УДК 681.3

А.Е. Дорошенко, Е.А. Яценко

СРЕДСТВА АВТОМАТИЗАЦИИ РАЗРАБОТКИ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ НА ОСНОВЕ ОНТОЛОГИЙ И АЛГЕБР АЛГОРИТМОВ

Предложен подход к формализованной разработке параллельных программ на основе использования онтологий и аппарата алгебры алгоритмов. С помощью онтологии описываются основные объекты разрабатываемой программы из выбранной предметной области – данные, функции и взаимосвязи между функциями. Дальнейшее проектирование приложения осуществляется в разработанном интегрированном инструментарии проектирования и генерации программ, основывающемся на системах алгоритмических алгебр. Подход проиллюстрирован на примере разработки параллельной MPI-программы сортировки.

Введение

В последние годы онтологии широко используются для описания и синтеза программ [1–3]. Отметим, что одной из основных причин нарастающего интереса к онтологиям является разработка Semantic Web [4] и Semantic Grid [5], которая может рассматриваться как управление знаниями в глобальном масштабе. Онтологии подобны концептуальным методам моделирования, таким, как UML [6] или модель типа “сущность–связь” [7]. Однако, онтологии как правило содержат в себе основанные на логике языки представлений с формальной семантикой и выполнимыми вычислениями. Последние позволяют проводить рассуждения и выполнять запросы с использованием семантических описаний на этапах разработки, развертывания или выполнения программы. Выполнение рассуждений и запросов позволяют автоматизировать или, по крайней мере, облегчить задачи управления разработкой программы, например, прогнозировать или наблюдать за взаимодействием, конфликтами или поведением компонентов программы. Алгеброалгоритмический метод, используемый в нашей работе [8], дополняет онтологии подробными высокоуровневыми спецификациями программ, что обеспечивает возможность синтеза программного кода. В отличие от вышеупомянутых работ по синтезу программ, разрабатываемые нами инструментальные средства позволяют разработчику в интерактивном режиме осуществлять трансформации программ.

В данной работе предложен подход к проектированию программ на основе онтологий и алгеброалгоритмического аппарата. С помощью онтологии предметной области описывается каркас разрабатываемого алгоритма – обрабатываемые данные; названия функций, их входные и выходные параметры, а также взаимосвязи между функциями. В качестве средств разработки онтологии были выбраны язык OWL и система Protégé [9, 10]. Алгеброалгоритмическую часть рассматриваемого подхода составляет разработанный в ИПС НАН Украины интегрированный инструментарий Проектирования и Синтеза программ (“ИПС”) [8]. Данный инструментарий предназначен для проектирования алгоритмов в виде схем с использованием систем алгоритмических алгебр Глушкова (САА) [11, 12]. По схемам алгоритмов выполняется генерация программного кода на выбранном целевом языке программирования (C++, Java). В данной работе разработанная в Protégé онтология предметной области передается в инструментарий “ИПС”, в котором осуществляется генерация соответствующей каркасной схемы алгоритма для выполнения следующего этапа проектирования – наполнения функций их алгоритмическим содержанием (детализации алгоритмов функций) и генерации программы на языке программирования. Подход проиллюстрирован на примере разработки параллельной MPI-программы сортировки.

Материал работы состоит из таких разделов. В разделе 1 рассматривается

применение аппарата онтологий для проектирования алгоритмов на примере задачи сортировки. Раздел 2 посвящен разработанному интегрированному инструментарию и его использованию для дальнейшей разработки программ, описанных онтологиями, с применением средств алгебр алгоритмов.

1. Проектирование программ с использованием онтологий

Онтология представляет собой формальное описание концепций (называемых также классами) рассматриваемой предметной области, свойств (слотов) каждой концепции, описывающих разнообразные атрибуты концепции, а также ограничений на свойства [13]. Она также может содержать экземпляры – конкретные объекты рассматриваемой предметной области. Свойства представляют собой бинарные отношения между концепциями. Различают так называемые объектные свойства – отношения между “концепцией и концепцией”, и свойства типов данных – отношения между “концепцией и значением типа данных”. Онтология вместе с множеством экземпляров классов составляют базу знаний. В данной работе экземпляры использованы для представления конкретных данных, обрабатываемых в программе, а также структуры программы.

Различают онтологии предметных областей и онтологии задач [5]. Онтологии предметных областей представляют необходимые объекты, свойства и отношения между объектами в определенной предметной области (например, биоинформатике, географии или медицине). Онтологии задач содержат концепции задач и процессов, их взаимосвязи и свойства. Например, такие онтологии могут содержать совокупность описаний этапов химического процесса, протокол зависимостей между методами оптимизации и т.п.

В работах, посвященных использованию онтологий для спецификации, генерации и синтеза программ [1, 2, 14], выделяют два источника концепций, описывающих объекты разрабатываемых программ – прикладную область (application space) и область решений (solution space). Прикладная область содержит все концепции, необходимые для описания прикладных задач в выбранной предметной области: типы данных, операции, преобразовывающие данные и предикаты. Упомянутые концепции задаются экспертами предметной области. Для концепций устанавливаются связи с соответствующими представлениями в языке спецификации, конструкциями в промежуточном языке программирования и конструкциями на целевом языке программирования. Концепции из прикладной области используются для составления спецификации решаемой задачи. Область решений содержит все концепции, необходимые для формулирования программ, которые генерируются, – элементы промежуточного языка или языков; элементы целевого языка и среды программирования; алгоритмы и их составные части [2].

В данной работе аппарат онтологий был применен для описания алгоритмов из области сортировки массивов. В качестве средств разработки онтологии был выбран язык Web Ontology Language (OWL) и система Protégé 3.2.1 [9, 10].

На рис. 1 показана иерархия классов разработанной онтологии. Подклассы класса `Data` представляют различные структуры данных, используемые в программах сортировки. Класс `ArrayToProcess` описывает сортируемый массив; `Pointer` – указатель, перемещающийся в процессе обработки по массиву, и отмечающий текущие обрабатываемые элементы сортируемого массива; `PointerArray` – массив указателей (обращение к конкретному указателю осуществляется по его индексу в массиве). Подклассы класса `File` отображают разновидности файлов, которые могут использоваться в программах (входной, выходной и файл для регистрации времени выполнения программы). Класс `Variable` предназначен для описания переменных, которые не относятся ни к одной из вышеперечисленных категорий. `Parameter` – формальные параметры составных операторов (функций), используемых в программе. Все подклассы класса `Data` имеют свойство `HasName` – идентификатор переменной и свойство `IsOfType`, связывающее экземпляр переменной с определенным типом данных,

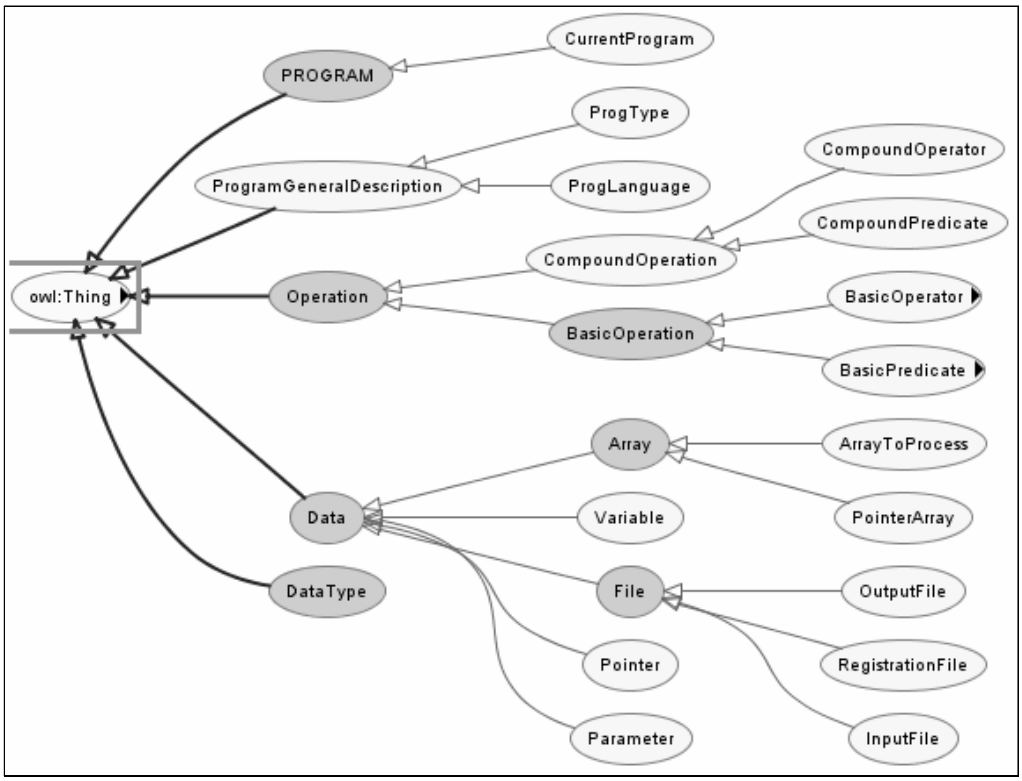

Рис. 1. Онтология, описывающая концепции для решения задач сортировки

представляющим собой экземпляр класса `DataType`. Кроме того, класс `Array` и его подклассы имеют дополнительное свойство `InitialSize`, а класс `Variable` – свойство `InitialValue`.

- Класс `Operation` представляет операции, применяемые к данным в программе – операторы и предикаты. Операции могут быть базисными или составными. Базисный предикат или оператор – первичная атомарная абстракция, используемая для построения схем алгоритмов. Составные операторы и предикаты строятся из элементарных посредством операций последовательного и параллельного выполнения операторов [12] (см. также раздел 2). Понятие составного элемента соответствует понятию функции или метода в программировании. Каждый базисный элемент соответствует определенному базисному элементу в базе данных интегрированного инструментария [8]. Все операции имеют такие свойства: `HasName` – идентификатор операции; `HasParameter` – свойство, связывающее операцию с экземплярами класса `Parameter`; `OperationOutput` – тип возвращаемого значения. Кроме того, базисные операции имеют дополнительное свойство `HasSAAText` – запись операции на языке САА, соответствующая тексту этой операции в базе данных “ИПС”. Составные операции также имеют дополнительное свойство `UsesOperation`, в котором указываются составные или базисные операторы, вызываемые в функции.

Класс `PROGRAM` предназначен для спецификации экземпляров программ, которые используют данные, определенные подклассами класса `Data`, и операции – экземпляры класса `Operation`. Предполагается, что онтология может содержать несколько экземпляров класса `PROGRAM`. Подкласс `CurrentProgram` класса `PROGRAM` используется для того, чтобы указать текущую программу, по которой в интегрированном инструментарии необходимо сгенерировать каркасную схему (см. раздел 2). Класс `ProgramGeneralDescription` предназначен для задания дополнительных характеристик программы – типа программы (обычная или MPI-программа) и целевого языка программирования. Свойства, ассоциированные с классом `PROGRAM` рассматриваются далее в примере 1.

Пример 1. На рис. 2 показан пример созданного в Protégé экземпляра MPI-программы адресной сортировки [8]. Поля приведенной формы содержат значения свойств программы. Свойство `HasName` содержит название программы (алгоритма). В свойстве `UsesData` указана совокупность переменных, которые используются в программе. Поле `IncludesCompoundOperation` содержит названия составных операторов алгоритма. В поле `ProgramCalls` указываются составные или базисные операторы, которые вызываются в основной функции (`main`). Свойство "`rdfs:comment`" содержит комментарий к программе. В полях `ProgramLanguage` и `ProgramType` указаны целевой язык (C++) и тип программы (MPI) соответственно.

□ Рассмотрим основные данные и составные операторы, используемые в алгоритме адресной сортировки. Алгоритм вычисляет для каждого элемента входного массива `in_array` его индекс (адрес) в выходном (отсортированном) массиве `out_array`. Входной массив считывается из файла `infile`, а отсортированный записывается в файл `outfile`. Функция `main` выполняет сортировку входного массива, запуская для этого `proc_num` параллельных процессов, каждый из которых вызывает функцию `sorting`. В свою очередь, `sorting` вызывает такие функции:

• `initialize` – функция, выполняющая чтение входного массива из файла и резервирование памяти для массивов, используемых в алгоритме;

• `calc_elem_pos` – вычисляет выходные индексы элементов массива `in_array` и сохраняет их в массиве `indexbuf`;

• `confluence` – осуществляет вставку элементов массива `in_array` в массив `out_array`, в соответствии с полученными выходными индексами.

Более подробно реализация данного алгоритма рассматривается в разделе 2 (см. пример 2).

В табл. 1 приведено описание составного оператора `sorting`, используемого в программе `adr_sort`. Данное описание было введено в Protégé с помощью формы, аналогичной рис. 2. Столбцы таблицы помечены названиями свойств составного оператора, назначение которых рассматривалось выше. Отметим, что в свойстве `UsesOperation` идентификаторы экземпляров базисных операторов записаны с большой буквы (например, `Shift_pointer_to_the_right`), а составных – с малой. Аналогичным образом описываются и другие составные операторы рассматриваемого алгоритма.

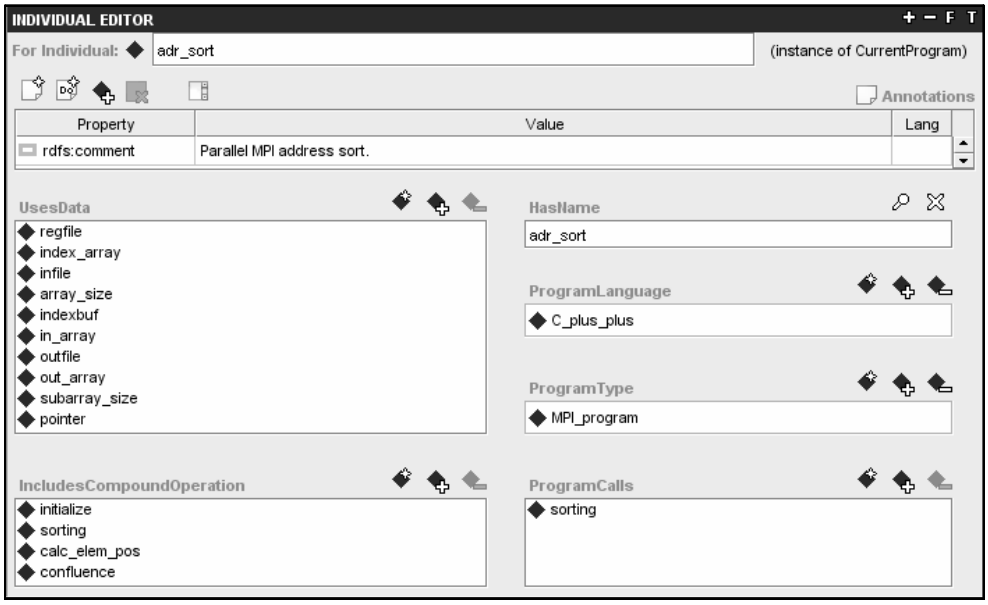

Рис. 2. Экземпляр MPI-программы адресной сортировки, созданный в Protégé

Таблица 1. Значения свойств составного оператора `sorting`

HasName	OperationOutput	HasParameter	UsesOperation
sorting	void	–	• initialize; • Shift_pointer_to_the_right; • calc_elem_pos; • confluence; • Output_array_to_file; • Output_information_about_ time_expired.

2. Генерация кода алгоритма на основе его онтологического описания

Онтологическое описание программы, рассмотренное в предыдущем разделе, далее используется для автоматической генерации соответствующей каркасной схемы алгоритма. Генерация осуществляется с помощью разработанного интегрированного инструментария [8], предназначенного для диалогового конструирования схем алгоритмов в САА и синтеза соответствующих программ на целевом языке программирования. Каркасная схема затем используется для более детального проектирования алгоритма в “ИПС”. Конструирование алгоритма в “ИПС” осуществляется путем суперпозиции языковых конструкций САА (операций САА, базисных операторов и предикатов), которые пользователь выбирает из списка. При этом на каждом шаге проектирования “ИПС” предоставляет пользователю только те конструкции, вставка которых в дерево алгоритма не нарушает синтаксическую правильность схемы. По сконструированному таким образом алгоритму “ИПС” выполняет генерацию программы на целевом языке программирования. Описание САА операций и базисных элементов, а также их отображение в язык программирования хранится в базе данных “ИПС”.

К основным операциям САА относятся [12]:

• `“op1” THEN “op2”` (или `“op1” * “op2”`) – последовательное выполнение операторов `“op1”` и `“op2”`;

• `IF ‘condition’ THEN “op1” ELSE “op2” END IF` – условное выполнение операторов;

• `WHILE NOT ‘condition’ CYCLE “op1” END OF CYCLE` – цикл.

Текст базисных операторов в САА-схемах заключается в двойные кавычки, базисных предикатов – в одинарные.

БД “ИПС” содержит также дополнительные операции, предназначенные для проектирования параллельных MPI-программ [16]:

• `PARALLEL_MPI(i = 0,..., m–1) ( “op1” )` – операция, которая инициализирует MPI вычисления и запускает *m* процессов, каждый из которых выполняет оператор `“op1”`, и завершает MPI вычисления после выполнения этого оператора. Реализация этой операции на языке программирования содержит вызовы стандартных процедур `MPI_Init` и `MPI_Finalize` [15];

• `“Collect data (var1) of type (data_type) with length (n) from all processes to variable (var2) of process with rank (root_proc)”` – базисный оператор, соответствующий функции `MPI_Gather` [15];

• базисные операторы, соответствующие MPI функциям `MPI_Send` и `MPI_Receive` [15, 16].

Для генерации каркасной САА схемы по онтологии предметной области на вход “ИПС” передается OWL файл онтологии, построенной в Protégé. В “ИПС” осуществляется синтаксический анализ данного файла, в процессе которого извлекается информация о таких элементах:

• классах онтологии и их подклассах (иерархия классов) вместе с комментариями (свойства “rdfs:comment”);

• экземплярах классов, включая значения ассоциированных с ними свойств.

При построении онтологии в Protégé для установления связи между элементами онтологии и соответствующими элементами схемы алгоритма было использовано свойство “rdfs:comment” (комментарий к классу). Так, для класса Array строки комментария имеют вид

```
SAABinding
SAAType=“BasicOperator”
SAAText=“Declare an array
({HasName}) of type ({IsOfType})”
SAADBCode=“25”
```

Здесь SAABinding – ключевое слово, указывающее на то, что комментарий содержит информацию о привязке элемента онтологии к элементу САА-схемы; SAAType – тип элемента; SAAText – текст элемента схемы, который необходимо сгенерировать; в фигурных скобках указаны имена свойств, значения которых нужно подставить в текст; SAADBCode – код базисного элемента в БД “ИПС”. В данном случае приведенный комментарий означает, что для экземпляра класса Array необходимо сгенерировать базисный оператор определения массива. Например, при значениях свойств HasName = “in_array” и IsOfType = “int”, будет сгенерирован такой базисный оператор:

```
“Declare an array (in_array)
of type (int)”,
```

где в скобках указаны параметры базисного элемента – идентификатор массива и его тип.

Генерация каркасной САА-схемы осуществляется по экземпляру класса CurrentProgram (см. раздел 1). Основой для генерации является файл с заранее подготовленной шаблонной САА-схемой, содержащей пустой блок описания глобальных переменных и определение основного составного оператора (main) схемы без реализации. В случае, если в свойстве ProgramType указано значение “MPI_program”, то используется шаблон, который дополнительно содержит операцию инициализации MPI вычислений. В упомянутый шаблон вставляются извлеченные из онтологии элементы. Вначале обрабатываются значения свойства UsesData – используемые в программе экземпляры данных. Для каждого экземпляра генерируется соответствующий базисный элемент определения данных, который включается в блок описания глобальных переменных схемы алгоритма. В случае MPI-программы дополнительно генерируются определения переменных myrank (номер текущего процесса) и proc_num (количество процессов). Затем обрабатывается свойство Includes-CompoundOperation. Названия, параметры и начальная алгоритмическая реализация составных операций, перечисленных в этом свойстве, вставляются в схему. Под начальной алгоритмической реализацией понимается последовательность вызовов базисных и составных операторов, перечисленных в свойстве UsesOperation составного элемента (см. раздел 1, табл. 1). В конце генерации в составной оператор main вставляются вызовы, перечисленные в свойстве ProgamCalls.

Пример 2. Далее приведена каркасная схема, сгенерированная в “ИПС” по онтологическому описанию программы, приведенному в примере 1 раздела 1. Для краткости, детализация составного оператора calc_elem_pos не приводится.

```
SCHEME ADR_SORT ====
     “Parallel MPI address sort.”
   END OF COMMENTS

“GlobalData” =
==== “Declare a file variable
(regfile), file name
(sortreg.txt)”;
     “Declare an array
(index_array) of type (int)”;
     “Declare a file variable
(infile), file name
(sort_in.txt)”;
     “Declare a variable (ar-
ray_size) of type (int)”;
```

```
     “Declare an array (indexbuf)
of type (int)”;
     “Declare an array (in_array)
of type (int)”;
     “Declare a file variable
(outfile), file name
(sort_out.txt)”;
     “Declare an array
(out_array) of type (int)”;
     “Declare a variable
(subarray_size) of type (int)”;
     “Declare an array (pointer)
of type (int)”;
     “Declare an array (myrank)
of type (int)”;
     “Declare an array (proc_num)
of type (int)”

“initialize” =
==== “Read array (A) of type (T)
and size (N) from file (F)”
   * “Allocate memory for array
(A) of type (T) and size (size)”

“sorting” =
==== “initialize”
   * “Shift pointer P(k) on (n)
positions in (A) to the right”
   * “calc_elem_pos(k)”
   * “confluence”
   * “Output array (m) of length
(n) to file (name)”
   * “Output the time expired
from the start of processing and
write it to file (name)”

“confluence” =
==== “Collect data (var1) of type
(data_type) with length (n) from
all processes to variable (var2)
of process (proc_rank)”
   * "Assign element (i) the val-
ue of element (j)"

“main” =
==== PARALLEL_MPI(i = 0,...,
proc_num–1)
     (
       “sorting”
     )

END OF SCHEME ADR_SORT
```

В приведенной схеме начало реализации составных операторов отмечено символами “=”. Каждый составной оператор содержит последовательность вызовов операторов в соответствии с его онтологическим описанием (например, схема оператора `sorting` соответствует значениям свойств, представленным в табл. 1). Далее в ДСП-конструкторе необходимо продолжить проектирование алгоритма на основе сгенерированной каркасной схемы. В данном случае необходимо установить параметры базисных элементов и сконструировать алгоритмы функционирования составных операторов, используя присутствующие в схеме вызовы операций, а также выбирая другие операции из списка в ДСП-конструкторе. Полная детализация составных операторов `initialize`, `sorting` и `confluence` будет иметь такой вид (функция `main` остается без изменений):

```
“initialize” =
==== "Read array (in_array) of
type (int) and size (array_size)
from file (infile)"
   * "(subarray_size) := (ar-
ray_size / proc_num)"
   * "Allocate memory for array
(out_array) of type (int) and
size (array_size)"
   * "Allocate memory for array
(indexbuf) of type (int) and size
(subarray_size)"
   * IF 'Number of process = (0)'
     THEN
        "Allocate memory for ar-
ray (index_array) of type (int)
and size (proc_num *
subarray_size)"
     END IF

“sorting” =
==== LocalData
     (
       “Declare a variable
(first_elem) of type (int)”;
       “Declare a variable
(last_elem) of type (int)”
     )
     “initialize”
   * (first_elem := myrank * su-
barray_size)
   * IF ‘Rank of process =
(proc_num – 1)’
     THEN
        “(last_elem := array_size
– 1)”
```

```
     ELSE “(last_elem := (myrank
+ 1) * subarray_size – 1)”
     END IF
   * “Place pointer P(1) at posi-
tion (first_elem)”
   * WHILE NOT ‘Pointer P(1)
reached position (last_elem + 1)
in array (in_array)’
     CYCLE
        “calc_elem_pos(P(1))”
      * “Shift pointer P(1) on
(1) positions in array (in_array)
to the right”
     END OF CYCLE
   * “confluence”
   * IF ‘Rank of process = (0)’
THEN
        “Output array
(out_array) of length
(array_size) to file (outfile)”
       * “Output the time expired
from the start of processing and
write it to file (regfile)”
     END IF

“confluence” =
==== IF ‘Quantity of processes >
(1)’
     THEN
        “Collect data (indexbuf)
of type (MPI_INT) with length
(subarray_size) from all
processes to variable
(index_array) of process (0)”
      * IF ‘Rank of process =
(0)’
        THEN
           FOR ‘(P(2)) from
               (subarray_size) to
               (array_size-1)’
         * CYCLE
              “Assign element
(out_array[index_array[P(2)]])
the value of element
(in_array[P(2)])”
            END OF CYCLE
        END IF
     END IF
```

Выполнение вышеприведенного алгоритма ADR_SORT начинается с функции main, которая запускает proc_num параллельных процессов. Реализация кода каждого процесса находится в функции sorting. Оператор sorting вначале вызывает функцию initialize, которая считывает входной массив (in_array) из файла и резервирует память для массивов, используемых в алгоритме. Затем массив in_array разделяется на proc_num подмассивов, каждый из которых обрабатывается отдельным процессом. Процесс для каждого элемента подмассива, обозреваемого указателем P(1), вычисляет его индекс в отсортированном массиве с помощью вызова функции calc_elem_pos (P(1)). Выходные индексы элементов сохраняются в буферном массиве индексов indexbuf. Далее выполняется функция слияния результатов (confluence), в которой процессы с номерами 1, ..., proc_num–1 передают массив indexbuf в результирующий массив индексов index_array процесса 0. При этом массивы indexbuf размещаются в массиве index_array путем их конкатенации в порядке номеров процессов (с помощью стандартной функции MPI_Gather [15]). После этого процесс 0 осуществляет вставку элементов массива in_array в выходной массив out_array в соответствии с полученными индексами из массива index_array. По схеме adr_sort в “ИПС” была сгенерирована MPI-программа на языке С.

В процессе конструирования схемы алгоритма для выполнения ее автоматизированных преобразований совместно с “ИПС” применяется система переписывающих правил TermWare [17]. Трансформация основывается на применении к алгоритму, представленному в виде терма, систем правил вида

$$f(x_1, ..., x_n) \rightarrow g(x_1, ..., x_n),$$

где f и g – термы; x_i – переменные термов.

В качестве примера выполним трансформацию терма алгоритма

```
Root (
   main (


      serial_exec (
          OutputArray(out_arr),
          OutputTime
      )
   )
)
```

Приведенный терм состоит из функции `main`, выполняющей два оператора, первый из которых выводит массив `out_arr`, а второй выводит время выполнения. (Аналогичная пара операторов присутствует в функции `sorting` вышерассмотренного алгоритма `ADR_SORT`). Пусть упомянутые операторы должны быть включены в блок условного оператора таким образом, чтобы они выполнялись только процессом с нулевым номером. Для этого необходимо к терму алгоритма применить такое правило:

```
main (serial_exec($x1, $x2)) →
main (if (ZeroRankProcess,
serial_exec($x1, $x2)))
```

Результирующий терм будет иметь вид

```
Root (
   main (
      if (ZeroRankProcess,
          serial_exec (
          OutputArray(out_arr),
          OutputTime)
         )
   )
)
```

Заключение

В работе предложен подход к проектированию параллельных программ на основе использования онтологий и аппарата алгебры алгоритмов. Онтологии позволяют описать основные объекты разрабатываемой программы из выбранной предметной области – данные и обрабатывающие их функции, а также взаимосвязи между функциями. Разработан метод генерации каркасной схемы алгоритма по онтологии предметной области. Метод реализован в интегрированном инструментарии проектирования и генерации программ. В инструментарии осуществляется дальнейшее наполнение сгенерированной каркасной схемы с использованием операций САА, а также трансформация алгоритма. По схеме алгоритма затем осуществляется генерация программы на выбранном языке программирования.

К перспективам дальнейшего развития интегрированного инструментария относится их настройка на разработку сервисно-ориентированных Grid программ.

1. *Nistor E.C.* Using Domain Models in Extensible Schema-based Software Synthesis. – NASA Ames Research Center, Technical Report, 2004. – http://www.ics.uci.edu/~enistor/research/nistor-report.pdf
2. *Bureš T., Denney E., Fischer B., Nistor E.C.* The Role of Ontologies in Schema-based Program Synthesis // In Workshop on Ontologies as Software Engineering Artifacts, Vancouver, Canada, 2004. – http://ti.arc.nasa.gov/people/edenney/papers/ontologies.pdf
3. *Waldinger R., Jarvis P., Dungan J.* Program Synthesis for Multi-Agent Question Answering // In International Symposium on Verification (Theory and Practice). – Springer Verlag, Lecture Notes in Computer Science. – 2003. – P. 747–761.
4. *Fensel D., Lausen H., Polleres A., de Bruijn J. et al.* Enabling Semantic Web Services. The Web Service Modeling Ontology. – Springer Verlag, 2007. – 188 p.
5. *de Roure D., Jennings N., Shadbolt N.* Research Agenda for the Semantic Grid: A Future e-Science Infrastructure – Technical report UKeS-2002-02, UK e-Science Technical Report Series, National e-Science Centre, Edinburgh, UK. – 2001. – 78 p.
6. *OMG*, Unified Modeling Language Specification. – http://www.uml.org
7. *Chen P.P.* Entity-Relationship Modelling: Historical Events, Future Trends, and Lessons Learned. – Software pioneers: contributions to software engineering. – Springer-Verlag, New York. – 2002. – P. 296–310.
8. *Андон Ф.И., Дорошенко А.Е., Цейтлин Г.Е., Яценко Е.А.* Алгеброалгоритмические модели и методы параллельного программирования. – Киев: Академпериодика, 2007. – 631 с.
9. *Protégé.* – http://protege.stanford.edu
10. *Protégé-OWL editor.* – http://protege.stanford.edu/overview/protege-owl.html
11. *Глушков В.М., Цейтлин Г.Е., Ющенко Е.Л.* Алгебра. Языки. Программирование. – Киев: Наук. думка, 1989. – 376 с.
12. *Цейтлин Г.Е.* Алгебраическая алгоритмика: теория и приложения // Кибернетика и системный анализ. – 2003. – № 1. – С. 8–18.
13. *Noy N.F., McGuinness D.L.* Ontology Development 101: A Guide to Creating Your First Ontology. – Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880. – 2001. – 25 p.
14. *Чарнецки К., Айзенекер У.* Порождающее программирование: методы, инструменты,

применение. Для профессионалов. – СПб.: Питер, 2005. – 731 с.
15. *MPI*: A Message-Passing Interface Standard. – http://www-unix.mcs.anl.gov/mpi
16. *Дорошенко А.Е., Жереб К.А., Яценко Е.А.* Средства синтеза параллельных MPI- программ // Проблемы программирования. – 2008. – № 2–3. – С. 595–604.
17. *Дорошенко А.Е., Шевченко Р.С.* Система символьных вычислений для программирования динамических приложений // Проблемы программирования. – 2005. – № 4. – С. 718–727.

Получено 01.10.2008

Об авторах:

Дорошенко Анатолий Ефимович,
доктор физико-математических наук,
профессор, заведующий отделом теории компьютерных вычислений,

Яценко Елена Анатольевна,
кандидат физико-математических наук,
старший научный сотрудник.

Место работы авторов:

Институт программных систем
НАН Украины.
Тел.: (044) 526 1538,
e-mail: dor@isofts.kiev.ua,
aiyat@i.com.ua aiyat@i.com.ua