УДК 681.3

*А.Е. Дорошенко, Е.А. Яценко*

# О СИНТЕЗЕ ПРОГРАММ НА ЯЗЫКЕ JAVA ПО АЛГЕБРОАЛГОРИТМИЧЕСКИМ СПЕЦИФИКАЦИЯМ

Рассмотрены средства алгоритмических описаний для представления последовательных и асинхронных алгоритмов для их формализованного проектирования. Предложены алгоритмы диалогового конструирования алгоритмов и генерации программ на целевых языках программирования. Описан метод синтеза многопоточных программ на языке Java по схемам асинхронных алгоритмов.

## Введение

Важной проблемой современного программирования является его математизация, разработка формализованных языков проектирования алгоритмов и программ, а также их абстрактных моделей. К средствам, ориентированным на решение проблем формализации, обоснования правильности, улучшения алгоритмов (по выбранным критериям) принадлежат алгебры алгоритмов. Украинской кибернетической школой внесен значительный вклад в разработку алгебр алгоритмов фундаментальными работами В.М. Глушкова, в русле которых была построена теория систем алгоритмических алгебр (САА) и их модификаций [1]. Аппарат САА стал основой метода многоуровневого структурного проектирования программ (МСПП). Начало инструментальным средствам метода МСПП в свое время было положено созданием системы МУЛЬТИПРОЦЕСИСТ, ориентированной на структурный синтез программ (на языках ПЛ/1, Фортран, Паскаль и др.) по их многоуровневым проектам, оформленным на входном языке проектирования САА/1 [2]. Развитие объектно-ориентированных технологий требует новых подходов к инструментальным средствам программирования, появились эффективные языки проектирования и средства генерации программ в объектно-ориентированных средах (например, язык UML и его инструментарий [3]). С развитием Интернет-технологий возникли языки, ориентированные на разработку Web-приложений. Отметим также, что разработка современного программного обеспечения тесно связана с развитием и использованием моделей параллельных вычислений, которые пронизывают большинство аспектов архитектуры и средств программирования современных компьютерных систем [4–6].

В данной работе описан разработанный интегрированный инструментарий, ориентированный на формализованное проектирование алгоритмов и синтез последовательных и параллельных программ в объектно-ориентированных средах программирования (Java, C++ и др.). Рассмотрены средства распараллеливания и синхронизации параллельных процессов в модифицированных САА. Описана реализация асинхронных алгеброалгоритмических моделей вычислений с общей памятью средствами языка программирования Java с использованием интегрированного инструментария.

## 1. Средства формализованного проектирования программ в системах алгоритмических алгебр

В основу предлагаемых интегрированных инструментальных средств проектирования и генерации программ положен аппарат САА и их модификаций [1]. Модифицированные САА (САА-М) предназначены для формализации процессов мультиобработки, возникающих, в частности, при конструировании программного обеспечения в мультипроцессорных системах.

САА являются двухосновной алгеброй $< U, B; \Omega >$, где $U$ – множество операторов, $B$ – множество логических условий [1]. Операторы представляют собой отображения информационного множества (ИМ) в себя, логические условия являются отображениями множества ИМ в множество значений 3-значной логики $E_3 = \{0, 1, \mu,\}$, где 0 – ложь; 1 – истина; $\mu$ – неопределенность. Сигнатура операций САА $\Omega = \Omega_1 \cup \Omega_2$ состоит из системы $\Omega_1$ логиче-

ских операций, принимающих значения в множестве $B$, и системы $\Omega_2$ операций, принимающих значения в множестве операторов $U$. В САА $< U, B; \Omega >$ фиксируется система образующих $\Sigma$, представляющая собой конечную функционально полную совокупность операторов и логических условий. С помощью этой совокупности посредством суперпозиции операций, входящих в $\Omega$, порождаются произвольные операторы и логические условия из множеств $U$ и $B$. К логическим операциям системы $\Omega_1$ относятся обобщенные булевы операции: дизъюнкция ($\alpha \vee \beta$), конъюнкция ($\alpha \wedge \beta$), отрицание ($\overline{\alpha}$), а также операция $\beta = A\alpha$ левого умножения условия на оператор. К основным операторным операциям системы $\Omega_2$ относятся: композиция $A * B$, альтернатива ($\alpha$-дизъюнкция) ($[\alpha] A, B$), цикл ($\alpha$-итерация) $\{[\alpha] A\}$. Представления в САА $< U, B; \Omega >$ операторов из $U$ посредством суперпозиций, входящих в сигнатуру $\Omega$, получили название регулярных схем (РС) [1].

Сигнатура $\Omega'$ модифицированных САА, кроме перечисленных конструкций, входящих в $\Omega$, содержит операции, предназначенные для представления параллельных вычислений. К этим операциям относится, в частности, асинхронная дизъюнкция $A \dot{\vee} B$ – бинарная операция на множестве $U$, состоящая в асинхронном применении операторов $A$ и $B$. Суперпозиция бинарных операций асинхронной дизъюнкции приводит к производной операции $m$-местной асинхронной дизъюнкции $\dot{\bigvee}_{i=1}^{m} A$, состоящей в параллельном выполнении $m$ ветвей вычислений. Для синхронизации параллельных ветвей в САА-М используются контрольные точки и синхронизаторы.

Контрольные точки представляют собой фиксированные позиции между операторами в схеме [1]. С каждой контрольной точкой $T$ ассоциировано условие $u$, ложное до тех пор, пока процесс вычислений не достиг точки $T$, истинное с момента достижения данной точки и неопределенное при наличии аварийных остановок на пути, ведущем к точке $T$ данной схемы. Условие $u$ называется условием синхронизации, ассоциированным с точкой $T$, которая далее обозначается $T(u)$.

Под синхронизатором понимается цикл $S(u) = \{[u] E\}$, где $u$ – логическая функция, зависящая от условий синхронизации, ассоциированных с некоторыми контрольными точками схемы [1]. Синхронизатор, установленный в некотором месте схемы, осуществляет задержку вычислений в данном месте схемы вплоть до момента, когда его условие синхронизации $u$ (сопряженное с прохождением соответствующих контрольных точек) станет истинным.

С помощью рассмотренных операций формализуется асинхронная мультиобработка [1], состоящая в совместном функционировании нескольких ветвей вычислений, снабженных специальными каналами, реализующими, в случае необходимости, ожидания и обменные взаимодействия между параллельными ветвями. Представления алгоритмов в САА-М, специфицирующие асинхронные операторные взаимодействия, называются параллельными регулярными схемами (ПРС). Отметим, что САА-М также содержит средства, необходимые для описания синхронных вычислений. В работах [1, 5, 7 - 10] аппарат САА-М применен при решении задач символьной мультиобработки: последовательных и параллельных сортировок, поиска, языкового процессирования и др. Проиллюстрируем рассмотренные понятия на примере параллельной адресной сортировки массива.

**Пример 1.** Суть адресной сортировки состоит в вычислении для каждого из элементов входного (сортируемого) массива $M_0$ его индекса (адреса) в выходном массиве $M$. При этом вычисление индекса для каждого элемента входного массива является независимым. В рассматриваемом алгоритме массив $M_0$ длины $n$ разбивается на $m$ подмассивов $M_0(i)$ ($i = 1, ..., m$), которые обрабатываются $m$ параллельными ветвями ($m \geq 1$). Ветвь с номером $i$ осуществляет вычисление выходных индексов элементов подмассива $M_0(i)$ и их вставку в массив $M$. ПРС АДРСОРТ-П$(M_0, M, m)$ адресной сортировки имеет следующий вид:

$$\text{АДРСОРТ - П}(M_0, M, m) ::= \text{СТАРТ} * (\bigvee_{i=1}^{m} \text{АДР\_ВЕТВЬ}(i)) * S(\text{ОБР\_ЗАК}) * \text{ФИН},$$

$$\text{АДР\_ВЕТВЬ}(i) ::= \text{ВЫЧ\_ГР}(i, m_1, m_2) * (j = m_1) *$$
$$* \{[j > m_2] \text{АДР\_ВСТАВ}(j) * (j = j + 1)\} * T(\text{ОБР\_ЗАК}(i)),$$

где СТАРТ – оператор инициализации данных (ввод сортируемого массива); АДР_ВЕТВЬ($i$) – ветвь, осуществляющая вставку элементов подмассива $M_0(i)$ в выходной массив $M$; ВЫЧ_ГР($i$, $m_1$, $m_2$) – вычисление границ ($m_1$, $m_2$) подмассива $M_0(i)$; АДР_ВСТАВ($j$) – составной оператор, вычисляющий индекс $j$-го элемента массива $M_0$ ($j$ = 1, ..., $m$) в выходном массиве $M$, и выполняющий вставку этого элемента в массив $M$; $T$(ОБР_ЗАК($i$)) – контрольная точка для фиксации момента завершения обработки в $i$-й ветви; $S$(ОБР_ЗАК) – синхронизатор по условию достижения контрольных точек во всех $m$ параллельных ветвях; ФИН – заключительный оператор (вывод отсортированного массива).

Смысл приведенного алгоритма состоит в параллельном функционировании $m$ ветвей вычислений АДР_ВЕТВЬ($i$), каждая из которых осуществляет вставку элементов подмассива $M_0(i)$ в массив $M$ с помощью оператора АДР_ВСТАВ($j$). В операторе АДРСОРТ - П($M_0$, $M$, $m$) находится синхронизатор $S$(ОБР_ЗАК), выполняющий задержку вычислений до тех пор, пока все ветви не закончат обработку. После завершения выполнения синхронизатора имеем отсортированный массив $M$. Более подробно метод параллельной адресной сортировки описан в [7].

## 2. Разработка приложений в интегрированном инструментарии проектирования и синтеза программ

В данном разделе рассматривается процесс автоматизированного проектирования и генерации программ по формализованным спецификациям в САА-М с использованием разработанного в [11] интегрированного инструментария. Инструментарий основывается на методе диалогового конструирования синтаксически правильных программ [9] и использует различные формы представления алгоритмов в процессе их проектирования – естественно-лингвистическую (САА-схемы), алгебраическую (регулярные схемы) и граф-схемную. Отметим, что интегрированный инструментарий может быть настроен на требуемый входной язык, представленный в алгебре алгоритмов, и выходной (целевой) язык программирования. В настоящее время он поддерживает генерацию последовательных и параллельных (многопоточных) программ на языке Java по САА-схемам алгоритмов. Для синтеза программ совместно с интегрированным инструментарием используются средства визуализированного проектирования программ – язык UML и система Rational Rose. Интегрированный инструментарий состоит из следующих основных компонентов:

- ДСП-конструктора, предназначенного для диалогового проектирования синтаксически правильных схем алгоритмов и синтеза программ;
- редактора граф-схем;
- трансформатора, осуществляющего диалоговое преобразование регулярных схем;
- генератора САА-схем по распределенным гиперсхемам;
- среды конструирования алгеброалгоритмических описаний (СКАО) [10] символьной мультиобработки, содержащей описание конструкций САА-М, базисных понятий и их программных реализаций; метаправила конструирования алгоритмов, схемы алгоритмов, стратегии обработки и гиперсхемы.

**2.1. Диалоговое конструирование схем алгоритмов.** В основу функционирования ДСП-конструктора положен диалоговый режим с использованием меню подстановок и дерева конструирования алгоритма (рис. 1). Меню состоит из конструкций САА-М, суперпозиция которых позволяет создавать алгоритмы в упомянутых ранее

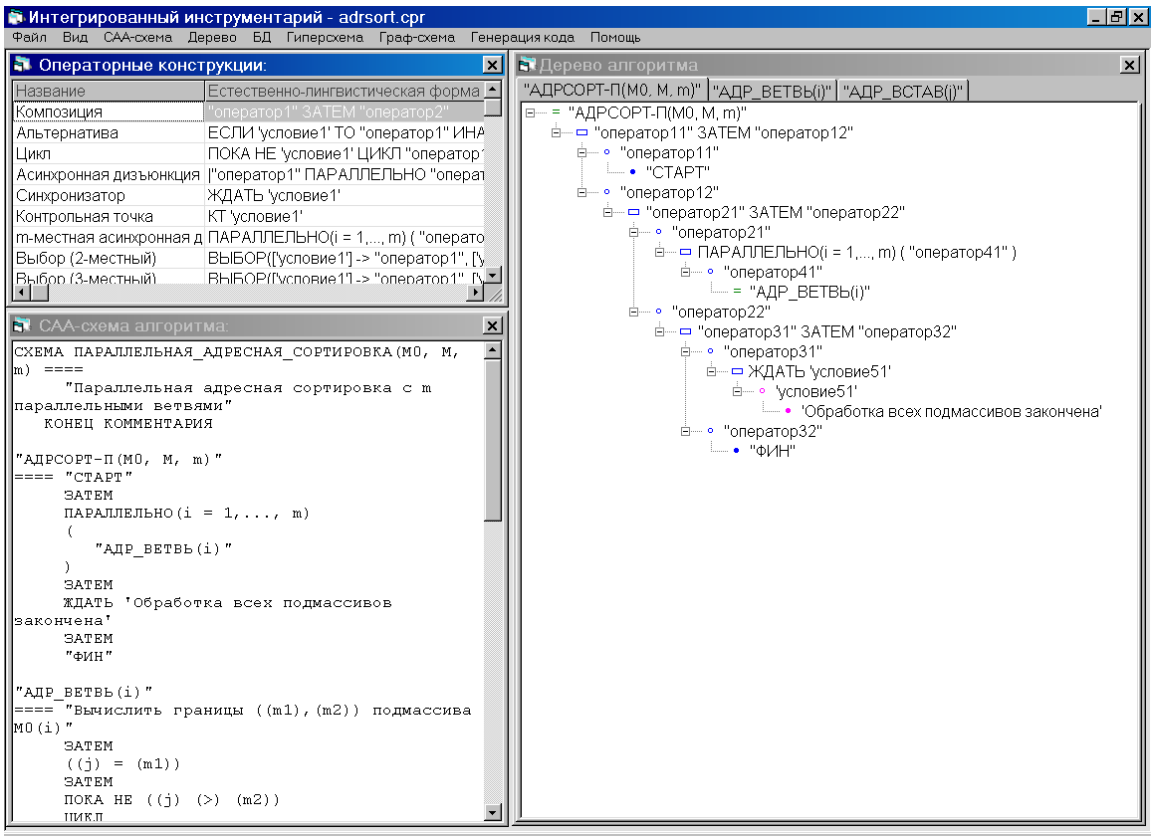


Рис. 1. Окно ДСП-конструктора с параллельным алгоритмом адресной сортировки

формах. Выбранные пользователем конструкции, а также операторные и логические переменные, входящие в них, отображаются в дереве с дальнейшей детализацией переменных. В зависимости от типа выбранной переменной система предлагает соответствующий список операций САА-М или базисных понятий из СКАО. Отметим поуровневый стиль конструирования алгоритма, а также возможность переходов на различные уровни (узлы дерева) с продолжением процесса диалогового конструирования.

Процесс построения дерева конструирования $T$ для некоторого алгоритма может быть представлен следующей РС:

$$\text{ДСП\_НАБОР}(T) = \text{ИНИЦ}(T) * \\ *\{[\text{КОНЕЦ\_ДСП}(T)]\ \text{ВНН} * \text{ВНК} * \\ * \text{ОБРАБОТКА\_УЗЛА}(T) * \text{ВЫВОД}\},$$

где ИНИЦ($T$) – формирование корневой вершины дерева $T$ с названием первого составного оператора, а также его дочернего узла, помеченного именем операторной переменной; КОНЕЦ_ДСП($T$) – условие, истинное, если все переменные конструируемой схемы проинтерпретированы; ВНН – оператор выбора нужной непроинтерпретированной переменной в дереве $T$; ВНК – выбор нужной конструкции САА-М, базисного или составного элемента; ОБРАБОТКА_УЗЛА($T$) – оператор, который формирует дочернюю вершину для текущего выбранного узла дерева и помечает ее текстом выбранного элемента меню, а также формирует вершины дерева с названиями операторных или логических переменных (если выбранный элемент – операция САА). В процессе ДСП конструирования операции ВНН и ВНК выполняются пользователем. По полученному текущему дереву конструирования $T$ осуществляется вывод текста схемы алгоритма с помощью оператора ВЫВОД. Разработанная схема ДСП_НАБОР($T$) подобна схеме ДСП.БС, приведенной в [9]. В схеме ДСП.БС осуществлялась обработка переменных конструированной схемы лишь в одном направлении (слева направо) и выполнялся последовательный переход сверху вниз от предыдущего уровня дерева конструирования к следующему. В отличие от упомянутой схемы, в построенной РС ДСП_НАБОР($T$) есть возможность выпол-

нять конкретизацию переменных конструируемой схемы в произвольном порядке, переходя на различные уровни дерева *T*.

Пример дерева конструирования показан на рис. 1 в окне ДСП-конструктора. Отметим, что в ДСП-конструкторе дерево для каждого из составных операторов схемы алгоритма приводитя на отдельной вкладке окна “Дерево алгоритма”. Текст САА-схемы, соответствующий дереву конструирования, отображается в отдельном текстовом окне в аналитической или естественно-лингвистической форме в зависимости от установленной опции (левое нижнее окно в окне ДСП-конструктора).

рованного кода. Проектирование классов программы и их взаимосвязей, а также генерация каркасного программного кода выполняется с помощью инструментария Rational Rose. В файл, полученный в результате моделирования в Rational Rose, выполняется подстановка кода, сгенерированного в предлагаемом инструментарии.

В табл. 1 приведено описание в СКАО [10] основных операторных операций САА (композиции, альтернативы, цикла), реализации которых на языке Java используются в процессе генерации программ. Приведенные реализации содержат строки “^оператор1^” и “^оператор2^”, вместо которых процессе синтеза будут подставлены тексты реализаций операндов этих операций.

Таблица 1. Описание в СКАО операторных операций САА

| Аналитическая форма | Естественно-лингвистическая форма | Реализация на языке Java |
|---|---|---|
| “оператор1” * “оператор2” | “оператор1” ЗАТЕМ “оператор2” | `^оператор1^;`<br>`^оператор2^` |
| ([‘условие1’] “оператор1”, “оператор2”) | ЕСЛИ “условие1”<br>ТО “оператор1”<br>ИНАЧЕ “оператор2”<br>КОНЕЦ ЕСЛИ | `if (^условие1^)`<br>`{`<br>`  ^оператор1^`<br>`}`<br>`else { ^оператор2^ }` |
| {[‘условие1’] “оператор1”} | ПОКА НЕ ‘условие1’<br>ЦИКЛ<br>“оператор1”<br>КОНЕЦ ЦИКЛА | `while (!(^условие1^))`<br>`{`<br>` ^оператор1^`<br>`}` |

**2.2. Синтез программ по схемам алгоритмов.** По полученному в процессе конструирования алгоритма дереву, реализациями элементарных операторов и условий на целевом объектно-ориентированном языке программирования, а также другими фрагментами, ДСП-конструктор выполняет синтез программы. В процессе синтеза управляющие конструкции САА-схемы алгоритма (асинхронная дизъюнкция, композиция, альтернатива, цикл и др.) отображаются в соответствующие операторы языка программирования, а вместо базисных операторов и предикатов подставляются их реализации на этом же языке из СКАО. Составные операторы могут быть представлены как подпрограммы (методы). На вход синтезатора поступает также файл, который содержит каркасное описание основного класса программы (без реализаций методов), в который выполняется подстановка синтези-

В табл. 2 приведены примеры описания в СКАО базисных элементов для алгоритмов сортировки. Естественно лингвистическая и аналитическая формы описания базисного элемента содержат в себе имена формальных параметров в скобках. Фактические параметры, которые задаются в САА-схемах, заменяют при синтезе программы соответствующие формальные параметры в тексте реализации базисного понятия на языке программирования. Реализации базисных элементов написаны с использованием компонента повторного использования – класса Sorting, предназначенного для решения различных задач сортировки (более подробно этот класс рассматривается далее в примере 2). Признаком формального параметра в реализации

является символ “%”, за которым следует порядковый номер параметра в тексте базисного оператора или предиката. Например, в базисном операторе “Переставить *l, r* по У(*i*) в (М)” присутствует параметр *i* – номер указателя. Значение этого параметра будет подставлено вместо соответствующего формального параметра в реализации этого оператора, который имеет вид: s.transp(%1), где *s* – экземпляр класса Sorting; transp – метод данного класса, выполняющий перестановку элементов массива М, соседних с указателем. Реализация других базисных элементов, приведенных в табл. 2, также являются вызовами методов класса Sorting. Отметим, что в процессе генерации программы перед текстом реализации того или иного базисного элемента добавляется его название в виде комментария.

жащего каркасный программный код, и имя результирующего файла генерации; параметры подстановки сгенерированного кода во входной файл. Код может быть подставлен вместо указанной цепочки символов в файле (например, “// <target>”) или в тело указанной в параметрах функции, которая описана во входном файле. В случае языка Java код обычно подставляется в метод main. В параметрах генерации для составных операторов и предикатов для каждого из них можно задать имя метода, в тело которого будет подставлен сгенерированный для них код на языке программирования.

Генерация кода в ДСП-конструкторе осуществляется по дереву конструирования, все терминальные вершины которого являются базисными операторами или предикатами (т.е. все переменные сконструирован-

Таблица 2. Описание в СКАО базисных операторов и предикатов для задач сортировки

| Тип | Естественно-лингвистическая форма | Аналитическая форма | Реализация на Java |
|---|---|---|---|
| Операторы | “Переместить У(*i*) на (*n*) по (М) вправо” | *P*(У(*i*), (*n*)) | `s.moveR(%1, %2);` |
| | “Переставить *l, r* по У(*i*) в (М)” | ТРАНСП(*l, r*\| У(*i*)) | `s.transp(%1);` |
| | “Установить У(*i*) на У(*j*) в (М)” | УСТ(У(*i*), У(*j*)) | `s.place(%1, s.getPointerPos(%2));` |
| Предикаты | ‘*l* > *r* по У(*i*) в (М)’ | *l* > *r*\| У(*i*) | `s.compare(%1, “>”)` |
| | ‘Расстояние между У(*i*) и У(*j*) в (М) равно (*n*)’ | *d*(У(*i*), У(*j*)) = (*n*) | `s.distance(%1, %2) == %3` |
| | ‘Указатель У(*i*) в конце (М)’ | *d*(У(*i*), К) | `s.atEnd(%1)` |

Отображение в языке программирования для любого из элементов СКАО может содержать несколько вариантов, один из которых можно выбрать в зависимости от решаемой задачи. Например, базисный оператор инициализации (СТАРТ) имеет различные варианты реализации на языке программирования для сортировки и поиска.

В ДСП-конструкторе есть возможность указать параметры генерации программы по схеме алгоритма. Данные параметры разделены на две группы: общие параметры и параметры генерации для составных элементов схемы. Общие параметры включают имя входного файла, содер-

ной схемы проинтерпретированы). В процессе генерации обход дерева конструирования осуществляется в направлении слева направо. Обработка начинается с нижнего листа самой левой ветви дерева, затем обрабатываются все листья и узлы дерева так, что ветви рассматриваются слева направо, а узел обрабатывается лишь после обхода всех ветвей, которые исходят из него. Отметим, что такой обход дерева обычно используется для получения постфиксной записи при трансляции выражений [12]. Обработка каждой вершины дерева конструирования осуществляется в зависимости от ее типа, а именно: базисное понятие, переменная, имя составного элемента или операция САА.

Если текущий элемент дерева – базисный оператор или предикат, то осуществляется поиск реализации данного элемента на языке программирования в СКАО. Если найденная реализация содержит несколько возможных вариантов, то из них выделяется вариант, специально указанный пользователем для текущего проекта или вариант по умолчанию (в случае, если вариант не был выбран пользователем). Затем осуществляется подстановка в реализацию значений фактических параметров базисного понятия. Далее перед полученной реализацией добавляется текст базисного элемента схемы в виде комментария и полученный текст записывается в стек. Если текущая вершина – операция САА, то в ее реализацию вместо строк, которые соответствуют именам переменных, осуществляется подстановка реализаций ее операндов, которые последовательно изымаются из стека. Полученный текст записывается в стек. Если текущий обрабатываемый элемент дерева – переменная некоторой операции САА, то вершина пропускается. Текущая вершина дерева также может быть именем составного оператора или предиката, который соответствует отдельному поддереву конструирования. В этом случае в стек записывается текст реализации данного элемента, полученный при обработке его дерева конструирования и сохраненный в специальной таблице. Перед текстом составного элемента добавляется комментарий с информацией о том, что в данной части программы начинается код, соответствующий составному оператору схемы алгоритма. Содержимое вершины стека после обработки корневой вершины дерева конструирования является результирующим программным кодом, который соответствует САА-схеме алгоритма. Полученный текст вставляется в требуемые позиции файла с каркасным программным кодом, поступающего на вход генератора программ перед началом генерации.

**2.3. Синтез параллельных многопоточных программ.** Асинхронные алгоритмы, представленные в САА-М, в таких языках программирования, как Java, C++, Perl, Python, Delphi, могут быть реализованы с помощью подпроцессов или потоков (threads). Поток в многопоточных операционных системах – наименьшая единица выполнения. Для каждого процесса (объекта, создаваемого при запуске программы) ОС создает один главный поток, который является потоком команд центрального процессора, которые выполняются поочередно. При необходимости главный поток может создавать другие потоки, пользуясь для этого программным интерфейсом ОС. Все потоки, созданные процессом, выполняются в адресном пространстве этого процесса и имеют доступ к его ресурсам. Если процесс создал несколько потоков, то все они выполняются параллельно, причем время центрального процессора (или нескольких центральных процессоров в мультипроцессорных системах) распределяется между этими потоками. Каждому потоку дается определенный интервал времени, на протяжении которого он находится в активном состоянии.

Текущая версия интегрированного инструментария обеспечивает синтез параллельных программ на языке программирования Java [13]. Отметим, что Java поддерживает многопоточность не только на уровне библиотек, но и на уровне самого языка, который значительно облегчает построение программ, которые надежно работают в многопоточном режиме. Java предоставляет возможность реализации подпроцессов с помощью класса Thread (из пакета java.lang), который содержит все средства, необходимые для создания потоков, управления их состоянием и синхронизации. При этом для организации многопоточных программ существует две возможности: первая предусматривает создание подкласса класса Thread, вторая – создание класса, который реализует интерфейс Runnable. В этих классах переопределяется метод run, который содержит код, предназначенный для выполнения в рамках отдельного потока. Отметим, что метод main в программах Java по умолчанию является подпроцессом, и из него или из другого активного метода, могут начинать свое выполнение другие подпроцессы.

Потоки, работающие параллельно в многопоточной системе, могут обращаться одновременно к общим объектам в памяти, что может привести к неправильной работе

программ. Решением этой проблемы является синхронизация потоков. Java обеспечивает два уровня синхронизации:

- защита совместно используемых ресурсов;
- сигнализация об изменениях в состояниях между процессами.

Для синхронизации фрагмента кода, осуществляющего доступ к разделяемым ресурсам, этот фрагмент включается в блок или метод, помеченный модификатором synchronized. Когда поток начинает выполнение синхронизированного фрагмента кода, этот фрагмент блокируется. До окончания выполнения синхронизированного фрагмента либо до тех пор, пока блокировка не будет явно отменена, ни один другой поток не может начать выполнение аналогичного фрагмента кода.

Другим важным аспектом процесса синхронизации в Java является передача подпроцессу сообщения, указывающего, что на данный момент уже выполнено условие, исполнения которого он “ждет” [13]. Если подпроцесс в определенный момент вычислений не может продолжать работу, так как объект, с которым он связан, не находится в надлежащем состоянии, он может вызвать метод wait, который переведет этот подпроцесс в состояние ожидания. Ожидающий поток может быть вновь активирован в случае, если другой поток вызовет для него метод notify или notifyAll. Метод notify возобновляет работу одного потока, а метод notifyAll активирует все потоки, ожидающие снятия блокировки, реализованной с помощью некоторого объекта. Как и вызов метода wait, вызовы методов notify и notifyAll могут быть исходить только из синхронизированного блока.

Методы wait, notify и notifyAll использованы в интегрированном инструментарии для реализации средств синхронизации САА-М (контрольных точек и синхронизаторов), рассмотренных в разделе 2.

**Пример 2.** В качестве иллюстрации рассмотрим реализацию многопоточной программы на языке Java, сгенерированной по регулярной схеме параллельной адресной сортировки (см. пример 1) с использованием интегрированного инструментария и системы Rational Rose. На рис. 2 показана диаграмма классов UML с основными классами разрабатываемого приложения. Класс Adrsort – основной класс программы адресной сортировки. Выполнение программы начинается с метода main данного класса, в котором вводится сортируемый массив, вызываются $m$ параллельных потоков, осуществляется синхронизация (ожидание завершения вычислений во всех ветвях) и выводятся результаты сортировки. В методе ins находится реализация составного оператора АДР_ВСТАВ($j$) схемы АДРСОРТ - $\Pi(M_0, M, m)$. Данный метод в процессе сортировки вызывается параллельными потоками. В классе SortThread, наследующем класс Thread, находится описание кода для $i$-го параллельного потока ($i = 1, \ldots, m$). Классы Adrsort и SortThread используют методы изображенного на диаграмме класса Sorting – повторно используемого компонента для решения различных задач сортировки (пузырьковой, челночной, адресной и др.). Этот класс содержит описание данных – обрабатываемого массива, указателей, маркеров, контрольных точек, методы доступа к этим данным, а также методы остановки и возобновления работы главного потока и отсчета времени, истекшего с начала сортировки.

В Rational Rose для класса Adrsort была выполнена генерация файла с каркасным программным кодом, не содержащим реализаций методов main и ins. Дальнейшая разработка алгоритмов этих методов, а также синтез и подстановка кода, реализующего эти функции, выполнены в интегрированном инструментарии. Генерация класса SortThread полностью осуществлена в ДСП-конструкторе.

В процессе синтеза кода программа ДСП-конструктора использует шаблоны программных реализаций конструкций САА и базисных понятий из СКАО, примеры которых были приведены выше в табл. 1 и 2. Далее рассмотрены примеры программных реализаций операций, предназначенных для формализации параллельных вычислений: асинхронной дизъюнкции, синхронизатора, контрольных точек. Приводится естественно-лингвистическая форма представления каждой из этих операций и их отображение в текстовый шаблон на языке Java.

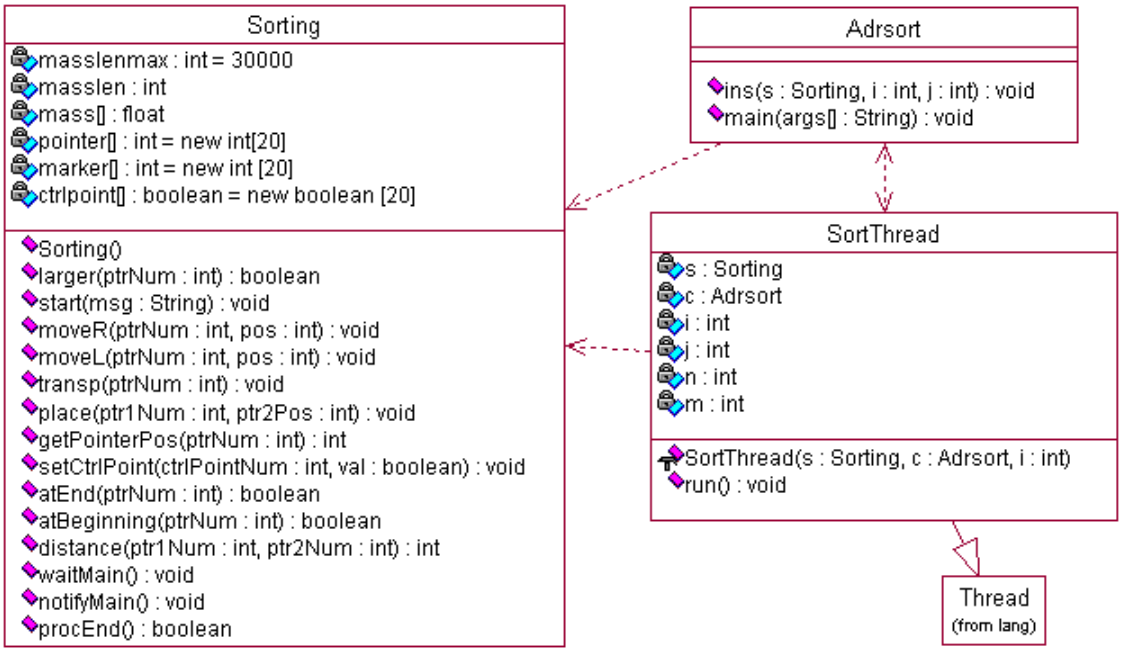


Рис. 2. Диаграмма классов для параллельной адресной сортировки

В СКАО интегрированного инструментария для реализации операции ***m*-местной асинхронной дизъюнкции** $\overset{m}{\underset{i=1}{\dot{\vee}}} A$ (естественно-лингвистическая форма этой операции – ПАРАЛЛЕЛЬНО ($i = 1, ..., m$) (“оператор1”)) для рассматриваемой адресной сортировки был выбран следующий вариант шаблона на языке Java:

```
int m = s.getThreadNum();
for (int i = 1; i <= m; i++)
{
  SortThread st = new SortThread(s, c, i);
}

// $classes

class SortThread extends Thread
{
 Sorting s;
 Adrsort c;
 int n, m, i, j;

 /* Конструктор класса */
 SortThread(Sorting s, Adrsort c, int i)
 {
  this.s = s;
  this.c = c;
```

```
  this.i = i;
  n = s.getDataLen();
  m = s.getThreadNum();
  new Thread(this).start(); /* Запустить поток на выполнение */
 }

 public void run()
 {
  ^оператор1^
  }
}
```

Приведенный фрагмент программной реализации разделен на две части строкой “// $classes”. В первой части описан цикл, в котором создаются m параллельных потоков (объектов класса SortThread). Во второй части фрагмента приведен класс SortThread (наследующий класс Thread), в котором описан шаблон класса потока (без реализации метода run()). Конструктор класса SortThread выполняет инициализацию данных и запуск подпроцесса с помощью метода start(). Через параметры этому конструктору передается объект класса Sorting, экземпляр основного класса программы и номер i создаваемого потока. Метод run класса SortThread содержит строку “^оператор1^”, вместо которой в процессе генерации будет подставлен код реализации операнда асинхронной дизъюнкции. В результате синтеза в том месте программы Java, которое соответствует асинхронной дизъюнкции, будет приведен цикл создания потоков. Текст класса SortThread (с текстом реализации *i*-й ветви алгоритма) будет включен после текста основного класса программы. Вместо строк “This_class”, приведенных в данном фрагменте, в процессе синтеза подставляется имя основного класса, указанное в параметрах генерации кода (например, Adrsort – основной класс программы адресной сортировки).

Для реализации на языке Java операции **синхронизатора** (естественно-лингвистическая форма этой операции – ЖДАТЬ ‘условие1’) для рассматриваемого алгоритма сортировки в СКАО был выбран следующий вариант шаблона:

```
if (! (^условие1^))
s.waitMain(); // Приостановить работу
                главного потока
```

Здесь waitMain – метод класса Sorting, выполняющий задержку вычислений в главном потоке программы, в случае, если условие ^условие1^ не выполнено. Вместо строки “^условие1^” для рассматриваемого алгоритма в процессе генерации будет подставлен вызов метода s.procEnd(), который возвращает истинное значение в случае, если все *m* параллельных ветвей завершили вычисления, и ложное в противном случае. Описание метода waitMain в классе Sorting имеет вид

```
public void waitMain()
{
  notified = false; // Установить признак того,
                    // что поток mainThread находится
                    // в состоянии ожидания
  synchronized (mainThread)
```

```
  {
   try { // Приостановить поток mainThread
        mainThread.wait();
       }
   catch (InterruptedException e) { e.printStackTrace(); }
  }
}
```

Здесь mainThread – переменная класса Thread, содержащая информацию о главном потоке выполняющейся программы. Поток mainThread в приведенном фрагменте переведен в состояние ожидания с помощью вызова стандартного метода wait().

Для реализации на языке Java оператора **контрольной точки** (естественно-лингвистическая форма этого оператора – КТ ‘условие1’) для случая рассмотренной адресной сортировки в СКАО был выбран следующий вариант шаблона:

```
^условие1^
if (s.procEnd())
s.notifyMain(); // Возобновить работу главного потока
```

Вместо строки “^условие1^” для рассматриваемого алгоритма в процессе генерации будет подставлена программная реализация базисного элемента ОБР_ЗАК(*i*), а именно, вызов метода s.setCtrlPoint(i, true) класса Sorting. Этот метод присваивает контрольной точке с номером i (соответствующим номеру потока) истинное значение. После выполнения этого метода проверяется истинность всех контрольных точек, ассоциированных с подпроцессами, с помощью вызова упомянутой выше функции s.procEnd(). После завершения вычислений во всех подпроцессах (s.procEnd() возвратила истинное значение) выполняется метод notifyMain() класса Sorting, возобновляющий работу главного потока. Описание этого метода имеет вид

```
public void notifyMain()
{
 if (! (notified))
 {
   synchronized (mainThread)
   {
        /* Возобновить работу потока mainThread */
          mainThread.notify();
          // Установить признак того,
          // что работа потока mainThread возобновлена
          notified = true;
   }
 }
}
```

В приведенной функции работа потока mainThread возобновляется с помощью вызова стандартного метода notify().

Таким образом, по асинхронным алгоритмам, представленным в САА-М, может быть синтезирован вручную или автоматизированным способом (с применением разработанного интегрированного инструментария) код на языках программирования, которые поддерживают создание многопоточных программ.

## Заключение

В работе рассмотрены средства формализованного проектирования и синтеза последовательных и асинхронных программ по их представлениям в алгебрах алгоритмов. Предложены алгоритмы диалогового конструирования алгоритмов и генерации программ на целевых языках программирования. Описан метод синтеза многопоточных программ на языке Java по асинхронным алгоритмам, формализованным в САА-М.

К ограничениям интегрированного инструментария можно отнести то, что в схемах алгоритмов не учитывается объектная ориентированность генерируемой программы. Она присутствует лишь на уровне программных реализаций элементов схемы в среде конструирования алгеброалгоритмических описаний инструментария, а также в каркасном файле, поступающим на вход синтезатора. Тем не менее, данная проблема может быть решена в дальнейшем путем настройки инструментария на объектные САА-схемы [14, 15].

К перспективам развития интегрированного инструментария следует отнести его настройку на синтез параллельных программ, использующих технологию MPI.


1. *Глушков В.М., Цейтлин Г.Е., Ющенко Е.Л.* Методы символьной мультиобработки. – Киев: Наук. думка, 1980. – 252 с.
2. *Многоуровневое* структурное проектирование программ: Теоретические основы, инструментарий // Е.Л. Ющенко, Г.Е. Цейтлин, В.П. Грицай, Т.К. Терзян. – М.: Финансы и статистика, 1989. – 208 с.
3. *Кватрани Т.* Rational Rose 2000 и UML. Визуальное моделирование / Пер. с англ. – М.: ДМК, 2001. – 176 с.
4. *Дорошенко А.Е.* Математические модели и методы организации высокопроизводительных параллельных вычислений. Алгебродинамический подход. – Киев: Наук. думка, 2000. – 176 с.
5. *Дорошенко А.Ю., Фінін Г.С., Цейтлін Г.О.* Алгеброалгоритмічні основи програмування. — К.: Наук. думка, 2004. — 458 с.
6. *Дорошенко А.Е., Цейтлин Г.Е.* Алгеброавтоматные спецификации параллельных программ над общей и распределенной памятью // Проблемы программирования. – 2003. – № 3. – С. 24–33.
7. *Яценко Е.А.* Регулярные схемы алгоритмов адресной сортировки и поиска // УСиМ. – 2004. – № 5. – С. 61 – 66.
8. *Цейтлин Г.Е.* Проектирование алгоритмов параллельной сортировки // Программирование. – 1989. – № 6. – С. 4–19.
9. *Цейтлин Г.Е.* Введение в алгоритмику. – Киев: Сфера. – 1998. – 310 с.
10. *Яценко О.А.* Середовище конструювання алгоритмічних знань та інструментарій синтезу програм // Проблемы программирования. – 2006. – № 1–2. – С. 349–358.
11. *Яценко Е.А., Мохница А.С.* Инструментальные средства конструирования синтаксически правильных параллельных алгоритмов и программ // Проблемы программирования. – 2004. – № 2–3. – С. 444–450.
12. *Ахо А., Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. – М.: Мир, 1973. – 1. – 612 с.
13. *Бишоп Д.* Эффективная работа: Java 2. – Киев: BHV, 2002. – 592 с.
14. *Цейтлин Г.Е., Теленик С.Ф., Амонс А.А.* Алгебро-логическая формализация в объектно-ориентированных технологиях // Проблемы программирования. – 2002. – № 1–2. – С. 136–146.
15. *Doroshenko A.E., Ragozin D.V.* Retargetable and tuneable code generation for high performance DSP // “Parallel Computing Technologies”, Proc. 7-th Int. Conf. PaCT’2003, LNCS, 2003. – 2763. – P. 452–466.


Получено 01.09.2006

***Об авторах:***

*Дорошенко Анатолий Ефимович,*
доктор физ.-мат. наук, профессор,
зав. отделом теории компьютерных вычислений,

*Яценко Елена Анатольевна,*
кандидат физ.-мат. наук,
научный сотрудник.

***Место работы авторов:***

Институт программных систем
НАН Украины,
проспект Академика Глушкова, 40.
03680, Киев-187, Украина.
тел. (044) 526 1538
e-mail: dor@isofts.kiev.ua, aiyat@i.com.ua