УДК 681.3

СРЕДСТВА ПРОЕКТИРОВАНИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ НА ОСНОВЕ АЛГЕБРЫ АЛГОРИТМИКИ

А.Е. Дорошенко, В.А. Иовчев

Институт программных систем НАН Украины, 03680, Киев-187, проспект Академика Глушкова, 40.
Тел. (044) 5261538 E-mail: dor@isofts.kiev.ua, iovchev.v@gmail.com

Рассмотрены средства алгоритмических описаний для представления последовательных и параллельных алгоритмов. Описаны методы диалогового конструирования алгоритмов и синтеза объектно-ориентированных программ. Рассмотрен разработанный инструментарий на основе данных методов. Описан метод синтеза последовательных и параллельных программ на языке Java.

Means for representation of sequential and parallel programs based on algorithmic description reviewed. Methods for dialog design and synthesis of parallel and object-oriented programs in Java described. Instrumental toolkit based on aforementioned methodology presented.

Введение

Одним из важных объектов исследования алгебраической алгоритмики (АА) является направление математизации современного программирования, разработки формальных языков проектирования алгоритмов и программ и развитие методов построения абстрактных моделей. Все языки программирования построены на абстракциях [1, 2].

Тенденции развития современных языков программирования все чаще и чаще сковывает основную абстракцию решения поставленной задачи. Процесс проектирования решения несет больше характер размышлений о структуре вычислительной среды, а не о решении задачи.

В процессе проектирования решения устанавливается связь между моделью вычислительной среды (пространство решений) и моделью задачи (пространство задачи). Очень часто все данные о процессе установления данной связи либо неформально оформлены, либо несут слабое отражение всех стадий проектирования. Что вызывает ряд проблем для стороннего разработчика. В результате появляются программы сложные в понимании и тяжелые в поддержании.

Так альтернативой моделированию вычислительной среды стало моделирование решаемой задачи [1, 2].

Объектно-ориентированный подход является одной из альтернатив [1 – 3], предоставляя разработчику средства для представления задачи в ее пространстве. Общий характер данного подхода не накладывает ограничений на тип решаемой задачи. Главное преимущество состоит в адаптации к специфике задачи посредством создания новых типов объектов, которые несут в себе описание решения подзадач. Данная мощная и гибкая абстракция, позволяет объектно-ориентированному программированию (ООП) описывать задачу в контексте самой задачи, а не в контексте вычислительной среды.

В процессе проектирования ООП решения возникают две взаимосвязанные задачи [4]:

- построение состава частей для решения подзадач, при этом специфицируются промежуточные и результирующие данные;
- специфицируются взаимосвязи между частями, образую общий алгоритм.

На данном этапе проектирования возникают проблема достаточной формализации и наличие четких переходов от неконкретизированных абстракций до реализаций с привязкой к модели вычислительной среды. Как решение данной проблемы в [1] предложен метод проектирования абстрактного типа данных с последующим этапом перехода в алгебраический класс. В качестве формального аппарата, для проектирования алгоритмов выбрана система алгоритмических алгебр (САА) В.М. Глушкова и метод многоуровневого структурного проектирования программ (МСПП) [5, 6]. Следует отметить, что украинской кибернетической школой внесен значительный вклад в разработку и развитие алгебр алгоритмов и их модификаций. А так же положило начало инструментальным средствам, ориентированным на синтез алгоритмов и программ (МУЛЬТИПРОЦЕССИСТ, ДСП конструктор, трансформатор регулярных схем) [7 – 10].

Данная работа описывает разработанный инструментарий, ориентированный на формализованное проектирование абстрактного типа данных, перехода в алгебраический класс, конкретизацию его и синтез объектно-ориентированного исходного кода. К преимуществам данного инструментария следует отнести полную спецификацию процесса построения и возможность расширения инструментария им же самим, т. е. путем «самораскрутки».

Приведен краткий обзор основополагающих инструментальных средств. Проиллюстрированы примеры разработки как построение части инструментария, так и параллельной сортировки.

1. Средства формализованного проектирования программ в системах алгоритмических алгебр

Процесс проектирования программ, использующий метод МСПП являет собой многоуровневую трансформацию постановки задачи в программное решение. Начало инструментальных средств, данного метода, было положено разработкой открытой системы МУЛЬТИПРОЦЕССИСТ, ориентированной на структурный синтез программ (на языках ПЛ/1, Фортран, Паскаль и др.) по их многоуровневым проектам, оформленным на входном языке проектирования САА/1 [7, 8]. С развитием технологий, таких как объектно-ориентированные и Internet, возникли новые потребности (RIA приложения) и новые языки (например, язык UML и его инструментарий [7]). Следует отметить, что разработка современного программного решения тесно связанна с развитием и применением моделей параллельных вычислений, которые пронизывают большинство аспектов архитектуры и средств программирования современных компьютерных систем [7, 9, 10].

Данные потребности привели к появлению следующему поколению инструментариев. Таких как, интегрированный инструментарий проектирования и синтеза классов алгоритмов и программ [1]. Данные инструментальные средства используют метод диалогового конструирования синтаксически правильных программ (ДСП-метод) [7, 9, 10], ориентированный не на поиск и исправление ошибок, а на исключение возможности их появления в процессе построения алгоритмов и программ. Инструментарий позволяет строить синтаксически правильные САА-схемы алгоритмов и выполнять синтез программ. Важным отличием инструментария от системы МУЛЬТИПРОЦЕССИСТ состоит в интеграции всех трех представлений алгоритма [7]:

- аналитического (формула в избранной алгебре);
- естественно-лингвистического или текстовом (САА-схемы);
- графового (граф-схемы Калужнина).

Инструментарий основывается на методе диалогового конструирования синтаксически правильных программ и использует различные формы представления алгоритмов в процессе их проектирования – естественно-лингвистическую (САА-схемы), алгебраическую (регулярные схемы) и граф-схемную [7]. 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) [7] символьной мультиобработки, содержащей описание конструкций САА-М, базисных понятий и их программных реализаций; метаправила конструирования алгоритмов, схемы алгоритмов, стратегии обработки и гиперсхемы.

Инструментарий, обозреваемый в данной статье, ориентирован на решение ряда проблем связанных с построением модульной структуры, основанной на типах объектов в объектно-ориентированной парадигме. Данный инструментарий основывается на методе проектирования абстрактного типа данных (АТД) в алгебре алгоритмики [1], а так же используется метод диалогового конструирования синтаксически правильных алгоритмов и программ.

Совмещая два этих метода, инструментарий представляет гибкое решение, содержащее описания высокого уровня, при этом не связанное с особенностями реализации. Основные преимущества:

- спецификации АТД как формальное математическое описание;
- модель АТД не несет императивных состояний;
- математическая модель для описания не всюду определенных операций;
- гибкий механизм перехода от анализа и спецификации к проектированию и реализации;
- разработка происходит в общем виде, удобном для повторного использования;
- описание является связью между АТД и классом на целевом языке программирования (ЯП).

В основу упомянутых инструментальных средств проектирования и генерации программ положен аппарат САА и их модификации [7 – 10]. Модифицированные САА (САА-М) предназначены для формализации процессов мультиобработки, возникающих, в частности при конструировании программного решения для мультипроцессорных систем.

САА являются двухосновной алгеброй $< U, B; \Omega >$, где U – множество операторов, B – множество логических условий [7]. 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; μ – неопределенность. Сигнатура операций САА $\Omega = \Omega_1 \cup \Omega_2$ состоит из системы Ω_1 логических операций, принимающих значения в множестве B и системы Ω_2 операций, принимающих значения в множестве операторов U. В САА $< U, B; \Omega >$ фиксируется система образующих Σ, представ-

ляющая собой конечную функционально полную совокупность операторов и логических условий. С помощью этой совокупности посредством суперпозиции операций, входящих в Ω, порождаются произвольные операторы и логические условия из множеств U и B. К логическим операциям системы Ω_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 Ω_2 относятся: композиция $A * B$, альтернатива (α-дизъюнкция) ([α] A, B), цикл (α-итерация) {[α] A}. Представления в САА < U, B; Ω > операторов из U посредством суперпозиций, входящих в сигнатуру Ω, получили название регулярных схем (РС) [7].

Сигнатура Ω' модифицированных САА, кроме перечисленных конструкций, входящих в Ω, 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 [7, 9, 10]. 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 [7]. 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 [7], 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.

Пример 1. Проиллюстрируем рассмотренные понятия на примере произвольной параллельной челночной сортировки. 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 челночную сортировку подмассива $M_0(i)$ и их вставку в массив M. Затем выполняется операция СЛИЯНИЕ, суть которой слияние упорядоченных подмассивов. Параллельная регулярная схема (ПРС) $ЧЕЛНОК\ (M_0, M, m)$ имеет следующий вид:

$$ЧЕЛНОК(M_0, M, m) ::= СТАРТ * ПСОРТ * СЛИЯНИЕ * ФИН$$

$$ПСОРТ ::= (\dot{\bigvee_{i=1}^{m}} \text{ЧЕЛ\_ВЕТВЬ}(i)) * S(ОБР\_ЗАК)$$

$$СЛИЯНИЕ ::= \{[l \leq r\,|!]\,(\dot{\bigvee_{i=1}^{m}} \text{СЛН\_ВЕТВЬ}(i)) * S(ОБР\_ЗАК), E\}$$

$$ЧЕЛ\_ВЕТВЬ(i) ::= \{[d(У_1, m_1, m_2)]\ R(У_2, У_1) * \{[l > r\,|\,У_2]\ ТРАНСП(l, r, У_2) * L(У_2)\}$$
$$* УСТ(У_2, У_1)\} * T(ОБР\_ЗАК(i))$$
$$СЛН\_ВЕТВЬ(i) ::= \{[l \leq r\,|\,У_1]\ ТРАНСП(l, r, У_2) * L(У_2)\} * УСТ(У_2, У_1) * T(ОБР\_ЗАК(i))$$

где,

- *СТАРТ* – оператор инициализации данных (ввод сортируемого массива);
- *ЧЕЛ_ВЕТВЬ*(i) – ветвь, осуществляющая сортировку элементов подмассива $M_0(i)$;
- *СЛН_ВЕТВЬ*(i) – ветвь осуществляющая слияние упорядоченного подмассива с остальными;
- 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).

Суть приведенного алгоритма состоит в параллельном функционировании m ветвей вычислений *ЧЕЛ_ВЕТВЬ*(i), каждая из которых осуществляет сортировку подмассива $M_0(i)$ в массив *M*. В операторах *ПСОРТ* и *СЛИЯНИЕ* 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 детальнее методы описаны в [5 – 8].

2. Разработка приложения в объектно-ориентированном инструментарии проектирования и синтеза программ

Как было описано раннее, инструментарий основывается на двух методах: метод проектирования абстрактного типа данных и метод диалогового конструирования синтаксически правильных программ.

2.1. Метод проектирования абстрактного типа данных в алгебре алгоритмики. Суть метода состоит в построении объектно-ориентированного решения как совокупность взаимодействующих, с разной степенью описания, абстрактных типов данных [1]. Сам процесс проходит три этапа:

1) анализ или спецификация – на данном этапе описываются расширенные абстрактные типы данных;
2) проектирование – на основе описанных АТД строятся отложенные и/или эффективные алгебраические классы (АКС). Под отложенными понимаются классы с частичным уровнем детализации, т.е. «классы-родители»;
3) реализация – происходит реализация всех построенных «эффективных» алгебраических классов к целевому языку программирования.

Рассмотрим кратко первые два этапа.

Этап 1. Состоит в построении расширенных абстрактных типов данных. Под расширенным АТД понимается система $< T, B, S, F, A, P >$, где:

- T – описываемый тип:

$$ADT[T];$$

- B – базис или совокупность основ (сортов):

$$\begin{vmatrix} \{q_1 \mid q_1 \in Q_1\} \\ \vdots \\ \{q_i \mid q_i \in Q_i\} \end{vmatrix}, Q_i (1 \le i \le n);$$

- S – объединение базисных предикатов и операций, определенных на совокупности основ:
-

$$\{Sign(o) \cup Sign(p)\};$$

- F – полные и/или частные функции для обработки основ. Под частной понимаются функция, если она определена не для всех элементов области определения, в этом случае она описывается с перечеркнутой стрелкой. Функции также делятся на три категории: конструкторы, запросы и команды;
- A – аксиомы, по сути, являются предикатами (в смысле логики), выражающими истинность некоторых свойств, для всех возможных значений из АТД;
- P – предусловия, булевские выражения, которые определяют область частных функций.

Этап 2. Состоит в построении алгебраических классов на основе построенных абстрактных типов данных. Под алгебраическим классом понимается система $< T, B, F, I >$, где:

- T – тип, характеризуемый абстрактным типом данных, либо (в случае множественной реализации) множеством абстрактных типов данных:

$$T : \{АТД_1, ..., АТД_n\},\ АТД_j (1 \le j \le n).$$

- B – множество, полученное в результате объединения базиса из $АТД_i$ и конкретизации (привязки к целевой платформе, или «реализации») базиса:

$$B = Basis^{АТД} \cup Basis;$$

- F – описание алгоритмов функций посредством *Сигнатуры* из АТД в строгих рамках аксиом и предусловий. Как пример, в статье в качестве описания выбрана Алгебра Дейкстры [7]:

$$f ::= S,$$

где S – структурная схема;

- I – множество, характеризующее интерфейс класса с точки зрения ООП. Содержит декларации функций, представляемые открыто классом. Отметим, что функции, описанные в АТД, но не содержащиеся в I, следует рассматривать как внутренние или «приватные».

$$I:\begin{Bmatrix} f_0 \in Функции \Rightarrow T \\ \dots \\ f_n \in Функции \Rightarrow T \end{Bmatrix}.$$

Для получения эффективного класса необходимо описать:

- спецификации АТД;
- выбор представления (View);
- отображение из АТД в View в виде множества компонентов (features), каждый из которых реализует одну из функций в рамках представления и при этом строго удовлетворяет аксиомам и предусловиям.

Следует отметить, что данные компоненты в процессе реализации могут сформироваться как в процедуры и/или функции, так и в качестве полей данных или атрибутов.

Отложенные классы служат для классификации групп связанных типов объектов, описывают важные многократно используемые модули высокого уровня, фиксируют общие свойства поведения. Именно они играют ключевую роль в полиморфизме, а также обеспечении децентрализации и расширяемости программной архитектуры.

Если спецификации АТД являются аппликативными, то в классах аппликативная точка зрения на функции отбрасывается, и команды переопределяются как операции, которые могут изменять объекты. Такое изменение четко отражает императивную парадигму, преобладающую при разработке ПО. Это в свою очередь влечет изменение в аксиомах АТД.

Пример 2. Проиллюстрируем создания класса, на примере части инструментария – «предусловия». Для начала опишем АТД *CoreClass*, который состоит из одинаковых данных для всех частей инструментария (для построения АТД).

АТД *CoreClass*:

1. Тип:

$$CoreClass[C].$$

2. Базис:

$$\begin{vmatrix} \{id \mid id \in Строка\} \\ \{adtId \mid adtId \in Строка\} \\ \{E \mid E \in Пусто\} \end{vmatrix}.$$

3. Сигнатура пустая.
4. Функции:

$$New :\to C;$$
$$getId :\to id;$$
$$setId\ id :\to E;$$
$$getAdtId :\to adtId;$$
$$setAdtId\ adtId :\to E.$$

5. Аксиомы:

@1 $getId(New) = id$;
@2 $getAdtId(New) = adtId$;
@3 $getId(setId(id)) = id$;
@4 $getAdtId(setAdtId(adtId)) = adtId$.

6. Предусловий нет.

Далее описывается АТД *Precondition* который расширяет (обогащает) вышеописанный АТД *CoreClass*.

АТД *Precondition* расширяет АТД *CoreClass*:

1. Тип:

$$Precondition[P].$$

2. Базис:

$$\begin{vmatrix} \{rightPartFunc \mid rightFunc \in Строка\} \\ \{condition \mid condition \in Строка\} \\ \{leftPartFunc \mid leftPartFunc \in Строка\} \\ \{void \mid void \in Пусто\} \end{vmatrix}.$$

3. Сигнатура также пустая, так как данный класс описывает структуру данных.
4. Функции:

$$New : \rightarrow P;$$
$$getRightPartFunc : \rightarrow rightPartFunc;$$
$$setRightPartFunc\ rightPartFunc : \rightarrow E;$$
$$getCondition : \rightarrow condition;$$
$$setCondition\ condition : \rightarrow E;$$
$$getLeftPartFunc : \rightarrow leftPartFunc;$$
$$setLeftPartFunc\ leftPartFunc : \rightarrow E.$$

5. Аксиомы:

@1 *getRightPartFunc*(*New*) = *rightPartFunc*;
@2 *getCondition*(*New*) = *condition*;
@3 *getLeftPartFunc*(*New*) = *leftPartFunc*;
@4 *getRightPartFunc*(*setRightPartFunc*(*rightPartFunc*)) = *rightPartFunc*;
@5 *getCondition*(*setCondition*(*condition*)) = *condition*;
@6 *getLeftPartFunc*(*setLeftPartFunc*(*leftPartFunc*)) = *leftPartFunc*;

6. Предусловий нет.

Следующий этап описание алгебраического эффективного класса.

Класс *Precondition*:

1. Тип:

АТД *Precondition*;

2. Базис:

$$Базис^{CoreClass} \cup Базис^{\Pr econdition} \cup \begin{vmatrix} \{Строка \mid Строка \in String\} \\ \{E \mid E \in void\} \end{vmatrix}.$$

3. Функции не нуждаются в детализации, так как класс содержит методы для установки и чтения полей (в ООП известны как "аксессоры" и "модификаторы").

4. Интерфейс:

$$Интерфейс = \left\{ \begin{array}{l} New, \\ getId(), \\ setId(String), \\ getAdtId(), \\ setAdtId(String), \\ getRightPartFunc(), \\ setRightPartFunc(String) \\ getCondition(), \\ setCondition(String), \\ getLeftPartFunc(), \\ setLeftPartFunc(String). \end{array} \right\}.$$

2.2. Метод диалогового конструирования схем алгоритмов. В основу построения функций алгебраических классов, в рассматриваемом инструментарии, положен диалоговый режим с использованием меню выбора функции, меню подстановок конструкций и дерева конструирования алгоритма функции. Меню выбора функции содержит декларации функций объявленные на этапе построения абстрактного типа данных. Меню выбора подстановок содержит САА-М конструкции, которые также объявлялись на этапе построения абстрактного типа данных в секции «сигнатура», суперпозиция которых позволяет создавать алгоритмы функций (методов класса). В зависимости от выбранной переменной инструментарий предлагает соответствующий список операций САА-М или базисных понятий.

Следует отметить поуровневый стиль конструирования алгоритма (ранее предложенный в ДСП-конструкторе [7, 9, 10]), а также возможность переходов на различные уровни (узлы дерева) с продолжением процесса диалогового конструирования.

Процесс построения дерева конструирования *T* для некоторого алгоритма показан на рисунке.

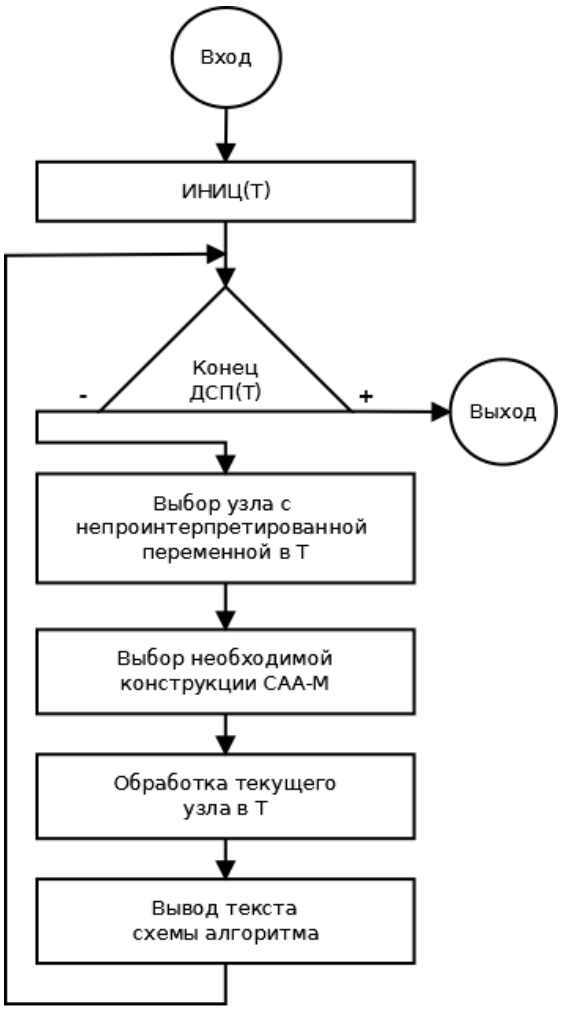

Рисунок. Процесс построения дерева *Т*

Где ИНИЦ(Т) формирование корневой вершины дерева *Т* с названием первого составного оператора, а также его дочернего узла, помеченного именем операторной переменной; КОНЕЦ ДСП(*Т*) – условие, истинное, если все переменные конструируемой схемы проинтерпретированы. В процессе ДСП конструирования, выбор узла и выбор необходимой конструкции (базисного или составного элемента) выполняются пользователем.

2.3. Синтез. Суть его сводится к сборке класса на целевом языке программирования (Java). Формируется каркас класса, затем его поля (данные) и его методы (для обработки данных). Все базисные абстракции из АТД детализируются исходя из базиса АКС. Затем к ним формируются методы для чтения и установки. Следует отметить, что данная опция является настраиваемой и при надобности отключается.

По полученным в процессе конструирования алгоритма деревьям функций, реализациям базисных операторов и предикатов, а также другим фрагментам ДСП-конструктор выполняет синтез методов класса. 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 описаний в сигнатуре АТД инструментария. Отметим тот факт, что в качестве «базисных» операторов и предикатов с точки зрения САА-схемы, на уровне детализации могут оказаться как составные операторы и предикаты, так и подпрограммы (методы).

В таблице приведены описания основных операторных конструкций САА (композиция, альтернатива и цикл), реализации, которых на языке Java используются в процессе синтеза кода.

Приведенные реализации содержат маркеры вида $a, $b и $u, вместо которых будут подставлены указанные в процессе проектирования реализации этих операций.

Таблица. Описание в СКАО операторных операций САА

Аналитическая форма	Естественно-лингвистическая форма	Реализация на языке Java
`A` * `B`	`A` ЗАТЕМ `B`	$a; $b
([`U`] `A`, `B`)	ЕСЛИ `U` ТО `A` ИНАЧЕ `B`	if ($u) { $a } else { $b }
{ [`U`] `A` }	ПОКА НЕ `U` ЦИКЛ `A`	while (!$u) { $a }
{ `A` [`U`] }	ЦИКЛ `A` ПОКА НЕ `U`	do {$a} while (!$u)

Генерация кода функций в ДСП конструкторе осуществляется по дереву конструирования, все терминальные вершины которого являются базисными операторами или предикатами (т.е. все переменные сконструированной 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. Обработка каждой вершины дерева конструирования осуществляется в зависимости от ее типа, а именно: базисное понятие или операция САА. Затем осуществляется подстановка в реализацию значений фактических параметров базисного понятия 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. Таким образом, формируются все детализированные функции из АКС. Если обнаружены функции, описанные в АТД, но не детализированные в АКС, то они формируются как абстрактные декларации.

Асинхронные алгоритмы функций, представленные в САА-М, могут быть реализованы с помощью потоков языка Java. Поток в многопоточных операционных системах – наименьшая единица выполнения. Для каж-

дого 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 [2]. Следует отметить, что языком реализации, Java была выбрана из-за поддержки многопоточности не только на уровне библиотек, но и на уровне самого языка, который значительно облегчает построение программ, которые в свою очередь 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» [2]. 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.

Пример 3. В качестве иллюстрации рассмотрим класс ThreadWorker реализации многопоточной программы, сгенерированной по ПРС ЧЕЛНОК (см. пример 1).

Первый этап абстрактный тип данных:

АДТ *ThreadWorker* (описания для установки и чтения полей упущены):

1. Тип:

$$\text{ThreadWorker[T]}.$$

2. Базис:

$$\left|\begin{array}{l}\{array \mid array \in ЧислМассив\}\\ \{pId \mid pId \in Число\}\\ \{workTime \mid workTime \in ВремяРаботы\}\\ \{isInvolved \mid isInvolved \in E2 = \{Истинно, Ложно\}\}\\ \{thread \mid thread \in Поток\}\\ \{E \mid E \in Пусто\}\end{array}\right|$$

3. Сигнатура:

$$Сиг н(o) = \left\{\begin{array}{l} R(У_2, У_1),\\ ТРАНСП(l, r, У_2),\\ УСТ(У_2, У_1),\\ L(У_2),\\ запуститьПоток(),\\ времяМиллСек() \end{array}\right\},$$

$$Сиг н(p) = \begin{Bmatrix} d(У_1, m_1, m_2), \\ l > r \mid У_2, \\ живПоток() \end{Bmatrix}.$$

4. Функции:

$$New : \to T,$$
$$start : \to E,$$
$$run : \to E,$$
$$\ldots$$
$$sort: \not\to E,$$
$$isInvolved : \not\to E2,$$
$$isAlive : \not\to E2.$$

5. Аксиомы:

$$\ldots$$
$$@\ sort(\ setArray()\ ) = E,$$
$$@\ isInvolved(\ New\ ) = E2,$$
$$@\ isInvolved(\ setInvolved(\ E2\ )\ ) = E2,$$
$$@\ isAlive(\ New\ ) = E2,$$
$$@\ isAlive(\ run\ ) = E2.$$

6. Предусловия:

$$\ldots$$
p: *isInvolved*() если *setThread*(),
p: *isInvolved*() если *run*(),
p: *isAlive*() если *setThread*(),
p: *isAlive*() если *run*(),
p: *sort*() если *setArray*().

Второй этап алгебраический класс:

Класс *ThreadWorker* (описания для установки и чтения полей упущены)

1. Тип:

АДТ *ThreadWorker*.

2. Базис:

$$\begin{vmatrix} \{ЧислМассив \mid ЧислМассив \in \text{int}[]\} \\ \{Число \mid Число \in \text{int}\} \\ \{ВремяРаботы \mid ВремяРаботы \in \text{int}\} \\ \{E2 \mid E2 \in Boolean\} \\ \{Поток \mid Поток \in Thread\} \\ \{Пусто \mid Пусто \in void\} \end{vmatrix}$$

3. Функции:

$$start ::= запуститьПоток();$$
$$sort ::= \{[d(У_1, m_1, m_2)]\ R(У_2, У_1) * \{[l > r \mid У_2] ТРАНСП(l, r, У_2) * L(У_2)\} * УСТ(У_2, У_1)\};$$
$$run ::= времяМиллСек(ВремяРаботы) * sort() * времяМиллСек(ВремяРаботы).$$

4. Интерфейс:

$$Интерфейс = \begin{Bmatrix} New, \\ start, \\ run, \\ \ldots \\ sort, \\ isInvolved, \\ isAlive \end{Bmatrix}$$

Заключение

В статье рассмотрены средства формализованного проектирования и синтеза абстрактных типов данных, алгебраических классов и объектно-ориентированных программ, по их представлениям в алгебрах алгоритмов.

Описан метод синтеза, как части самого инструментария, так и многопоточной программы на языке Java, формализованной в САА-М.

К ограничениям инструментария можно отнести то, что исходный код генерируется, пока что, на языке программирования Java. Это ограничение будет убрано в дальнейшей работе над инструментарием.

Так же к перспективам инструментария следует отнести обогащение его на синтез параллельных программ, использующих технологию MPI.

1. *Дорошенко А.Е., Иовчев О.В.* О методе проектирования абстрактного типа данных в алгебре алгоритмики // Проблеми програмування. – 2012. – № 1. – С. 3 – 16.
2. *Bruce Eckel.* Thinking in Java, 4th edition, 2006.
3. *Bertrand Meyer.* Object-Oriented Software Construction, Second Edition, Prentice Hall. – 1997.
4. *Акуловский В.Г.* Некоторые аспекты формализации архитектурного этапа разработки алгоритмов // Проблеми програмування. – 2009. – № 2. – С. 3 – 11.
5. *Ющенко Е.Л., Цейтлин Г.Е., Грицай В.П., Терзян Т.К.* Многоуровневое структурное проектирование программ: Теоретические основы, инструментарий. – М.: Финансы и статистика, 1989. – 208 с.
6. *Глушков В.М., Цейтлин Г.Е., Ющенко Е.Л.* Алгебра. Языки. Программирование. 3-е изд., перераб. и доп. – Киев: Наук. думка, 1989. – 376 с.
7. *Андон Ф.И., Дорошенко А.Е., Цейтлин Г.Е., Яценко Е.А.* Алгеброалгоритмические модели и методы параллельного программирования. – Киев: Академпериодика, 2007. – 634 с.
8. *Глушков В.М., Цейтлин Г.Е., Ющенко Е.Л.* Методы символьной мультиобработки. – Киев: Наук. думка, 1980. – 252 с.
9. *Иовчев В.А., Мохница А.С.* Инструментальные средства алгебры алгоритмики на платформе WEB 2.0 // Проблеми програмування. (матеріали конф. УкрПрог-2010). – 2010. – № 2 – 3. – С. 547 – 556.
10. *Иовчев В.А., Мохница А.С.* Формальный метод генерации программ в инструментальных средствах алгебры алгоритмики // матеріали конф. TAAPSD’2010.