© 2012 г. Д.И. ХАРИТОНОВ, к-т техн. наук
(Институт автоматики и процессов управления ДВО РАН, Владивосток)

РАСШИРЕНИЕ ЯЗЫКА C++ ДЛЯ СПЕЦИФИКАЦИИ ОБЪЕКТОВ, ИНТЕРФЕЙСОВ КЛАССОВ, А ТАКЖЕ ПРИНЦИПОВ ГЕОМЕТРИЧЕСКОЙ ДЕКОМПОЗИЦИИ[1]

В статье представлено расширение синтаксиса языка C++, описывающее формально способ аннотирования объектно-ориентированных программ с целью предоставления дополнительной информации интеллектуальным средствам анализа и распараллеливания программ. Особенностью представленного способа является использование дополнительных файлов спецификаций для описания протоколов классов и принципов декомпозиции программ, связываемых с текстом программы короткими аннотациями. Аннотированная программа может контролироваться на соответствие протоколам и принципам декомпозиции в течение всего своего жизненного цикла.

C++ PROGRAMMING LANGUAGE EXTENSION FOR SPECIFICATION OF OBJECTS, INTERFACES OF CLASSES AND GEOMETRICAL DECOMPOSITION PRINCIPLES / D.I. Kharitonov (Institute for automation and control processes FEBRAS, Radio street 5, Vladivostok 690041, Russia, E-mail: demiurg@dvo.ru). An abstract in English. This article represents an extension of C++ programming language syntax defining formal approach to object-oriented program annotation with the aim of giving extra information to intellectual tools of program analysis and paralleling. Peculiarity of approach presented is use of external specifications files for class's protocols and decomposition principles bound with the text of program by short annotations. Program annotated this way can be controlled for compliance with protocols and decomposition principles during all its life cycle.

1. Введение

Развитие современных программных средств во многом следует тенденциям в развитии аппаратного обеспечения. В последние десять лет на рынке микроэлектроники действуют два основных вектора развития - многопроцессорность и миниатюризация, связанная с развитием переносных устройств. Соответственно и в программном обеспечении усилия направлены на два практически противоположных направления: развитие многопроцессорных и многопоточных приложений, а также на переносимость программного обеспечения. Однако у этих направлений можно выделить общее сходство –

[1] Работа выполнена при финансовой поддержке программ Президиума РАН №14 и №15 (гранты № 2012-2014.ДВО.12-I-П18-03 и 2012-2014.ДВО.12-I-П15-04).

для построения качественных параллельных программ, как и для построения переносимых программ, требуется лучшее понимание и анализ работы программ, чем в традиционном последовательном программировании.

Необходимо отметить, что переносимость наряду с эффективностью и корректностью исполняемого кода являются основными проблемами параллельного программирования. Проблема переносимости состоит в том, что программы, предназначенные для выполнения на одной архитектуре МВС, либо не могут быть выполнены на другой архитектуре, либо их выполнение будет неэффективно. Проблема эффективности состоит в том, что начиная с определённого числа процессоров или ядер процессоров, производительность параллельной программы начинает падать, вследствие роста издержек, связанных с взаимодействием процессов. Проблема корректности исполняемого кода параллельной программы (в узком смысле этого слова) непосредственно связана с взаимодействием процессов внутри параллельной программы, вследствие которого возможны ошибки называемые race conditions и deadlocks.

Учитывая, что с момента появления первого современного языка программирования Fortran прошло уже 55 лет трудно ожидать, что в обозримом будущем появятся принципиально новые языки программирования, отличающиеся и от современных императивных, и от объектно-ориентированных, и от функциональных. Поэтому следует ожидать повышения уровня интеллектуальности в средствах программирования, компиляторах и средах исполнения программ. Но, руководствуясь только исходным текстом программы, очень сложно получить нечто отличное от самой программы. Например, автоматическое распараллеливание последовательных алгоритмов представляется логичным подходом к решению проблем параллельного программирования. Однако эффективность распараллеливающих компиляторов недостаточно велика, что связано с необходимостью сложного анализа последовательного алгоритма для «понимания» принципов его работы, поэтому даже небольшие подсказки распараллеливающему компилятору со стороны программиста могут существенно повысить его эффективность.

Значительное количество параллельных программ связано с вычислениями на решётках, в связи с этим заслуживает внимания метод геометрической декомпозиции данных для распараллеливания последовательного алгоритма. Программист, реализующий распараллеливание алгоритма с использованием этого метода должен определить зависимость вычислений значений узлов решётки от значений в других узлах, основные фазы вычислений и пределы распараллеливания программы. Представляется перспективным расширить синтаксис языка C++ директивами передающими компилятору эту необходимую информацию для автоматизации распараллеливания программы методом геометрической декомпозиции. Основной идеей такого расширения является не только облегчение создания распараллеливающего компилятора, но и необходимая для корректности распараллеливания проверка программы на соответствие спецификации. Такая проверка будет обеспечивать самоконтроль программиста при длительной разработке программ.

Изложение в настоящей статье представлено следующим образом. Во втором разделе описывается синтаксис расширения языка C++ и приводится простой пример аннотированной программы со спецификацией протокола класса. В третьем разделе описывается семантика расширения языка C++ и приводится пример аннотированной программы со спецификацией принципов декомпозиции программы. В четвертом разделе описывается прагматика предложенного расширения языка C++. В пятом разделе приводится сравнение предложенного расширения с другими методами расширения языков программирования, предназначенными для построения параллельных программ. Статья заканчивается заключением, в котором суммируются полученные результаты.

2. Синтаксис

Расширение синтаксиса языка C++ необходимо для внедрения в исходный текст программы информации, используемой внешними по отношению к компилятору C++ средствами для анализа, верификации и трансформации исходного текста программы. Текст программы на языке C++ обычно хранится в файлах с расширениями “c++” или “cpp”, “h”, “hpp”. Текст программы с расширенным синтаксисом хранится в файлах с расширениями “xc++”, “xcpp”, “xh”, “xhpp”, такие файлы называются аннотированными.

Стандарт C++ предусматривает девять последовательных фаз трансляции исходного текста программы, обусловленных приоритетами синтаксических правил:

1) Обработка символов исходного файла, замена специальных последовательностей символов, преобразование символов из исходного множества к внутренней кодировке транслятора.
2) Формирование из строк исходного файла логических строк (удаление символов продолжения строки перед символом переноса строки).
3) Декомпозиция файла на токены препроцессора и последовательности пробельных символов (включая комментарии).
4) Выполнение директив препроцессора, замена макроподстановок. Выполнение директивы #include влечёт за собой рекурсивное выполнение фаз с 1 по 4.
5) Преобразование символьных строк к кодировке исполняемого файла.
6) Конкатенация соседних символьных строк.
7) Удаление пробельных символов. Преобразование токенов препроцессора в токены синтаксического разбора. Выполнение синтаксического разбора единицы трансляции программы (файла программы).
8) Для каждой обработанной единицы трансляции формируется список экземпляров реализации. Все необходимые экземпляры синтезируются, порождая единицу реализации (объектный файл).
9) Разрешаются внешние связи. Все единицы реализации, включая библиотечные компоненты, объединяются в образ программы.

Фазы трансляции с 1 по 6 подготавливают исходный файл к основной трансляции. Наиболее подходящей фазой внедрения расширения является фаза №3 – то есть до выполнения директив препроцессора. Естественным аналогом аннотаций являются комментарии, но так как текст комментариев может быть любым, аннотации оформляются подобным, но не идентичным способом.

Расширение синтаксиса языка производится добавлением синтаксической конструкции Annotation (в стандарте языка C++ нет правил использующих символ “@”):

(1)
Annotation → “@” “[“ AnnText “]”
AnnText → AnnCommand AnnParams
AnnParams →
AnnParams → AnnParam AnnParams

Остальной синтаксис аннотаций определяется в дополнительных файлах спецификаций, который подключаются в тексте программы при помощи аннотации:

(2) @[include FILENAME]

аналогично заголовочным файлам C++. Аннотированная программа содержит как минимум один файл спецификаций - манифест программы. На рисунке 1 приведён пример файла декларации класса IndexList аннотированного для описания интерфейса взаимодействия. Для удобства чтения аннотации выделены курсивом.

```
#pragma once
@[include Specification.xar]
#include "BaseClasses.h"
class IndexList{ //Декларация класса IndexList
public:
    @[constructor] IndexList(); //инициализ. внутр. переменные
    @[destructor]~IndexList(); // освобождение выделенной памяти
private:
    . . .           //внутренние функции класса
public:
    bool @[in_state1] Add(const char *index,void *pElement); // добавление
                                                        // элемента в список
    bool @[in_state1] Del(int pos); // удаление элемента из списка
    bool @[to_state1] MakeIndex(); // построение индекса из списка
    bool @[to_state2] MakeList(); // переход к редактированию списка
    int @[in_state2] Find(const char *index); // поиск позиции по индексу
    void* @[in_state2] Get(int pos); // получение элемента по позиции
    const char *@[in_state2] Index(int pos); // индекс по позиции
    int @[free_call] Count(); // количество элементов списка
};
```

Рис. 1. Пример аннотированного исходного файла.

Грамматика файлов спецификаций в форме БНФ выглядит следующим образом:

(3)

*File → List

List → Name “{“ InsideList “}” *newline*

List → Name “(“ Value “)” “{“ InsideList “}” *newline*

InsideList →

InsideList → InsideList Item

InsideList → InsideList List

Item → Name *newline*

Item → Name “:” Value *newline*

Таким образом текст спецификаций записывается в XML-подобном списке из простых и сложных элементов. Каждый элемент имеет собственное имя и может содержать

текстовое значение. Сложные элементы также могут содержать список из простых и сложных элементов.

```
ProtocolManifest {
      ProjectName: DictionaryCombine
      MainFile: DictionaryCombine.cpp
      PetriNetsDir: Protocols
Prototypes{
      IndexList:Index.pn3
      . . .
}
}
```

Index.pn3

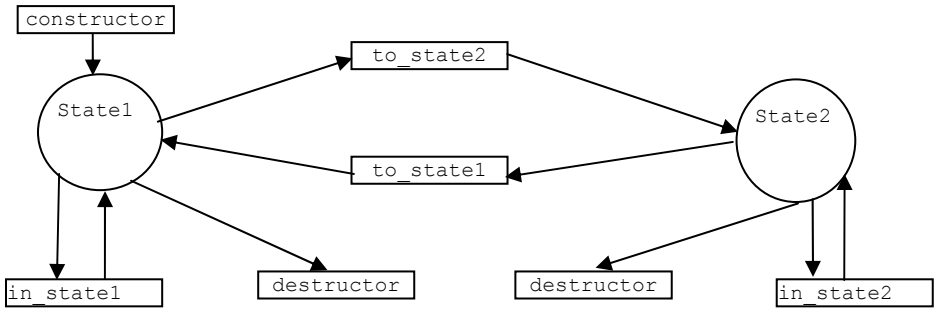

Рис. 2. Пример файла спецификации интерфейса взаимодействия объектов и протокола класса в терминах сетей Петри.

На рисунке 2 сверху представлен пример текста спецификации интерфейсов взаимодействия единиц анализа программы. Этот файл состоит из двух вложенных списков. Первый список - основной, называется ProtocolManifest он содержит второй список Prototypes и общие настройки, предназначенные для передачи в средства анализа программы. Настройки состоят из наименования анализируемого проекта, указания на главный файл проекта и указания на директорию с файлами протоколов единиц анализа. Второй список содержит перечисление задействованных в программе единиц анализа и указания на файлы с описанием протоколов этих единиц. На рисунке 2 снизу представлен пример протокола класса в терминах сетей Петри. Переходы помечены символами, используемыми при аннотировании программы.

3. Семантика

Целью аннотирования программы является предоставление дополнительной информации интеллектуальным средствам анализа и распараллеливания программ, для этого программа логически разделяется на части, называемые единицами анализа. Каждой единице анализа сопоставляется часть исходного текста программы, называемая реализацией. С целью анализа программы должна быть построена модель единицы анализа и модель окружения - оставшейся части программы. На модели проверяются свойства программы. Свойства, относящиеся к поведению единицы анализа, могут быть записаны в форме протокола единицы анализа в терминах сетей Петри или их расширений – цветных и композициональных сетей Петри[2]. Далее будем считать, что все модели и проверки выполняются с использованием сетей Петри. Имея в наличии протокол только одной единицы анализа, верификация требований к программе[5] будет состоять из двух шагов базового алгоритма:

1) Анализа соответствия поведения единицы анализа своему протоколу. В ходе этого анализа проверяется, что пространство состояний протокола единицы анализа имеет отображение в пространство состояний модели единицы анализа.
2) Анализа корректности использования единицы анализа окружением. В ходе этого анализа проверяется, что композиция модели окружения единицы анализа и протокола единицы анализа не имеет дедлоков.

Построение модели окружения единицы производится способом аналогичным построению модели программы из функции main[3]. Для этого методом синтаксически управляемой трансляции императивные конструкции языка C++ отображаются в шаблонные конструкции в терминах композициональных сетей Петри. Вызов функций и методов в данном отображении имеет точки доступа для слияния с моделью функции. Таким образом, имея модели всех функций и методов, доступных из функции main, получается композициональная модель всей программы. Модель окружения единицы анализа отличается от модели всей программы отсутствием моделей функций и методов, относящихся к единице окружения. Дополнительного разделения процесса верификации на части можно добиться, если множество функций и методов программы разделить на несколько непересекающихся множеств, представляющих разные алгоритмические пространства. При этом в протоколах единиц анализа функции и методы одного алгоритмического пространства должны быть обособлены, то есть не должны изменять состояния другого алгоритмического пространства.

При наличии протоколов нескольких единиц анализа можно применить базовый алгоритм анализа в отношении каждой единицы анализа. Если единицы анализа не взаимодействуют друг с другом, то анализ корректности использования единиц можно производить одновременно в отношении всех единиц. Если же единицы анализа взаимодействуют друг с другом, то возможно разделение процедуры анализа путём объединения нескольких единиц анализа в одну более сложную – комплексную единицу, со своим протоколом. Тогда анализ этой единицы будет состоять из следующих шагов:

1) Анализ соответствия внутренних единиц своим протоколам, как в 1ом шаге базового алгоритма.
2) Анализ корректности использования внутренних единиц комплексной единицей. В ходе этого анализа проверяется отсутствие дедлоков в композиции протоколов внутренних единиц и протокола комплексной единицы.
3) Анализ корректности использования комплексной единицы окружением, как во 2ом шаге базового алгоритма. В ходе этого анализа проверяется отсутствие дедлоков в композиции модели окружения комплексной единицы и протокола комплексной единицы.

Таким образом, для проверки корректности использования в программе множества простых и комплексных единиц анализа необходимо разбить всё множество единиц анализа на группы попарно не взаимодействующих единиц анализа и построить модель окружения для каждой группы.

Описание принципов декомпозиции программы производится на основании следующих допущений:

1) Вычисления производятся на регулярной решётке в N мерном пространстве.
2) В каждом узле вычислительного пространства производятся вычисления по собственным формулам.
3) Распараллеливание потока управление затруднено тем, что алгоритм вычислений «размазан» по тексту программы (не собран в одной функции).

4) Вычислительная программа может изменяться со временем, то есть она не является законченной.

Конечной целью описания принципов декомпозиции является трансформация программы на множество взаимодействующих процессов для эффективного исполнения на кластерных системах с использованием библиотеки MPI. Спецификация принципов декомпозиции содержит:

1) Количество измерений, каждое из которых имеет своё название
2) Указание на границы вычислительного пространства
3) Способ распределения вычислительного пространства между узлами.
4) Описание принадлежности переменных в программе заданным измерениям.
5) Описание ограничений на вычислительные формулы, для формирования зон обмена граничными значениями

Чтобы метод декомпозиции был корректным, ограничения на формулы должны формально проверяться в процессе компиляции программы. Для этого предлагается использовать алгоритм проверки, подобный верификации протокола класса, состоящий из следующих шагов:

1) Построить протоколы всех переменных программы в терминах сетей Петри, содержащие события чтения и присвоения переменным. Для объектов, подлежащих геометрической декомпозиции, на операции присвоения накладываются ограничения из спецификации принципов декомпозиции.
2) Построить модель потока управления программы (модель окружения переменных) в терминах композициональных сетей Петри.
3) Проверить для каждого присвоения индексные ограничения. Для этого проверяется отсутствие дедлоков в композиции модели потока управления программы и моделей всех переменных комплексной программы.

При синтаксическом разборе текста программы на языке C++, аннотации могут располагаться в любом месте программы, где допустимы комментарии, поэтому файл C++ снабжённый дополнительными конструкциями может быть скомпилирован стандартными компиляторами C++ только после удаления аннотаций. Инструментом работы с аннотациями является аннотированное дерево абстрактного синтаксиса программы. При построении дерева абстрактного синтаксиса программы каждая аннотация из исходного текста программы размещается в наименьших объемлющих узлах дерева, разделителем в исходном тексте которого была встречена аннотация.

Название главного списка в файле спецификаций определяет область применения спецификаций. В приведённом выше примере название списка – ProtocolManifest, следовательно область применения спецификаций – описание протоколов интерфейсов объектов программы. Универсальный подход к описанию аннотаций позволяет определить обработчик для данного файла - название обработчика формируется заменой постфикса Manifest на постфикс Clerk. Таким образом, при трансляции файла будет использован обработчик ProtocolClerk. Простые элементы внутри главного списка определяют параметры, используемые обработчиком. Названия сложных элементов внутри главного списка являются зарезервированными словами и определяют тип содержимого списка. В зависимости от типа содержимого списка формируется множество элементов спецификации программы. Имена элементов в спецификациях программы должны быть уникальны и отличаться от зарезервированных слов. Элементам спецификации могут назначаться параметры числовых и перечислимых типов. Имена элементов спецификации

используются для аннотирования программы, как команды. Семантика аннотаций зависит от конкретной области применения.

4. Прагматика

Аннотированные файлы программы не могут быть откомпилированы традиционным компилятором, поэтому процесс анализа исходных текстов программы является неотъемлемой частью жизненного цикла программы. В зависимости от решаемых аннотированием программы задач в спецификациях программы содержится различная информация. В настоящий момент определено две области применения аннотирования. Во-первых – это автоматизация распараллеливания объектно-ориентированных программ. Во-вторых – это верификация взаимодействия объектов объектно-ориентированных программ.

Для автоматизации распараллеливания файл манифеста содержит описание принципов геометрической декомпозиции, а аннотация теста программы на языке C++ - помечает переменные и функции для определения объектов декомпозиции. Определено следующее множество зарезервированных слов:

(4) Keywords = { Algo, Decomposition, Stages, Dimensions, Repeated, Parallel, Exclusive, Restrictions, Deps, Src, Dst, Min, Max, ProjectName, MainFile, OutputDir, private, protected, public}

Для верификации взаимодействия объектно-ориентированных программ файлы спецификаций содержат описание протоколов классов и взаимосвязей между протоколами.

Определено следующее множество зарезервированных слов:

(5) Keywords = { Algo, States, Transition, Protocol, Prototypes, Open, Closed, For, Forbidden, Stages, private, protected, public}

К примеру зарезервированное слово Stages позволяет в обоих спецификациях определить множество стадий алгоритма. Подразумевается, что стадии алгоритма выполняются последовательно, то есть из функций одной стадии, не могут быть вызваны функции другой стадии. На разных стадиях разрешается пользоваться функциональностью различных алгоритмических пространств. Algo – алгоритмическое пространство (функции принадлежащие алгоритмическому пространству подчиняются единому протоколу). Между алгоритмическими пространствами определяются отношения, позволяющие вызывать функции и обращаться к переменным других алгоритмических пространств.

Задача построения спецификаций и аннотирования исходного текста – это предоставление информации для построения модели поведения программы. Далее модель программы может быть использована для проверки корректности реализации программы и/или декомпозиции программы на процессы для распределённого выполнения программы.

В приложении 1 представлен тест декларации класса, аннотированного для геометрической декомпозиции, а в приложении 2 – файл спецификаций принципов геометрической декомпозиции, использованный в тексте декларации класса. В файле спецификации принципов декомпозиции содержится пять разделов: Dimensions, Decomposition, Stages, Algo, Restrictions. В разделе Dimensions описана размерность вычислительного пространства. Заданы размерности X,Y,Z, причём по размерностям X и Y будет осуществляться геометрическая декомпозиция, а размерность Z дробиться не будет. В разделе

Stages описаны три стадии выполнения программы: Initialization, IterativeCalc, Finalization. Причём стадия Initialization должна выполняться на каждом вычислительном узле, стадия IterativeCalc выполняется в соответствии с геометрической декомпозицией параллельно, а стадия Finalization выполняется только на одном из узлов. В разделе Algo описано одно алгоритмическое пространство - InOut, доступное на стадиях Initialization и Finalization. Раздел Decomposition содержит параметры геометрической декомпозиции для стадии IterativeCalc. Выбран статический блочный алгоритм декомпозиции (SquareBlock), для размерностей X и Y, с весовой функцией пропорциональной площади блока. Раздел Restrictions содержит ограничения для формул в узлах вычислительной решётки. При присвоении значений переменным узла с координатами X, Y, Z определены ограничения на индексы узлов переменных в правой части формул: по координате X - плюс минус два узла, по координате Y - плюс минус один узел, а по координате Z - точное значение индексов в правой части должно быть равно 1-Z.

Аннотации исходного текста декларации класса в приложении 1 содержат: указания на файл со спецификациями принципов геометрической декомпозиции; пометку множества методов класса, относящихся к разделу Initialization и к алгоритмическому пространству InOut; пометку функций, возвращающих индексные значения по размерностям X и Y; пометку переменных, относящихся к размерностям X и Y; пометку переменных имеющих условно постоянные значения (Constant) и значения, применяющиеся для управления потоком команд (ControlFlow). Теоретически большая часть аннотаций может быть получена из анализа программы. Однако, во-первых, в процессе изменения программы автоматический расчет этих аннотаций может оказаться нестабильным, что влечёт резкое изменение эффективности распараллеливания. А во-вторых, определение этих аннотаций программистом оставляет управление процессом декомпозиции в руках человека, что в процессе проверки ограничений даст положительную обратную связь в течение жизненного цикла программы.

5. Сравнение с другими методами расширения синтаксиса языков программирования.

Проблемы параллельного программирования имеют системный характер и превращают задачу построения корректной эффективной программы в искусство, при этом алгоритмы параллельных программ становятся сложными, наукоёмкими и трудномодифицируемыми. В книге[1] отмечается, что многие пользователи параллельных вычислительных систем согласились бы использовать традиционные последовательные языки программирования для построения параллельных программ, если бы работу по распараллеливанию взял бы на себя компилятор. Однако в таком случае компилятору требуются подсказки, содержащие указания на те или иные свойства программ. Для передачи подсказок компилятору зачастую расширяется синтаксис языков программирования, вводятся директивы, записанные в комментариях, новые конструкции, дополнительные служебные функции и т.д.

Методы расширения синтаксиса языков программирования можно условно разделить на две группы. К первой группе относятся методы, использующие существующие синтаксические конструкции языка программирования, изменяя семантическое значение, обычно очень небольшой, группы конструкций. При этом программа, записанная в терминах расширенного языка программирования может быть скомпилирована как стандартным компилятором, так и компилятором расширенного языка. Примером подобного расширения является расширение синтаксиса языков C, C++ и Fortran стандартом OpenMP, а также расширения C-DVM и FORTRAN-DVM. В OpenMP применитель-

но к языкам C и C++ используется директива препроцессора #pragma, использующая ключевое слово omp для определения команд OpenMP. В C-DVM директивы, описывающие параллельное исполнение программы, описаны в виде пустых макросов, так что компиляция программы обычным компилятором генерирует последовательный код.

Ко второй группе методов расширения синтаксиса относятся расширения, выводящие текст программы на расширенном языке из области, воспринимаемых стандартным компилятором текстов. Типичным примером такого расширения является CUDA - разработанная компанией NVIDIA программно-аппаратная архитектура, позволяющая производить вычисления с использованием графических процессоров. В настоящее время существует два компилятора CUDA. Один, реализованный компанией NVIDIA, для расширенного языка C++, другой компании Portland Group, для расширения языка Fortran. Архитектура CUDA предполагает добавление следующих синтаксических элементов в исходные языки программирования:

1) Определение *атрибутов функций*, показывающих, где будет выполняться функция и откуда она может быть вызвана;
2) Определение *атрибутов переменных*, задающих тип используемой памяти;
3) *Оператор запуска ядра*, определяющий иерархию нитей, очередь команд и размер разделяемой памяти;
4) *Встроенные переменные* и дополнительные *типы данных*.

Основным элементом, определяющим параллелизм выполнения программы, является оператор запуска ядра, который не имеет ни синтаксического, ни семантического аналога в последовательных языках программирования.

Существуют также расширения языков программирования, которые трудно отнести лишь к одной из перечисленных групп. Например к таким расширениям можно отнести QT - кроссплатформенный инструментарий разработки приложений на языке C++, OpenCL - фреймворк для написания компьютерных программ, связанных с параллельными вычислениями на различных видах процессоров, включая графические, а также uC++ - расширение языка C++ такими конструкциями как сопрограмма, монитор, задача и т.д . С точки зрения синтаксиса QT, OpenCL и uC++ остаются в пределах синтаксиса C++, дополняемого набором макросов. Однако для построения исполняемой программы исходный текст программы должен быть скомпилирован специальными средствами, обеспечивающими целостность конечного кода.

Предложенное в настоящей статье расширение синтаксиса языка C++ не вводит новых императивных конструкций, как например CUDA. Программы, составленные с использованием расширенного синтаксиса не могут быть скомпилированы стандартным компилятором, как в случае OpenMP. С точки зрения синтаксиса языка C++ расширение не может быть представлено набором макросов. Предложенное расширение обязывает использовать специальный компилятор, тем самым интегрируя средства анализа программы в её жизненный цикл. С точки зрения семантики распараллеливания программы, исходный текст программы рассматривается как алгоритм вычислений, над которым необходимо выполнить преобразование геометрической декомпозиции, заданное в дополнительном файле спецификации. Для корректности преобразования необходимо, чтобы исходная программа удовлетворяла ряду ограничений, накладываемых на операции с данными на регулярной сетке. Для проверки этих ограничений строится модель программы в терминах композициональных сетей Петри, в которой описывается как модель потока управления, так и протоколы объектов данных.

6. Заключение.

В статье предложено расширение синтаксиса языка C++, предназначенное для аннотирования исходного текста программ. Аннотирование не изменяет императивных конструкций языка программирования, но позволяет описать внешние требования к программе, которые могут быть использованы сторонними средствами для проверки корректности взаимодействия классов в программе и геометрической декомпозиции программы. Основной массив требований к программе записывается во внешних файлах спецификаций. Короткие аннотации в тексте программы образуют связи между реализацией программы и внешними требованиями. Благодаря формализации синтаксиса файлов спецификаций и аннотаций эти связи определяется единственным образом. При помощи похожего подхода были реализованы high-performance расширения языков программировании C и Fortran (Fortran-DVM, C-DVM, HPF, OpenCL). Новизна предложенного в настоящем докладе расширения C++ заключается в универсальности подхода, позволяющего строить различные интеллектуальные средства анализа и трансформации программ, опираясь на единый синтаксис расширения. Кроме того, аннотированная программа может контролироваться на соответствие протоколам классов и принципам декомпозиции в течение всего своего жизненного цикла.

ПРИЛОЖЕНИЕ 1

Тест декларации класса, аннотированного для геометрической декомпозиции.

```
#ifndef ADOMAIN_H_INCLUDED
#define ADOMAIN_H_INCLUDED

@[include Specification.xar]
#include "BaseClasses.h"

class ACompData;
class AZone;
class ABoundary;

class ADomain
{
public:
     ADomain();
     virtual ~ADomain();

public @[Initialization]:
     static ADomain* MakeNew(const char* ClassName, ADomain*
BaseClass = NULL);
     virtual const char* GetClassName() = 0;

public @[Initialization]:
     virtual void InitZone(AZone& Zone);
     virtual void InitBoundaries() = 0;
     void CopyTo(ADomain& Dom);
     ADomain* Duplicate();
     ADomain* Split(int width);
     virtual void CalcInitials(ACompData *pComp);
```

```
public @[IterativeCalc]:
     virtual void PreIteration(ACompData *pComp);
     virtual void CycleIteration(ACompData *pComp);
public:
     inline int minX() @[X] { return X;}
     inline int minY() @[Y] { return Y;}
     inline int maxX() @[X] { return X + Width - 1;}
     inline int maxY() @[Y] { return Y + Height - 1;}
public @[InOut]:
     void ReadFrom(XArchList *pList, bool FullFlag);
     void WriteTo(XArchList *pList, bool FullFlag);
public:
     @[X] int X;              // координаты начала
     @[Y] int Y;              // координаты начала
     int @[Constant] Width;   // ширина домена
     int @[Constant] Height;  // высота домена
     bool @[ControlFlow] Closed;   // признак закрытости
     int @[Constant] NeedLeftWidth;
     int @[Constant] NeedRightWidth;
public @[ControlFlow]: // граничные области
     ABoundary* Left;
     ABoundary* Right;
     ABoundary* Top;
     ABoundary* Bottom;
};
#endif // ADOMAIN_H_INCLUDED
```

ПРИЛОЖЕНИЕ 2

Файла спецификаций принципов геометрической декомпозиции:

```
DecompositionManifest {
     ProjectName:GasMovement
     MainFile:GasMovement.cpp
     OutputDir:DecomposedTask
Dimensions(3) {
     X (decomposable){
          Min:0
          Max:dynamic
     }
     Y (decomposable){
          Min:0
          Max: dynamic
     }
     Z (solid){
          Min:0
          Max:1
```

```
     }
}
Stages {
     Initialization:Repeated
     IterativeCalc:Parallel
     Finalization:Exclusive
}
Algo {
     InOut {
          Initialization
          Finalization
     }
}
Decomposition (IterativeCalc){
     Dimension:X
     Dimension:Y
     Algo :SquareBlock
     LoadBalancing :Static
     WeightFunction :Area
}
Restrictions {
     Deps{
          Dst(X=i){
          Src(X){
               Min:i-2
               Max:i+2
               }
          }
          Dst(Y=j){
          Src(Y){
               Min:j-1
               Max:j+1
               }
          }
          Dst(Z=i){
          Src(Z){
               Min:1-i
               Max:1-i
               }
          }
     }
}
}
```

СПИСОК ЛИТЕРАТУРЫ

1. *В.В. Воеводин, Вл.В. Воеводин* Параллельные вычисления. // СПб.:БХВ-Петербург, 2002г, 602с.

2. *Анисимов Н.А., Голенков Е.А., Харитонов Д.И.* Композициональный подход к разработке параллельных и распределенных систем на основе сетей Петри. // Программирование, №6, 2001г, стр. 30-43.
3. *Тарасов Г.В., Харитонов Д.И., Голенков Е.А.* Об одном представлении функции в модели императивной программы, заданной сетями Петри. // Моделирование и анализ информационных систем, Т. 18, №2 (2011), с. 18-38.
4. *Харитонов Д.И., Шиян Д.С.* Влияние синхронных и асинхронных взаимодействий на производительность параллельной программы. // Материалы седьмой международной конференции-семинара "Высокопроизводительные параллельные вычисления на кластерных системах" Нижний Новгород, 26-30 ноября 2007 г., стр. 314-320
5. *Харитонов Д.И.* Раздельная верификация объектно-ориентированных программ с построением протокола C++ класса в терминах сетей Петри. // Моделирование и анализ информационных систем, Том 16, Номер 1, 2009. С. 92-112.
6. *Харитонов Д.И., Шиян Д.С.* Анализ процесса распараллеливания программы. // Программные продукты и системы, Номер 1, 2009. С. 20-22.