TASK_VARIN	→	
TASK_VAROUT	→	**out**$_{1049}$ CVAR_NAME CVAR_NAMELIST E$_{1050}$
TASK_VAROUT	→	
A_EXPR_SIGNED$_{1061}$	→	A_EXPR_S A_EXPR_FINISH

A_EXPR_UNSIGNED$_{1061}$	→	A_EXPR_U A_EXPR_FINISH
A_EXPR_S	→	A_EXPR_OP1 A_EXPR_U A_EXPR_R
A_EXPR_U	→	**(**$_{1068}$ A_EXPR_S **)**$_{1069}$
A_EXPR_U	→	A_EXPR_T
A_EXPR_R	→	A_EXPR_OP2 A_EXPR_U A_EXPR_R
A_EXPR_R	→	**?**$_{1066}$ A_EXPR_S **:**$_{1067}$ A_EXPR_S
A_EXPR_R	→	
A_EXPR_OP1	→	**+**$_{1064}$
A_EXPR_OP1	→	**-**$_{1064}$
A_EXPR_OP1	→	**!**$_{1064}$
A_EXPR_OP1	→	**~**$_{1064}$
A_EXPR_OP1	→	
A_EXPR_OP2	→	**&&**$_{1065}$
A_EXPR_OP2	→	**||**$_{1065}$
A_EXPR_OP2	→	**-**$_{1065}$
A_EXPR_OP2	→	**%**$_{1065}$
A_EXPR_OP2	→	**>>**$_{1065}$
A_EXPR_OP2	→	**<<**$_{1065}$
A_EXPR_OP2	→	**==**$_{1065}$
A_EXPR_OP2	→	**!=**$_{1065}$
A_EXPR_OP2	→	*****$_{1065}$
A_EXPR_OP2	→	**/**$_{1065}$
A_EXPR_OP2	→	**+**$_{1065}$
A_EXPR_OP2	→	**>**$_{1065}$
A_EXPR_OP2	→	**<**$_{1065}$
A_EXPR_OP2	→	**>=**$_{1065}$
A_EXPR_OP2	→	**<=**$_{1065}$
A_EXPR_OP2	→	**&**$_{1065}$
A_EXPR_OP2	→	**|**$_{1065}$
A_EXPR_OP2	→	**^**$_{1065}$
A_EXPR_T	→	**const**$_{1063}$
A_EXPR_T	→	CVAR_NAME E$_{1062}$
A_EXPR_FINISH$_{1070}$	→	
INDEX_RANGE$_{1071}$	→	A_EXPR_SIGNED INDEX_R
INDEX_RANGE$_{1071}$	→	*****$_{1072}$ E$_{1075}$
INDEX_R$_{1073}$	→	INDEX_STEP INDEX_R1 E$_{1075}$
INDEX_R$_{1076}$	→	
INDEX_R1	→	**:** INDEX_RT
INDEX_STEP	→	**(** A_EXPR_SIGNED **)**$_{1078}$
INDEX_STEP$_{1079}$	→	
INDEX_RT	→	*****$_{1074}$
INDEX_RT	→	A_EXPR_SIGNED E$_{1077}$
E	→	

Приложение 2

Таблица LL(1)-синтаксического разбора грамматики разработанного языка

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
1	model property module macro EOP	2	false	false	true	0	Rule	ENTRY_MAIN ->
2	model property module macro EOP	6	false	false	true	0	N	START
3	compute	4	false	false	true	103	Rule	ENTRY_TASK ->
4	compute	400	false	true	true	1043	N	TASK_DEF
5	EOP	0	true	false	true	104	T	EOP
6	model property module macro	8	false	false	false	0	Rule	START ->
7	EOP	9	false	false	true	0	Rule	START ->
8	model property module macro	10	false	false	true	0	N	MAIN
9	EOP	0	true	false	true	102	T	EOP
10	macro	14	false	false	false	0	Rule	MAIN ->
11	module	25	false	false	false	0	Rule	MAIN ->
12	model	31	false	false	false	0	Rule	MAIN ->
13	property	37	false	false	true	0	Rule	MAIN ->
14	macro	15	true	false	true	1100	T	macro
15	id	16	true	false	true	1101	T	id
16	(	17	true	false	true	1103	T	(
17	id	42	false	true	true	0	N	MACRO_PAR
18	,)	45	false	true	true	0	N	MACRO_PARLIST
19	)	20	true	false	true	1106	T	)
20	{ :	51	false	true	true	0	N	MACRO_MODUSE
21	{	22	true	false	true	1107	T	{
22	<любой> { }	63	false	true	true	0	N	MACRO_BODY
23	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	535	false	true	true	1108	N	E
24	model property module macro EOP	6	false	false	true	1109	N	START
25	module	26	true	false	true	1011	T	module
26	id	27	true	false	true	1012	T	id
27	string	28	true	false	true	1013	T	string
28	id	29	true	false	true	1014	T	id
29	;	30	true	false	true	1015	T	;
30	model property module macro EOP	6	false	false	true	0	N	START
31	model	32	true	false	true	1001	T	model
32	id	33	true	false	true	1002	T	id

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
33	{ ; :	72	false	true	true	0	N	MODEL_INHERIT
34	{ ;	77	false	true	true	0	N	MODEL_DEF
35	;	36	true	false	true	1003	T	;
36	model property module macro EOP	6	false	false	true	0	N	START
37	property	38	true	false	true	1004	T	property
38	id	133	false	true	true	0	N	PROP_DEF
39	; ,	140	false	true	true	0	N	PROP_DEFLIST
40	;	41	true	false	true	1010	T	;
41	model property module macro EOP	6	false	false	true	0	N	START
42	id	43	false	false	true	0	Rule	MACRO_PAR ->
43	id	44	true	false	true	1104	T	id
44	id	0	true	false	true	1105	T	id
45	,	47	false	false	false	0	Rule	MACRO_PARLIST ->
46	)	50	false	false	true	0	Rule	MACRO_PARLIST ->
47	,	48	true	false	true	0	T	,
48	id	42	false	true	true	0	N	MACRO_PAR
49	,)	45	false	false	true	0	N	MACRO_PARLIST
50	)	0	false	false	true	0	Empty	
51	:	53	false	false	false	0	Rule	MACRO_MODUSE ->
52	{	56	false	false	true	0	Rule	MACRO_MODUSE ->
53	:	54	true	false	true	0	T	:
54	id	55	true	false	true	1102	T	id
55	{ ,	57	false	false	true	0	N	MACRO_MODUSELIST
56	{	0	false	false	true	0	Empty	
57	,	59	false	false	false	0	Rule	MACRO_MODUSELIST ->
58	{	62	false	false	true	0	Rule	MACRO_MODUSELIST ->
59	,	60	true	false	true	0	T	,
60	id	61	true	false	true	1102	T	id
61	{ ,	57	false	false	true	0	N	MACRO_MODUSELIST
62	{	0	false	false	true	0	Empty	
63	{	66	false	false	false	0	Rule	MACRO_BODY ->
64	}	69	false	false	false	0	Rule	MACRO_BODY ->
65	<любой>	70	false	false	true	0	Rule	MACRO_BODY ->
66	{	67	true	false	true	1114	T	{
67	<любой> { }	63	false	true	true	0	N	MACRO_BODY
68	<любой> { }	63	false	false	true	1114	N	MACRO_BODY
69	}	0	true	false	true	0	T	}
70	<любой>	71	true	false	true	1114	T	<любой>

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
71	<любой> { }	63	false	false	true	0	N	MACRO_BODY
72	:	74	false	false	false	0	Rule	MODEL_INHERIT ->
73	{ ;	76	false	false	true	0	Rule	MODEL_INHERIT ->
74	:	75	true	false	true	0	T	:
75	id	0	true	false	true	1052	T	id
76	{ ;	0	false	false	true	0	Empty	
77	{	79	false	false	false	0	Rule	MODEL_DEF ->
78	;	82	false	false	true	0	Rule	MODEL_DEF ->
79	{	80	true	false	true	1022	T	{
80	id operation } for typeof undefined	83	false	true	true	0	N	MBODY
81	}	0	true	false	true	1023	T	}
82	;	0	false	false	true	0	Empty	
83	for	89	false	false	false	0	Rule	MBODY ->
84	typeof	93	false	false	false	0	Rule	MBODY ->
85	undefined	101	false	false	false	0	Rule	MBODY ->
86	id	106	false	false	false	0	Rule	MBODY ->
87	operation	109	false	false	false	0	Rule	MBODY ->
88	}	111	false	false	true	0	Rule	MBODY ->
89	for	361	false	true	true	0	N	FOR_DEF
90	id operation { for	126	false	true	true	0	N	MBODY_FOR
91	id operation } for typeof undefined	398	false	true	true	0	N	FOR_UP
92	id operation } for typeof undefined	83	false	false	true	0	N	MBODY
93	typeof	94	true	false	true	0	T	typeof
94	(	95	true	false	true	1056	T	(
95	id	188	false	true	true	0	N	CVAR_NAME
96	)	97	true	false	true	1057	T	)
97	id	158	false	true	true	0	N	VAR_DEFNAME
98	; ,	168	false	true	true	0	N	VAR_DEFLIST
99	;	100	true	false	true	1018	T	;
100	id operation } for typeof undefined	83	false	false	true	0	N	MBODY
101	undefined	102	true	false	true	1055	T	undefined
102	id	158	false	true	true	0	N	VAR_DEFNAME
103	; ,	168	false	true	true	0	N	VAR_DEFLIST
104	;	105	true	false	true	1018	T	;
105	id operation } for typeof undefined	83	false	false	true	0	N	MBODY
106	id	107	true	false	true	1000	T	id
107	id = . [	152	false	true	true	0	N	SOMEMODELDEF

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
108	id operation } for typeof undefined	83	false	false	true	0	N	MBODY
109	operation	223	false	true	true	0	N	OP_DECLARE
110	id operation } for typeof undefined	83	false	false	true	0	N	MBODY
111	}	0	false	false	true	0	Empty	
112	for	115	false	false	false	0	Rule	MBODY_NV ->
113	operation	118	false	false	false	0	Rule	MBODY_NV ->
114	id	119	false	false	true	0	Rule	MBODY_NV ->
115	for	361	false	true	true	0	N	FOR_DEF
116	id operation { for	126	false	true	true	0	N	MBODY_FOR
117	id operation } for typeof undefined	398	false	false	true	0	N	FOR_UP
118	operation	223	false	false	true	0	N	OP_DECLARE
119	id	120	true	false	true	1000	T	id
120	= . [	174	false	false	true	0	N	UNI_DECLARE
121	id operation for	123	false	false	false	0	Rule	MBODY_NV_LIST ->
122	}	125	false	false	true	0	Rule	MBODY_NV_LIST ->
123	id operation for	112	false	true	true	0	N	MBODY_NV
124	id operation } for	121	false	false	true	0	N	MBODY_NV_LIST
125	}	0	false	false	true	0	Empty	
126	{	128	false	false	false	0	Rule	MBODY_FOR ->
127	id operation for	132	false	false	true	0	Rule	MBODY_FOR ->
128	{	129	true	false	true	0	T	{
129	id operation for	112	false	true	true	0	N	MBODY_NV
130	id operation } for	121	false	true	true	0	N	MBODY_NV_LIST
131	}	0	true	false	true	0	T	}
132	id operation for	112	false	false	true	0	N	MBODY_NV
133	id	134	false	false	true	0	Rule	PROP_DEF ->
134	id	135	true	false	true	1005	T	id
135	=	136	true	false	true	0	T	=
136	{	137	true	false	true	1006	T	{
137	id	138	true	false	true	1007	T	id
138	} ,	146	false	true	true	0	N	PROP_DEFVALLIST
139	}	0	true	false	true	1008	T	}
140	,	142	false	false	false	0	Rule	PROP_DEFLIST ->
141	;	145	false	false	true	0	Rule	PROP_DEFLIST ->
142	,	143	true	false	true	0	T	,
143	id	133	false	true	true	0	N	PROP_DEF
144	; ,	140	false	false	true	0	N	PROP_DEFLIST
145	;	0	false	false	true	0	Empty	
146	,	148	false	false	false	0	Rule	PROP_DEFVALLIST->

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
147	}	151	false	false	true	0	Rule	PROP_DEFVALLIST->
148	,	149	true	false	true	0	T	,
149	id	150	true	false	true	1007	T	id
150	} ,	146	false	false	true	0	N	PROP_DEFVALLIST
151	}	0	false	false	true	0	Empty	
152	id	154	false	false	false	1016	Rule	SOMEMODELDEF ->
153	= . [	157	false	false	true	0	Rule	SOMEMODELDEF ->
154	id	158	false	true	true	0	N	VAR_DEFNAME
155	; ,	168	false	true	true	0	N	VAR_DEFLIST
156	;	0	true	false	true	1018	T	;
157	= . [	174	false	false	true	0	N	UNI_DECLARE
158	id	159	false	false	true	0	Rule	VAR_DEFNAME ->
159	id	160	true	false	true	1017	T	id
160	; , [	161	false	false	true	0	N	VAR_DEFARRAY
161	[	163	false	false	false	0	Rule	VAR_DEFARRAY ->
162	; ,	167	false	false	true	0	Rule	VAR_DEFARRAY ->
163	[	164	true	false	true	1115	T	[
164	id const (! ~ + - *	509	false	true	true	0	N	INDEX_RANGE
165	]	166	true	false	true	1059	T	]
166	; , [	161	false	false	true	0	N	VAR_DEFARRAY
167	; ,	0	false	false	true	0	Empty	
168	,	170	false	false	false	1058	Rule	VAR_DEFLIST ->
169	;	173	false	false	true	1058	Rule	VAR_DEFLIST ->
170	,	171	true	false	true	0	T	,
171	id	158	false	true	true	0	N	VAR_DEFNAME
172	; ,	168	false	false	true	0	N	VAR_DEFLIST
173	;	0	false	false	true	0	Empty	
174	= . [=	175	false	false	true	1019	Rule	UNI_DECLARE ->
175	property id operation out { } ; = , . ( ) call for : + - * / % >> << & | ^ ? [ ] ... > < >= <= == != && || EOP	200	false	true	true	1020	N	CVAR_ITEMPOST
176	property id operation out { } ; = , . ( ) call for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	191	false	true	true	0	N	CVAR_NAMEPART
177	=	178	true	false	true	1040	T	=
178	id number	213	false	true	true	1021	N	CVAR_CONST
179	; =	182	false	true	true	0	N	UNI_LIST

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
180	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	535	false	true	true	1026	N	E
181	;	0	true	false	true	1099	T	;
182	=	184	false	false	false	1021	Rule	UNI_LIST ->
183	;	187	false	false	true	1021	Rule	UNI_LIST ->
184	=	185	true	false	true	0	T	=
185	id number	213	false	true	true	0	N	CVAR_CONST
186	; =	182	false	false	true	0	N	UNI_LIST
187	;	0	false	false	true	0	Empty	
188	id	189	false	false	true	1019	Rule	CVAR_NAME ->
189	id	197	false	true	true	0	N	CVAR_ITEM
190	property id operation out { } ; = , . ( ) call for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	191	false	false	true	0	N	CVAR_NAMEPART
191	.	193	false	false	false	0	Rule	CVAR_NAMEPART ->
192	property id operation out { } ; = , ( ) call for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	196	false	false	true	1024	Rule	CVAR_NAMEPART ->
193	.	194	true	false	true	0	T	.
194	id	197	false	true	true	0	N	CVAR_ITEM
195	property id operation out { } ; = , . ( ) call for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	191	false	false	true	0	N	CVAR_NAMEPART
196	property id operation out { } ; = , ( ) call for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	0	false	false	true	0	Empty	
197	id	198	false	false	true	0	Rule	CVAR_ITEM ->
198	id	199	true	false	true	1020	T	id
199	property id operation out { } ; = , . ( ) call for : + - * / % >> << & | ^ ? [ ] ... > < >= <= == != && || EOP	200	false	false	true	0	N	CVAR_ITEMPOST
200	[	202	false	false	false	0	Rule	CVAR_ITEMPOST ->
201	property id operation out { } ; = , . ( ) call for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	206	false	false	true	0	Rule	CVAR_ITEMPOST ->

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
202	[	203	true	false	true	1116	T	[
203	id const (! ~ + - *	509	false	true	true	0	N	INDEX_RANGE
204	]	205	true	false	true	1053	T	]
205	property id operation out { } ; = , . ( ) call for : + - * / % >> << & | ^ ? [ ] ... > < >= <= == != && || EOP	200	false	false	true	0	N	CVAR_ITEMPOST
206	property id operation out { } ; = , . ( ) call for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	0	false	false	true	0	Empty	
207	,	209	false	false	false	1021	Rule	CVAR_NAMELIST ->
208	property out } ; EOP	212	false	false	true	1021	Rule	CVAR_NAMELIST ->
209	,	210	true	false	true	0	T	,
210	id	188	false	true	true	0	N	CVAR_NAME
211	property out } ; , EOP	207	false	false	true	0	N	CVAR_NAMELIST
212	property out } ; EOP	0	false	false	true	0	Empty	
213	number	215	false	false	false	0	Rule	CVAR_CONST ->
214	id	216	false	false	true	0	Rule	CVAR_CONST ->
215	number	0	true	false	true	1060	T	number
216	id	188	false	false	true	0	N	CVAR_NAME
217	,	219	false	false	false	1021	Rule	CVAR_CONSTLIST ->
218	)	222	false	false	true	1021	Rule	CVAR_CONSTLIST ->
219	,	220	true	false	true	0	T	,
220	id number	213	false	true	true	0	N	CVAR_CONST
221	,)	217	false	false	true	0	N	CVAR_CONSTLIST
222	)	0	false	false	true	0	Empty	
223	operation	224	false	false	true	0	Rule	OP_DECLARE ->
224	operation	225	true	false	true	1027	T	operation
225	id	226	true	false	true	1028	T	id
226	property in out ; = [	229	false	true	true	0	N	OP_DEFINDEX
227	property in out ; =	236	false	true	true	0	N	OP_DEF
228	;	0	true	false	true	1031	T	;
229	[	231	false	false	false	0	Rule	OP_DEFINDEX ->
230	property in out ; =	235	false	false	true	0	Rule	OP_DEFINDEX ->
231	[	232	true	false	true	0	T	[
232	id	233	true	false	true	1009	T	id
233	]	234	true	false	true	0	T	]
234	property in out ; = [	229	false	false	true	0	N	OP_DEFINDEX
235	property in out ; =	0	false	false	true	0	Empty	
236	property in out ; =	237	false	false	true	0	Rule	OP_DEF ->
237	property in out ; =	241	false	true	true	0	N	OP_IS

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
238	property in out ;	262	false	true	true	0	N	OP_VARIN
239	property out ;	269	false	true	true	0	N	OP_VAROUT
240	property ;	276	false	false	true	0	N	OP_PROP
241	=	243	false	false	false	0	Rule	OP_IS ->
242	property in out ;	245	false	false	true	0	Rule	OP_IS ->
243	=	244	true	false	true	0	T	=
244	{ if	246	false	false	true	0	N	OP_IFRE
245	property in out ;	0	false	false	true	0	Empty	
246	{	248	false	false	false	0	Rule	OP_IFRE ->
247	if	249	false	false	true	0	Rule	OP_IFRE ->
248	{	258	false	false	true	0	N	OP_RE
249	if	250	true	false	true	0	T	if
250	(	251	true	false	true	0	T	(
251	id const (! ~ + -	429	false	true	true	0	N	A_EXPR_SIGNED
252	)	253	true	false	true	1054	T	)
253	property in out { ;	254	false	false	true	0	N	OP_REA
254	{	256	false	false	false	0	Rule	OP_REA ->
255	property in out ;	257	false	false	true	0	Rule	OP_REA ->
256	{	258	false	false	true	0	N	OP_RE
257	property in out ;	0	false	false	true	0	Empty	
258	{	259	false	false	true	0	Rule	OP_RE ->
259	{	260	true	false	true	0	T	{
260	compute mcall for	327	false	true	true	0	N	OP_BODY
261	}	0	true	false	true	0	T	}
262	in	264	false	false	false	0	Rule	OP_VARIN ->
263	property out ;	268	false	false	true	0	Rule	OP_VARIN ->
264	in	265	true	false	true	1041	T	in
265	id	188	false	true	true	0	N	CVAR_NAME
266	property out } ; , EOP	207	false	true	true	0	N	CVAR_NAMELIST
267	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	535	false	false	true	1029	N	E
268	property out ;	0	false	false	true	0	Empty	
269	out	271	false	false	false	0	Rule	OP_VAROUT ->
270	property ;	275	false	false	true	0	Rule	OP_VAROUT ->
271	out	272	true	false	true	1042	T	out
272	id	188	false	true	true	0	N	CVAR_NAME
273	property out } ; , EOP	207	false	true	true	0	N	CVAR_NAMELIST

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
274	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	535	false	false	true	1030	N	E
275	property ;	0	false	false	true	0	Empty	
276	property	278	false	false	false	0	Rule	OP_PROP ->
277	;	280	false	false	true	0	Rule	OP_PROP ->
278	property	279	true	false	true	0	T	property
279	in out	281	false	false	true	0	N	OP_PROP1
280	;	0	false	false	true	0	Empty	
281	in	283	false	false	false	0	Rule	OP_PROP1 ->
282	out	285	false	false	true	0	Rule	OP_PROP1 ->
283	in	290	false	true	true	0	N	OP_PROPIN
284	out ;	286	false	false	true	0	N	OP_PROP2
285	out	313	false	false	true	0	N	OP_PROPOUT
286	out	288	false	false	false	0	Rule	OP_PROP2 ->
287	;	289	false	false	true	0	Rule	OP_PROP2 ->
288	out	313	false	false	true	0	N	OP_PROPOUT
289	;	0	false	false	true	0	Empty	
290	in	291	false	false	true	0	Rule	OP_PROPIN ->
291	in	292	true	false	true	0	T	in
292	id	294	false	true	true	0	N	OP_PROPINDEF
293	out ; ,	301	false	false	true	0	N	OP_PROPINDEFLIST
294	id	295	false	false	true	1032	Rule	OP_PROPINDEF ->
295	id	296	true	false	true	1033	T	id
296	=	297	true	false	true	0	T	=
297	{	298	true	false	true	0	T	{
298	id	299	true	false	true	1034	T	id
299	} ,	307	false	true	true	0	N	OP_PROPINVALLIST
300	}	0	true	false	true	0	T	}
301	,	303	false	false	false	1035	Rule	OP_PROPINDEFLIST ->
302	out ;	306	false	false	true	1035	Rule	OP_PROPINDEFLIST->
303	,	304	true	false	true	0	T	,
304	id	294	false	true	true	0	N	OP_PROPINDEF
305	out ; ,	301	false	false	true	0	N	OP_PROPINDEFLIST
306	out ;	0	false	false	true	0	Empty	
307	,	309	false	false	false	0	Rule	OP_PROPINVALLIST->
308	}	312	false	false	true	0	Rule	OP_PROPINVALLIST->
309	,	310	true	false	true	0	T	,
310	id	311	true	false	true	1034	T	id

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
311	} ,	307	false	false	true	0	N	OP_PROPINVALLIST
312	}	0	false	false	true	0	Empty	
313	out	314	false	false	true	0	Rule	OP_PROPOUT ->
314	out	315	true	false	true	1042	T	out
315	id	317	false	true	true	0	N	OP_PROPOUTDEF
316	; ,	321	false	false	true	0	N	OP_PROPOUTDEFLIST
317	id	318	false	false	true	1036	Rule	OP_PROPOUTDEF ->
318	id	319	true	false	true	1037	T	id
319	=	320	true	false	true	0	T	=
320	id	0	true	false	true	1038	T	id
321	,	323	false	false	false	1039	Rule	OP_PROPOUTDEFLIST ->
322	;	326	false	false	true	1039	Rule	OP_PROPOUTDEFLIST ->
323	,	324	true	false	true	0	T	,
324	id	317	false	true	true	0	N	OP_PROPOUTDEF
325	; ,	321	false	false	true	0	N	OP_PROPOUTDEFLIST
326	;	0	false	false	true	0	Empty	
327	mcall	330	false	false	false	0	Rule	OP_BODY ->
328	for	337	false	false	false	1093	Rule	OP_BODY ->
329	compute	341	false	false	true	0	Rule	OP_BODY ->
330	mcall	331	true	false	true	1110	T	mcall
331	id	332	true	false	true	1111	T	id
332	(	333	true	false	true	1112	T	(
333	id number	213	false	true	true	0	N	CVAR_CONST
334	,)	217	false	true	true	0	N	CVAR_CONSTLIST
335	)	336	true	false	true	1113	T	)
336	;	0	true	false	true	0	T	;
337	for	361	false	true	true	0	N	FOR_DEF
338	call for	343	false	true	true	0	N	OP_PSIITEM
339	id operation } for typeof undefined	398	false	true	true	0	N	FOR_UP
340	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	535	false	false	true	1095	N	E
341	compute	400	false	true	true	1046	N	TASK_DEF
342	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	535	false	false	true	1051	N	E

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
343	call	345	false	false	false	0	Rule	OP_PSIITEM ->
344	for	349	false	false	true	0	Rule	OP_PSIITEM ->
345	call	346	true	false	true	0	T	call
346	id	352	false	true	true	0	N	COP_NAME
347	; ,	355	false	true	true	0	N	COP_NAMELIST
348	;	0	true	false	true	0	T	;
349	for	361	false	true	true	0	N	FOR_DEF
350	call for	343	false	true	true	0	N	OP_PSIITEM
351	id operation } for typeof undefined	398	false	false	true	0	N	FOR_UP
352	id	353	false	false	true	1092	Rule	COP_NAME ->
353	id	197	false	true	true	0	N	CVAR_ITEM
354	property id operation out { } ; = , . ( ) call for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	191	false	false	true	0	N	CVAR_NAMEPART
355	,	357	false	false	false	1094	Rule	COP_NAMELIST ->
356	;	360	false	false	true	1094	Rule	COP_NAMELIST ->
357	,	358	true	false	true	0	T	,
358	id	352	false	true	true	0	N	COP_NAME
359	; ,	355	false	false	true	0	N	COP_NAMELIST
360	;	0	false	false	true	0	Empty	
361	for	362	false	false	true	0	Rule	FOR_DEF ->
362	for	363	true	false	true	1081	T	for
363	id	371	false	true	true	0	N	FOR_INDEX
364	id operation { , call for	365	false	false	true	1088	N	FOR_INDEXLIST
365	,	367	false	false	false	0	Rule	FOR_INDEXLIST ->
366	id operation { call for	370	false	false	true	1089	Rule	FOR_INDEXLIST ->
367	,	368	true	false	true	0	T	,
368	id	371	false	true	true	0	N	FOR_INDEX
369	id operation { , call for	365	false	false	true	1088	N	FOR_INDEXLIST
370	id operation { call for	0	false	false	true	0	Empty	
371	id	372	false	false	true	0	Rule	FOR_INDEX ->
372	id	373	true	false	true	1082	T	id
373	=	374	true	false	true	0	T	=
374	id const (! ~ + -	429	false	true	true	0	N	A_EXPR_SIGNED
375	id operation { , (call for : ...	376	false	false	true	1083	N	FOR_REST
376	(: ...	378	false	false	false	0	Rule	FOR_REST ->
377	id operation { , call for	380	false	false	true	0	Rule	FOR_REST ->
378	(: ...	381	false	true	true	0	N	FOR_STEP
379	: ...	387	false	false	true	0	N	FOR_RANGE

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
380	id operation { , call for	0	false	false	true	0	Empty	
381	(	383	false	false	false	0	Rule	FOR_STEP ->
382	: ...	386	false	false	true	1090	Rule	FOR_STEP ->
383	(	384	true	false	true	0	T	(
384	id const (! ~ + -	429	false	true	true	0	N	A_EXPR_SIGNED
385	)	0	true	false	true	1084	T	)
386	: ...	0	false	false	true	0	Empty	
387	:	389	false	false	false	0	Rule	FOR_RANGE ->
388	...	391	false	false	true	0	Rule	FOR_RANGE ->
389	:	390	true	false	true	1080	T	:
390	id const (! ~ + - *	393	false	false	true	0	N	FOR_ENDVAL
391	...	392	true	false	true	1085	T	...
392	id const (! ~ + - *	393	false	false	true	0	N	FOR_ENDVAL
393	id const (! ~ + -	395	false	false	false	0	Rule	FOR_ENDVAL ->
394	*	397	false	false	true	0	Rule	FOR_ENDVAL ->
395	id const (! ~ + -	429	false	true	true	0	N	A_EXPR_SIGNED
396	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	535	false	false	true	1086	N	E
397	*	0	true	false	true	1087	T	*
398	id operation } for typeof undefined	399	false	false	true	1091	Rule	FOR_UP ->
399	id operation } for typeof undefined	0	false	false	true	0	Empty	
400	compute	401	false	false	true	0	Rule	TASK_DEF ->
401	compute	402	true	false	true	0	T	compute
402	on	403	true	false	true	0	T	on
403	model id	406	false	true	true	0	N	TASK_MODEL
404	in out } EOP	415	false	true	true	0	N	TASK_VARIN
405	out } EOP	422	false	false	true	0	N	TASK_VAROUT
406	id	408	false	false	false	0	Rule	TASK_MODEL ->
407	model	409	false	false	true	0	Rule	TASK_MODEL ->
408	id	0	true	false	true	1044	T	id
409	model	410	true	false	true	1001	T	model
410	{ ; :	72	false	true	true	0	N	MODEL_INHERIT
411	{	412	true	false	true	1022	T	{
412	id operation } for typeof undefined	83	false	true	true	0	N	MBODY
413	}	414	true	false	true	1023	T	}

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
414	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	535	false	false	true	1045	N	E
415	in	417	false	false	false	0	Rule	TASK_VARIN ->
416	out } EOP	421	false	false	true	0	Rule	TASK_VARIN ->
417	in	418	true	false	true	1047	T	in
418	id	188	false	true	true	0	N	CVAR_NAME
419	property out } ; , EOP	207	false	true	true	0	N	CVAR_NAMELIST
420	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	535	false	false	true	1048	N	E
421	out } EOP	0	false	false	true	0	Empty	
422	out	424	false	false	false	0	Rule	TASK_VAROUT ->
423	} EOP	428	false	false	true	0	Rule	TASK_VAROUT ->
424	out	425	true	false	true	1049	T	out
425	id	188	false	true	true	0	N	CVAR_NAME
426	property out } ; , EOP	207	false	true	true	0	N	CVAR_NAMELIST
427	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	535	false	false	true	1050	N	E
428	} EOP	0	false	false	true	0	Empty	
429	id const (! ~ + -	430	false	false	true	1061	Rule	A_EXPR_SIGNED ->
430	id const (! ~ + -	435	false	true	true	0	N	A_EXPR_S
431	id operation { , ( ) call for :] ...	507	false	false	true	0	N	A_EXPR_FINISH
432	id const (	433	false	false	true	1061	Rule	A_EXPR_UNSIGNED->
433	id const (	439	false	true	true	0	N	A_EXPR_U
434	id operation { , ( ) call for :] ...	507	false	false	true	0	N	A_EXPR_FINISH
435	id const (! ~ + -	436	false	false	true	0	Rule	A_EXPR_S ->
436	id const (! ~ + -	456	false	true	true	0	N	A_EXPR_OP1
437	id const (	439	false	true	true	0	N	A_EXPR_U
438	id operation { , ( ) call for : + - * / % >> << & | ^ ?] ... > < >= <= == != && ||	445	false	false	true	0	N	A_EXPR_R
439	id const	441	false	false	false	0	Rule	A_EXPR_U ->
440	(	442	false	false	true	0	Rule	A_EXPR_U ->

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
441	id const	502	false	false	true	0	N	A_EXPR_T
442	(	443	true	false	true	1068	T	(
443	id const (! ~ + -	435	false	true	true	0	N	A_EXPR_S
444	)	0	true	false	true	1069	T	)
445	?	448	false	false	false	0	Rule	A_EXPR_R ->
446	+ - * / % >> << & | ^ > < >= <= == != && ||	452	false	false	false	0	Rule	A_EXPR_R ->
447	id operation { , ( ) call for :] ...	455	false	false	true	0	Rule	A_EXPR_R ->
448	?	449	true	false	true	1066	T	?
449	id const (! ~ + -	435	false	true	true	0	N	A_EXPR_S
450	:	451	true	false	true	1067	T	:
451	id const (! ~ + -	435	false	false	true	0	N	A_EXPR_S
452	+ - * / % >> << & | ^ > < >= <= == != && ||	466	false	true	true	0	N	A_EXPR_OP2
453	id const (	439	false	true	true	0	N	A_EXPR_U
454	id operation { , ( ) call for : + - * / % >> << & | ^ ?] ... > < >= <= == != && ||	445	false	false	true	0	N	A_EXPR_R
455	id operation { , ( ) call for :] ...	0	false	false	true	0	Empty	
456	-	461	false	false	false	0	Rule	A_EXPR_OP1 ->
457	!	462	false	false	false	0	Rule	A_EXPR_OP1 ->
458	~	463	false	false	false	0	Rule	A_EXPR_OP1 ->
459	+	464	false	false	false	0	Rule	A_EXPR_OP1 ->
460	id const (	465	false	false	true	0	Rule	A_EXPR_OP1 ->
461	-	0	true	false	true	1064	T	-
462	!	0	true	false	true	1064	T	!
463	~	0	true	false	true	1064	T	~
464	+	0	true	false	true	1064	T	+
465	id const (	0	false	false	true	0	Empty	
466	||	484	false	false	false	0	Rule	A_EXPR_OP2 ->
467	-	485	false	false	false	0	Rule	A_EXPR_OP2 ->
468	%	486	false	false	false	0	Rule	A_EXPR_OP2 ->
469	>>	487	false	false	false	0	Rule	A_EXPR_OP2 ->
470	<<	488	false	false	false	0	Rule	A_EXPR_OP2 ->
471	==	489	false	false	false	0	Rule	A_EXPR_OP2 ->
472	!=	490	false	false	false	0	Rule	A_EXPR_OP2 ->
473	*	491	false	false	false	0	Rule	A_EXPR_OP2 ->
474	/	492	false	false	false	0	Rule	A_EXPR_OP2 ->
475	+	493	false	false	false	0	Rule	A_EXPR_OP2 ->
476	>	494	false	false	false	0	Rule	A_EXPR_OP2 ->
477	<	495	false	false	false	0	Rule	A_EXPR_OP2 ->

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
478	>=	496	false	false	false	0	Rule	A_EXPR_OP2 ->
479	<=	497	false	false	false	0	Rule	A_EXPR_OP2 ->
480	&	498	false	false	false	0	Rule	A_EXPR_OP2 ->
481	|	499	false	false	false	0	Rule	A_EXPR_OP2 ->
482	^	500	false	false	false	0	Rule	A_EXPR_OP2 ->
483	&&	501	false	false	true	0	Rule	A_EXPR_OP2 ->
484	||	0	true	false	true	1065	T	||
485	-	0	true	false	true	1065	T	-
486	%	0	true	false	true	1065	T	%
487	>>	0	true	false	true	1065	T	>>
488	<<	0	true	false	true	1065	T	<<
489	==	0	true	false	true	1065	T	==
490	!=	0	true	false	true	1065	T	!=
491	*	0	true	false	true	1065	T	*
492	/	0	true	false	true	1065	T	/
493	+	0	true	false	true	1065	T	+
494	>	0	true	false	true	1065	T	>
495	<	0	true	false	true	1065	T	<
496	>=	0	true	false	true	1065	T	>=
497	<=	0	true	false	true	1065	T	<=
498	&	0	true	false	true	1065	T	&
499	|	0	true	false	true	1065	T	|
500	^	0	true	false	true	1065	T	^
501	&&	0	true	false	true	1065	T	&&
502	id	504	false	false	false	0	Rule	A_EXPR_T ->
503	const	506	false	false	true	0	Rule	A_EXPR_T ->
504	id	188	false	true	true	0	N	CVAR_NAME
505	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	535	false	false	true	1062	N	E
506	const	0	true	false	true	1063	T	const
507	id operation { , ( ) call for :] ...	508	false	false	true	1070	Rule	A_EXPR_FINISH ->
508	id operation { , ( ) call for :] ...	0	false	false	true	0	Empty	
509	*	511	false	false	false	1071	Rule	INDEX_RANGE ->
510	id const (! ~ + -	513	false	false	true	1071	Rule	INDEX_RANGE ->
511	*	512	true	false	true	1072	T	*

N	Терминалы	Jump	Accept	Stack	Error	Action	Тип	Элемент грамматики
512	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	535	false	false	true	1075	N	E
513	id const (! ~ + -	429	false	true	true	0	N	A_EXPR_SIGNED
514	(:]	515	false	false	true	0	N	INDEX_R
515	(:	517	false	false	false	1073	Rule	INDEX_R ->
516	]	520	false	false	true	1076	Rule	INDEX_R ->
517	(:	524	false	true	true	0	N	INDEX_STEP
518	:	521	false	true	true	0	N	INDEX_R1
519	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	535	false	false	true	1075	N	E
520	]	0	false	false	true	0	Empty	
521	:	522	false	false	true	0	Rule	INDEX_R1 ->
522	:	523	true	false	true	0	T	:
523	id const (! ~ + - *	530	false	false	true	0	N	INDEX_RT
524	(	526	false	false	false	0	Rule	INDEX_STEP ->
525	:	529	false	false	true	1079	Rule	INDEX_STEP ->
526	(	527	true	false	true	0	T	(
527	id const (! ~ + -	429	false	true	true	0	N	A_EXPR_SIGNED
528	)	0	true	false	true	1078	T	)
529	:	0	false	false	true	0	Empty	
530	id const (! ~ + -	532	false	false	false	0	Rule	INDEX_RT ->
531	*	534	false	false	true	0	Rule	INDEX_RT ->
532	id const (! ~ + -	429	false	true	true	0	N	A_EXPR_SIGNED
533	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	535	false	false	true	1077	N	E
534	*	0	true	false	true	1074	T	*
535	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	536	false	false	true	0	Rule	E ->
536	model property module id operation in out { } ; , ( ) call macro for : + - * / % >> << & | ^ ?] ... > < >= <= == != && || EOP	0	false	false	true	0	Empty	

Приложение 3

Примеры описания на разработанном языке

Пример 1

```
module C "math.h" math;
// макроопределения
macro length(real x1,real y1,real x2,real y2,real len):math
{ $len=sqrt(($x2-$x1)*($x2-$x1)+(y2-y1)*($y2-$y1)); }
macro sum3half(real x,real y,real z,real p) {
$p=($x+$y+$z)/2; }
macro geron(real a,real b,real c,real p,real s):math
{ $s=sqrt($p*($p-$a)*($p-$b)*($p-$c)); }
// модель "точка"
model point { real x,y; };
// модель "отрезок"
model sector
{ point p1,p2;
  real length;
  operation a={ mcall length(p1.x,p1.y,p2.x,p2.y,length); }
    in p1.x,p1.y,p2.x,p2.y out length;
};
// модель "треугольник"
model triangle
{ sector s1,s2,s3;
  s1.p2=s2.p1; s2.p2=s3.p1; s3.p2=s1.p1;
  real area,p;
  operation count_p=
    { mcall sum3half(s1.length,s2.length,s3.length,p); }
    in s1.length,s2.length,s3.length out p;
  operation count_area=
    { mcall geron(s1.length,s2.length,s3.length,p,area); }
    in s1.length,s2.length,s3.length,p out area;
};
```

Пример постановки задачи (вычисление площади треугольника по трем сторонам):

```
  compute on triangle
    in s1.length,s2.length,s3.length
    out area
```

Пример 2

```
macro mul_add(real x,real y,real z,real s)
{ $s=$x*$y+$z; }
macro mul(real x,real y,real z) { $z=$x*$y; }
// скалярное произведение
model inner_product
{ integer n;
  real x[n],y[n],s[n];
  operation start={ mcall mul(x[0],y[0],s[0]); }
    in x[0],y[0] out s[0];
  operation add[i]=
    { mcall mul_add(x[i+1],y[i+1],s[i],s[i]); }
    in x[i+1],y[i+1],s[i] out s[i+1];
  operation do={ for i=1:n-1 call add[i] }
    in x[*],y[*],s[0] out s[*];
}
```

Пример постановки задачи (вычисление скалярного произведения):

```
compute on inner_product
  in n,x[0:n-1],y[0:n-1] out s[n-1];
```

Приложение 4

Описание объектов, формирующих внутреннее представление предметной области в трансляторе.

Название объекта	**Основные компоненты**
vSubject – предметная область	Множества объектов vType, vProperty, vModule, vMacro, vModel, vTemplate, vTask.
vType – тип объектов модели: переменных и подмоделей	Информация о типе объектов: для подмоделей – ссылка на модель, для переменных неопределенного типа – ссылка на шаблон ограничений.
vProperty свойство операции	Список допустимых значений.
vModule – внешняя библиотека	Имя файла и тип библиотеки.
vMacro – макроопределение	Описание параметров (набор ссылок на объекты vType) и фрагмент текста программы на языке Си.
vModel – вычислительная модель	Множества объектов vOwnVariable, vVariable, vVariableComponent, vVariableTemplate, vOperation, vAExpression, vAExpressionItem, vForExpression, vForSet, vConst, vUnification.
vOwnVariable – собственная переменная модели	Ссылка на объект vType, для каждого измерения по одной ссылке на объект vForSet.
vVariable – переменная модели, использованная в описании	Ссылка на объект vType, множество объектов vVariableComponent, ссылка на объект vVariableTemplate.
vVariableComponent – элемент составного имени переменной модели	Ссылка на объект vOwnVariable, для каждого измерения по одной ссылке на объект vForSet либо vForExpression.
vVariableTemplate – шаблон составного имени переменных (без учета индексов)	Набор ссылок на объекты vOwnVariable, множество ссылок на объекты vVariable.
vAExpression – арифметическое выражение	Множество объектов vAExpressionItem.
vAExpressionItem – элемент арифметического выражения	Тип элемента. В зависимости от типа может быть: - ссылка на объект vVariable, - ссылка на объект vForSet
vForExpression – индексное выражение	Множество ссылок на объекты vVariable, vForSet, целочисленное значение.

vForSet – множество значений индексов	Три ссылки на объекты vForExpression (начальное значение, шаг, конечное значение).
vOperation – операция	Два множества ссылок на объекты vVariable (параметры), два множества ссылок на объекты vProperty (свойства), для условных – ссылка на объект vAExpression, в зависимости от типа реализации операции одно из следующих: - ссылка на объект vMacro; - ссылка на объект vTask; - множество ссылок на объекты vTask, множество ссылок на объекты vForExpression, множество ссылок на объекты vForSet.
vConst – константа	Тип и значение константы (в строке).
vUnification – унификация	Множество ссылок на объекты vVariable, ссылка на объект vConst.
vTemplate – шаблон ограничений переменных неопределенного типа	Множество ссылок на объекты vOwnVariable.
vTask – постановка задачи	Ссылка на объект vModel, два множества ссылок на объекты vVariable (параметры).

Приложение 5

Формат основных таблиц, получаемых в результате работы транслятора

1. Описание модели – таблицы могут использоваться для получения информации об объектах при планировании.

PVarUse – таблица используемых в модели переменных

Поле	**Описание**
N	номер элемента в таблице
var	ссылка на объект vVariable
IsA	номер элемента таблицы PVarUse (учет унификаций)
Template	номер элемента таблицы PVarTemplate
Pattern	шаблон массива (описывает множество значений индексов, содержит номера элементов в таблице PCounter)
In	множество номеров элементов таблицы POper
Out	множество номеров элементов таблицы POper
unif	номер элемента таблицы unif

PVarTemplate – таблица шаблонов составных имен переменных (без учета индексов)

Поле	**Описание**
N	номер элемента в таблице
vt	ссылка на объект vVariableTemplate
Name	имя (уникальное в пределах модели)
Parent	номер элемента таблицы PVarStruct (подмодель, содержащая переменную)
IsA	номер элемента таблицы PVarTemplate (объединение ссылок на одну переменную из различных подмоделей)
Var	множество номеров элементов таблицы PVarUse – множество переменных, соответствующих шаблону
Struct	множество номеров элементов таблицы PVarStruct – последовательность включения родительских подмоделей

PVarStruct – таблица собственных переменных и подмоделей

Поле	**Описание**
N	номер элемента в таблице
ov	ссылка на объект vOwnVariable
pm	номер вычислительной модели, соответствующей подмодели
Pattern	шаблон массива (определяет множества значений индексов, заданных при описании массива переменных или подмоделей)
Parent	номер элемента таблицы PVarStruct (подмодель, содержащая объект)

POper – таблица операций

N	номер элемента таблицы
op	ссылка на объект vOperation
Name	имя (уникальное в пределах модели)
Parent	номер элемента таблицы PVarStruct (подмодель, содержащая операцию)
Pattern	шаблон массива (содержит номера элементов таблицы PCounter)
In	множество номеров элементов таблицы PVarUse
Out	множество номеров элементов таблицы PVarUse
Body	описание реализации операции
PIn	описание входных свойств
POut	описание выходных свойств

PUnif – таблица унификаций

N	номер элемента в таблице
uni	ссылка на объект vUnification
Parent	номер элемента в таблице PVarStruct (подмодель, содержащая описание унификации)
Var	множество номеров элементов таблицы PVarUse
Const	Тип и значение константы (в случае унификации с константой)

PCounter – таблица счетчиков

N	номер элемента в таблице
fs	ссылка на объект vForSet
Pattern	шаблон массива (определяет множества значений индексов счетчика)

2. Описание задачи – таблицы могут использоваться для инициализации процесса планирования.

Var – таблица переменных модели, учитываемых при планировании

N	номер элемента в таблице
Num	номер элемента таблицы PVarUse

Op – таблица операций модели, учитываемых при планировании

N	номер элемента в таблице
Num	номер элемента таблицы POper

Source – таблица входных переменных задачи

N	номер элемента в таблице
Num	номер элемента таблицы Var

Target – таблица выходных переменных задачи

N	номер элемента в таблице
Num	номер элемента таблицы Var

Counter – таблица счетчиков (для планирования на ПВМ в пси-операциях)

N	номер элемента в таблице
Num	номер элемента таблицы PCounter