УДК 519.682

TREEDL: ЯЗЫК ОПИСАНИЯ ГРАФОВЫХ СТРУКТУР ДАННЫХ И ОПЕРАЦИЙ НАД НИМИ

А. В. Демаков[1]

Рассматриваются методы представления и обработки графов в программах на объектно-ориентированных языках программирования. Анализируются основные возможности специализированного языка описания структуры графов и его инструментальной поддержки. Описывается язык TreeDL, обладающий такими возможностями. Предлагаемое решение сравнивается с другими средствами описания структуры графов.

1. Введение. Графы являются естественным и наглядным средством представления сложных структур и процессов. Это позволяет широко использовать их в компьютерных системах при решении различных задач [1 – 3].

Во многих задачах графы используются для представления данных неоднородной структуры. Соответственно, структура таких графов также может быть неоднородна и нагружена дополнительной информацией. К таким задачам относятся:

— расчет гидродинамических характеристик физических моделей;

— представление графовых моделей программ в трансляторах, в том числе деревьев абстрактного синтаксиса программ;

— представление объектных моделей структурированных или размеченных документов (например, XML документов);

— разработка объектных моделей данных предметной области, в частности при реализации графического пользовательского интерфейса по схеме Model–View–Controller (MVC).

Обеспечение заданной структуры графа и поддержание соответствия между структурой графа и определениями операций над ним в процессе разработки и развития программы является одной из основных задач методов представления графов и реализации алгоритмов их обработки.

При использовании для представления и обработки графов объектно-ориентированных языков общего назначения многие аспекты корректности описания структуры графов и соответствия между структурой графов и определениями операций поддерживаются на уровне соглашений и стиля программирования, а не на уровне языка, поэтому соответствующие требования не проверяются на этапе трансляции программы. Прежде всего имеются в виду обладающие схожим набором возможностей языки Java [6] и C# [7]. Несмотря на то что язык C++ имеет существенные отличия от упомянутых языков в организации наследования и управлении памятью, основные положения данной работы применимы и к этому языку. Указанные недостатки приводят к увеличению трудоемкости разработки и снижению надежности программного обеспечения.

С увеличением сложности задач растет и необходимость создания методов представления и обработки графов, которые облегчают разработку программного обеспечения. Подход, предлагаемый в этой статье, основан на использовании специализированного языка описания структуры графов и операций над ними.

2. Цели и задачи. Целью настоящей статьи является рассмотрение такого метода представления и обработки графов, который предоставляет удобные способы представления графов и определения операций над ними, а также обеспечивает проверку корректности описания структуры графов и операций. Для достижения указанной цели:

— в разделе 3 обсуждаются широко распространенные методы представления графов с использованием объектно-ориентированных языков программирования общего назначения; показано, что такие языки не обладают всеми необходимыми возможностями для определения структуры графов и операций над ними;

— в разделе 4 описаны предлагаемый специализированный язык TreeDL, обладающий перечисленными в разделе 3 необходимыми возможностями, и его инструментальная поддержка;

[1] Институт системного программирования РАН, Б. Коммунистическая, 25, 109004, Москва; e-mail: demakov@ispras.ru

— в разделе 5 возможности языка TreeDL сравниваются с возможностями существующих языков описания структуры данных.

3. Методы объектно-ориентированного описания графовых структур данных.

3.1. Общие сведения. Далее приведены общие определения, использующиеся в данной работе. Формальные определения можно найти в книгах по теории графов (см., например, [4, 5]).

Граф — это множество вершин, некоторые пары которых соединены *ребрами*. Вершины, соединенные ребром, называются *смежными* или *соседними* вершинами.

Ориентированный граф — это граф, имеющий *ориентированные* ребра (или *дуги*). Вершины, соединенные дугой, называют *началом* и *концом* дуги.

Атрибутированный граф — это граф, с каждой вершиной которого связан набор атрибутов. Определение атрибутированного графа не включает возможность атрибутирования ребер, поскольку любой граф с атрибутированными ребрами легко заменяется на эквивалентный граф без атрибутированных ребер. Для этого атрибутированное ребро заменяется на вершину, связанную с двумя вершинами, которые в исходном графе были соединены этим ребром. Атрибуты ребра становятся атрибутами вершины.

Типизированный атрибутированный граф — это атрибутированный граф, каждая вершина которого имеет тип, определяющий структуру набора соседних вершин и структуру набора атрибутов, связанных с вершиной. Любой граф может рассматриваться как типизированный атрибутированный граф.

3.2. Представление графов в программах на объектно-ориентированных языках.

3.2.1. Структуры данных для представления графов. При использовании объектно-ориентированного подхода [8 – 10] граф естественным образом представляется объектом, интерфейс которого составляют операции, определенные над графом. Рассмотрим возможные пути реализации класса, соответствующего графам с определенной структурой и набором операций.

В рамках объектно-ориентированного подхода естественным представлением типизированных атрибутированных графов заданной структуры является следующее часто используемое представление:

— вершинам графа соответствуют объекты, класс или классы которых реализуют типы вершин графа;

— дуга ориентированного графа представляется ссылкой, которую предоставляет вершина-начало дуги; эта ссылка указывает на вершину-конец дуги; неориентированное ребро может быть представлено двумя взаимно-обратными ссылками;

— значения атрибутов вершин графа также хранятся объектами-вершинами графа;

— средства типизации, предоставляемые языком программирования, используются для задания структуры набора атрибутов и соседних вершин.

3.2.2. Операции над графами. Операция над графом в общем случае требует выполнения над каждой из его вершин определенных действий, зависящих от типа вершины. Естественный способ задания таких действий в объектно-ориентированных языках — это определение в каждом классе вершин виртуального метода, выполняющего соответствующие действия.

Однако такой способ задания операций обладает следующими существенными недостатками, которые обсуждались многими авторами.

1) Расширение набора типов вершин графа путем создания подтипов приводит к ошибкам в реализации операций, поскольку по умолчанию подтипы наследуют методы, реализующие операцию в супертипах, а эта реализация чаще всего не подходит подтипам. Иными словами, отсутствие переопределения метода в данном случае является потенциально ошибочной ситуацией, которая, однако, при использовании большинства существующих объектно-ориентированных языков программирования не приводит к ошибке или предупреждению на этапе компиляции.

2) Расширение набора операций приводит к необходимости изменения всех типов вершин графа для определения соответствующего виртуального метода. В системе, рассчитанной на подобное расширение, изменения должны носить локальный характер.

3) Операции, реализация которых зависит не от одного, а от нескольких типов вершин, требуют весьма сложной схемы диспетчеризации — последовательного вызова методов для определения фактического типа каждого параметра. Такое решение обладает значительной трудоемкостью и является потенциальным источником ошибок.

Первые две проблемы могут быть решены с использованием шаблона проектирования *Посетитель* (*Visitor*) [11]. При этом все методы, реализующие операцию, собираются в один класс Посетитель, а выбор необходимого для обработки вершины метода осуществляется на этапе выполнения программы с помощью механизма *двойной диспетчеризации* (*double dispatch*) [12]. При использовании шаблона Посетитель:

— расширение набора типов вершин графа требует определения нового метода в классе Посетитель; наличие этого метода может быть проверено средствами языка программирования;

— расширение набора операций требует создания одного дополнительного класса Посетитель, т.е. изменения в системе при таком расширении носят локальный характер.

Порядок выполнения действий над вершинами в общем случае определяют сами действия. Однако во многих частных случаях этот порядок может быть отделен от действий и выделен в самостоятельный класс *Итератор* (*Iterator*) [11]. Такая декомпозиция позволяет комбинировать действия и применять их в различном порядке [13].

Следующие недостатки шаблона Посетитель также многократно обсуждались в различных работах.

1) Необходимость изначальной поддержки этого шаблона со стороны типов вершин.

2) Фиксированная сигнатура методов, которая затрудняет передачу параметров и возврат результатов.

3) Возможность определения только унарных операций над графом, т.е. операций, выбор реализации которых зависит от фактического класса одного параметра.

4) Необходимость определения методов для всех типов вершин, даже если логически операция определена только на подтипах определенного типа. Применение операции к вершинам других типов недопустимо и является ошибкой в программе, но средствами языка реализации невозможно обнаружить эту ошибку на этапе компиляции.

5) Предыдущая проблема часто решается введением суперкласса, определяющего реализацию операции по умолчанию. Подклассы используют эту реализацию, переопределяя только часть методов. Это приводит к возникновению ошибок при изменении набора типов вершин — при появлении нового типа вершин и определении в суперклассе реализации по умолчанию для него недостаточно средств языка реализации для получения информации о том, какие операции требуется доопределить на этом типе вершин. Как и при использовании виртуальных методов для определения операций, в этом случае не хватает средств языка программирования для указания методов, которые должны быть переопределены при наследовании классов.

3.3. Необходимые возможности языка описания графовых структур данных. В этом подразделе рассмотрены необходимые возможности языка описания графовых структур данных и их поддержка объектно-ориентированными языками программирования общего назначения.

3.3.1. Типизация вершин графа. Многие возможности языка описания структуры графов необходимы и языкам описания структуры данных общего назначения. Перечислим некоторые общие возможности, которые присутствуют в рассматриваемых здесь объектно-ориентированных языках общего назначения.

1) Декларативное задание типов данных — описание правил работы с данными в форме, пригодной для статического анализа программы. Это позволяет выявить многие ошибки еще до выполнения программы. На такой форме задания типов основаны системы контроля типов языков программирования со строгой типизацией. Применительно к описанию структуры графов, декларативное задание типов вершин графа позволяет для каждого типа вершины указать допустимый набор соседних вершин и атрибутов.

2) Императивное задание контекстных ограничений на совокупность данных или инвариантов, которые не могут быть заданы средствами декларативной системы типов.

3) Возможность расширения декларативной системы типов языка дополнительной декларативной информацией, правила использования которой могут быть заданы пользователем. В языке Java дополнительная декларативная информация задается аннотациями, а в языке C# — атрибутами. Такая возможность во многих случаях позволяет обойтись без императивного задания ограничений путем переноса проверок времени выполнения программы на этап анализа. Это способствует повышению надежности программы. Примерами такого использования дополнительной декларативной информации являются ссылки, не допускающие нулевые значения [14], а также ссылки с ограничением доступа и немодифицирующие методы [15, 16].

4) Наследование типов данных, которое обеспечивает полиморфную обработку данных различных типов и повторное использование реализации типов данных. Для полиморфной обработки достаточно наследования интерфейсов. Повторное использование реализации требует наследования не только интерфейса, но и самой реализации. Различают одиночное и множественное наследование. Одиночное наследование допускает использование лишь одного предка. Множественное наследование ограничений на количество предков не накладывает. Множественное наследование реализации может приводить к конфликтам, если граф наследования не является деревом. Поэтому в языках Java и C#, в отличие от C++,

используется множественное наследование интерфейсов, но одиночное наследование реализации. Для множественного наследования реализации может быть использовано делегирование [11].

Специфика типов вершин графа состоит в том, что они описывают способ хранения вершины графа, но не операции над графом. Как было изложено ранее, использование методов типов вершин для реализации операций над графом обладает значительными недостатками. Поэтому для реализации операций целесообразно использовать другие подходы, ограничив интерфейс типа вершины базовыми методами доступа к информации о вершине, т.е. к полям, в которых хранятся атрибуты и ссылки на соседние вершины.

Очевидно, что реализация базовых методов доступа (чтения/записи) к полю типа вершины определяется декларацией самого поля. Современные среды разработки программ на языке Java предоставляют возможность автоматического создания базовых методов доступа к выбранным полям. Таким образом, для определения типа вершины достаточно декларативно задать набор полей. Эта информация одновременно описывает и интерфейс, и реализацию типа вершины. На основании изложенного рассуждения можно сделать вывод, что язык описания структуры графов может предоставить возможности более компактного описания типов данных по сравнению с языками программирования общего назначения.

Перечислим возможности, которые может предоставлять язык описания структуры графов при таком подходе.

1) Декларативное описание типов вершин в виде набора полей, которое определяет и структуру данных, и базовые методы доступа к данным: чтение/запись полей вершины, конструкторы, итераторы соседних вершин и т.п. Такое описание компактно и хорошо подходит для статического анализа описания структуры графа.

2) Возможность задания дополнительной декларативной информации, относящейся к отдельным полям и типам в целом. Правила использования этой информации задает пользователь. Эта информация может быть использована как для расширения декларативной системы типов, так и для уточнения базовых методов доступа к данным.

3) Множественное наследование типов вершин графа. Поскольку тип вершин графа определяет интерфейс вершин, то множественное наследование типов вершин, определенное в соответствии с правилами множественного наследования интерфейсов, не приводит к конфликтам. Наследование типов вершин определяет и наследование реализации.

3.3.2. Определение операций над графами. Специализированный язык описания структуры графов дает возможность использовать для реализации операций наиболее подходящие средства, свободные от недостатков применения виртуальных методов или шаблона проектирования Посетитель.

Одним из таких средств определения операций являются мультиметоды [17 – 19]. Мультиметоды расширяют предоставляемую обычными виртуальными методами возможность динамического выбора вызываемого метода.

Сигнатура виртуального метода определяется по статическим типам параметров вызова метода, их фактические типы значения не имеют. Тип возвращаемого значения виртуального метода допускает *ковариантные* изменения, т.е. при переопределении виртуального метода в подтипе тип возвращаемого значения может быть заменен своим подтипом. Очевидно, что это не нарушает контракт метода супертипа, т.е. возможность вызова переопределенного метода везде, где можно вызвать метод супертипа. Сигнатуры мультиметодов допускают и *контравариантные* изменения, т.е. тип параметра может быть заменен подтипом. Выбор вызываемого мультиметода происходит на основе фактических типов параметров.

При использовании мультиметодов расширение набора операций также носит локальный характер — добавление новой операции требует определения семейства мультиметодов, которые в зависимости от используемого языка программирования либо принадлежат одному классу, либо не принадлежат никакому классу.

Рассмотрим, как мультиметоды помогают избежать проблем, возникающих при использовании шаблона проектирования Посетитель.

1) Для использования мультиметодов не требуется поддержка со стороны типов вершин. Семейство мультиметодов может быть определено для любой существующей иерархии типов.

2) Мультиметоды могут иметь произвольную сигнатуру, что облегчает передачу параметров и возврат результатов.

3) Произвольное количество параметров мультиметодов и выбор реализации в зависимости от фактических классов всех параметров позволяют использовать мультиметоды для определения операций с произвольной арностью, т.е. операций, выбор реализации которых зависит от фактических классов лю-

бого количества параметров.

4) Мультиметоды задаются и уточняются только на тех типах вершин, на которых логически определена операция.

Недостатком использования мультиметодов для определения операций над графами является отсутствие контроля над переопределением мультиметодов для каждого типа вершин. Реализация мультиметодов в языках программирования общего назначения предполагает наличие у типа параметра мультиметода неопределенного количества подтипов. Отсутствие мультиметода с точной сигнатурой для какого-либо подтипа параметра не является ошибкой с точки зрения языка программирования — при этом используется реализация с более общей сигнатурой. Как было указано ранее, в случае вершин графа эта реализация чаще всего не обрабатывает вершину корректно. При использовании мультиметодов эта потенциально ошибочная ситуация не выявляется средствами языка программирования.

Существует достаточно много языков программирования, поддерживающих использование мультиметодов [20–23], однако в наиболее широко распространенных языках эта конструкция не поддерживается.

3.3.3. Модульность описания структуры графов. Одним из важнейших методов облегчения разработки и сопровождения программного обеспечения является повторное использование программных компонентов. Описание структуры графа или его часть, имеющая самостоятельную ценность, могут рассматриваться в качестве модуля, пригодного для повторного использования. Такая возможность может быть обеспечена средствами языка описания структуры графов, позволяющими явно определять модули и связи между ними.

Выделение группы типов в самостоятельную сущность языка программирования и определение отношений между группами предлагается в работах [24–27]. Такой подход позволяет справиться с проблемами расширения групп, плохо поддающимися решению в рамках языков программирования, система типов которых допускает наследование только на уровне отдельных типов.

Явное определение всех допустимых типов вершин графа позволяет преодолеть недостатки, свойственные мультиметодам языков программирования общего назначения, которые определены на неограниченном множестве типов. При определении операции на графах с помощью мультиметода можно проверить, что для каждого допустимого типа вершин графа определена реализация мультиметода с точной сигнатурой.

3.4. Требования к инструментальной поддержке. Использование специализированного языка описания данных требует инструментальной поддержки. Традиционно основой инструментальной поддержки является транслятор, преобразующий описание данных в код на языке программирования общего назначения, совместно с которым используется язык описания данных. Возможно и непосредственное создание исполнимых модулей целевой машины, библиотечных или объектных модулей и т.п.

Помимо трансляции описания структуры данных перед инструментами обработки специализированного языка описания данных стоят и другие задачи генерации кода. Описание структуры данных содержит достаточно информации для генерации различных программных компонентов, решающих различные задачи обработки этих структур данных, например:

— сериализация/десериализация структуры данных для долговременного хранения;

— обход графа с возможностью выполнения заданных действий в каждой вершине;

— сравнение;

— копирование.

Возможность задания в описании структуры данных дополнительной декларативной информации расширяет набор программных компонентов, которые могут быть сгенерированы автоматически. Программные компоненты, которые необходимо сгенерировать, и способ их реализации могут зависеть от конкретной задачи. Следовательно, у пользователей специализированного языка может возникнуть потребность в расширении набора генерируемых программных компонентов.

Таким образом, инструмент обработки специализированного языка описания структуры данных должен иметь открытую архитектуру, т.е. допускать возможность расширения функциональности сторонним разработчиком как с целью расширения набора программных компонентов, генерируемых по описанию структуры данных, так и с целью поддержки новых целевых языков программирования.

Инструментальная поддержка современных языков программирования не ограничена компилятором и отладчиком, как это было несколько десятилетий назад. Современные интегрированные среды разработки обеспечивают поддержку создания программ, в том числе и без редактирования исходных текстов, с использованием визуальных средств разработки, навигацию по исходным текстам, поддержку их согласованной модификации (рефакторинга [28]) и т.п.

Использование дополнительного специализированного языка, не обладающего соответствующей поддержкой среды разработки, может существенно затруднить процесс разработки, сводя на нет преимущества использования этого языка.

Таким образом, инструментальная поддержка специализированного языка описания структуры данных помимо транслятора с расширяемой функциональностью должна включать в себя средства интеграции с распространенными средами разработки.

4. Язык описания графовых структур данных и операций над ними. В этом разделе описан предлагаемый специализированный язык описания графовых структур данных и операций над ними TreeDL, обладающий перечисленными в предыдущем разделе необходимыми возможностями. Кроме того, описана реализованная инструментальная поддержка этого языка.

4.1. Типизация вершин графа. Основой типовой системы языка TreeDL являются типы вершин графа, которые задаются как именованные *записи* (*record*). Этот тип данных часто называют *структурой*, возможно из-за ключевого слова `struct`, которое используется для обозначения таких типов данных в языке C. Чтобы не создавать конфликта с понятием *структура данных*, в этой работе используется термин *запись* (*record*). Запись состоит из набора *полей*, которые задают набор *атрибутов* и *соседних вершин*, допустимых для вершин данного типа. Для полей определены *операции получения значения* и *установки нового значения*, аналогично *свойствам* (*properties*) в языке C# или в связанной с языком Java технологии JavaBeans. Операции полей вершины составляют интерфейс типа вершин (рис. 1).

```
type VarDecl {
    Type varType;
    string name;
}
```

Рис. 1. Пример типа вершины

Язык TreeDL предоставляет набор предопределенных типов, соответствующий системе типов современных объектно-ориентированных языков: `bool, char, short, int, long, float, double, string, object, node`. Единственным дополнением к системе типов объектно-ориентированных языков общего назначения, которое отражает специфику графовых структур данных, является тип node. Этот тип позволяет среди всех объектных типов выделить типы вершин. В дополнение к предопределенным типам в описании структуры графа могут использоваться перечислимые типы, которые имеют явно заданное множество именованных значений.

Описанные базовые возможности определения типов вершин графа не зависят от конкретного языка программирования. При необходимости использования типов данных языка программирования могут применяться пользовательские типы, определенные на языке программирования (рис. 2).

```
type File = { java.io.File }
```

Рис. 2. Пример определения пользовательского типа

Определение таких типов относится к расширенным возможностям языка описания структуры графов TreeDL.

На основе существующих типов данных могут быть построены типовые выражения, используемые при определении типов вершин графа и операций над ними.

Пусть `T` — предопределенный тип, перечислимый тип, пользовательский тип или тип вершин. На его основе могут быть построены следующие типовые выражения.

1) `T` — поле такого типа содержит ссылку на объект типа `T`. Эта ссылка является обязательной, т.е. в отличие от большинства языков программирования она обязательно должна указывать на объект такого типа. Если тип не является ссылочным, поле содержит само значение.

2) `T?` — поле такого типа содержит необязательную ссылку на объект типа `T`. Значением ссылки является объект типа `T` или специальное значение `null`. Для типов, которые не являются ссылочными, модификатор ? не меняет набора возможных значений. Контроль использования `null`-значения приводит к исчезновению или сокращению числа ошибок, связанных с попыткой доступа к объекту по ссылке, имеющей такое значение. В зависимости от возможностей целевого языка программирования информация о запрете `null`-значения может использоваться либо для статического анализа, либо для генерации проверок времени выполнения.

3) $\texttt{T}^*$ — список из 0 или более значений типа `T`.

4) `T+` — список из 1 или более значений типа `T`.

5) $\texttt{T}_1$:$\texttt{T}_2$ — отображение, ключами которого являются значения типа $\texttt{T}_1$, а значениями — значения типа $\texttt{T}_2$.

4.2. Наследование типов вершин графа. В отличие от современных языков программирования Java и C# в языке TreeDL допустимо множественное наследование, поскольку это позволяет более точно описывать структуру графа. Для разрешения известного конфликта, возникающего при множественном наследовании, независимо от количества путей в иерархии наследования данные наследуются в одном

экземпляре.

При наследовании типов вершин могут быть определены дополнительные поля и переопределены некоторые свойства унаследованных полей. Переопределение может дополнять, уточнять или усиливать свойства поля, но не нарушать свойства, заданные ранее. Например, при переопределении можно потребовать инициализацию поля при создании вершины, если ранее этого не требовалось, но тип поля изменить нельзя.

4.3. Модульность описания структуры графа. Описание структуры графа или его часть, имеющая самостоятельную ценность для повторного использования, оформляются в виде именованных модулей (рис. 3).

```
[ treedl.language.java ]
structure ast.Variable;
type Type {
    string name;
}
type VarDecl {
    child Type type;
    string name;
}
type VarExpr {
    string name;
    late VarDecl decl;
}
```

Рис. 3. Пример определения модуля

```
feature ast.ToString : ast.Variable;
operation string toString( virtual node n ) {
    case ( Type n ):
    case ( VarExpr n ):
    { return n.getName(); }
    case ( VarDecl n ):
    {
        return toString( n.getType() )
      + " " + n.getName();
    }
}
```

Рис. 4. Пример определения операции

Модули могут использовать друг друга. Типы вершин используемых модулей могут применяться следующими способами.

1) Расширение — определение новых типов вершин, унаследованных от типов вершин используемого модуля.

2) Использование — определение в новых типах вершин полей, типы которых определены в используемом модуле.

3) Доопределение — добавление определенным в используемом модуле типам вершин базовых типов вершин и полей.

Фактически модуль описания структуры графа является самостоятельным типом данных, описывающим графы заданной структуры. Далее рассмотрен предоставляемый языком TreeDL способ определения операций над такими типами данных.

4.4. Определение операций над графами. Определение операций над графом в языке TreeDL осуществляется с помощью мультиметодов. По сравнению с общим случаем определение мультиметодов на типах вершин графа позволяет преодолеть отсутствие контроля над определением специализации мультиметода для каждого возможного типа вершин. В случае определения операции над графом набор возможных типов вершин известен заранее, и при анализе определения мультиметода может быть проверена полнота определения, т.е. наличие специализации мультиметода для каждого наследника типа его параметра.

Определение операции состоит из сигнатуры и тела (рис. 4). Сигнатура операции аналогична сигнатуре обычного метода и определяет имя, набор типов входных параметров и тип возвращаемого значения. В отличие от обычных методов некоторые параметры операции могут быть помечены как виртуальные. Тип виртуального параметра может быть типом вершин, перечислимым или булевским типом.

Для каждого возможного набора фактических типов или значений виртуальных параметров должна быть определена своя реализация операции — *ветвь операции*. Возможными типами виртуального параметра, тип которого является типом вершин, считаются все неабстрактные наследники формального типа виртуального параметра, видимые в модуле определения операции, включая сам тип. Возможными значениями виртуальных параметров перечислимого типа являются все константы этого типа, булев-

ского типа — значения `true` и `false`.

Ветвь операции состоит из метки и блока кода на языке программирования. Блок кода разделяется несколькими ветвями операции, метки которых находятся непосредственно перед блоком. Разделение блока кода несколькими ветвями операции является сокращением такой записи, когда каждой ветви операции соответствует своя текстуальная копия блока кода.

В зависимости от фактического типа или значения параметра выполняется блок кода, соответствующий ветви операции. После выполнения блока кода выполнение операции завершается, переход к следующему блоку кода не происходит, даже если оператор возврата управления в блоке кода отсутствует.

Семантика операции, единственным параметром которой является виртуальный параметр перечислимого типа, аналогична семантике конструкции `switch` языков программирования.

Операции над модулем описания структуры, который использует другие модули описания структуры, могут быть определены как расширение операций над используемыми модулями. При этом требуется определить только ветви операции, не покрытые расширяемыми операциями.

В рамках выбранного подхода, т.е. без фиксации целевого языка программирования, полный контроль проблем, связанных с расширением модулей, осуществить невозможно, поскольку код на языке программирования не анализируется.

Использование описанного способа определения операций над графами предотвращает возникновение ошибок, связанных с изменением описания структуры графа. Предположим, что в модуль описания структуры был добавлен новый тип вершин, который является наследником типа вершин, использованного в качестве типа виртуального параметра некоторых операций. При проверке корректности определения этих операций возникнет сообщение об ошибке, что для этого типа вершин не определена ветвь операции. Аналогичное сообщение возникнет при удалении типа вершины — в этом случае тип, используемый в метке ветви операции, окажется не определен. Трудоемкость внесения изменений в реализацию операций при добавлении нового типа вершины неизбыточна: если обработка нового типа вершины текстуально совпадает с одним из блоков, использующихся ветвями операции, достаточно перед этим блоком добавить метку ветви операции. Если требуется специальная обработка нового типа вершины, то следует реализовать ее в новом блоке.

4.5. Дополнительная декларативная информация. Для каждой сущности в описании структуры графа возможно указать дополнительную декларативную информацию. Это позволяет более детально описать структуру данных без изменения самого языка описания.

Декларативная информация доступна на этапе обработки структуры графа и может быть использована инструментом обработки. Декларативная информация задается в виде набора именованных свойств булевского, целочисленного или строкового типов. Имена свойств являются квалификационными идентификаторами, что позволяет избежать конфликтов имен. Пример описания дополнительной декларативной информации приведен на рис. 5.

```
[ check.final ]
[ print.pattern = "($exprList;separator=',')" ]
type ProductExpr {
    [ assert.length.min = 2 ]
    child Expr* exprList;
}
```

Рис. 5. Указание декларативной информации

Декларация типа вершин, представляющих абстрактный синтаксис прямого произведения выражений, дополнена тремя свойствами.

1) Булевское свойство `check.final` аналогично модификатору `final` языка программирования Java — оно сообщает о запрете создания подтипов данного типа вершин. Аналогичные свойства могут быть определены для всех необходимых модификаторов. Соответствующие проверки выполняются подключаемыми анализаторами описания структуры данных.

2) Строковое свойство `print.pattern` определяет соответствие между абстрактным и конкретным синтаксисом, которое может быть использовано при генерации компонента, осуществляющего это преобразование.

3) Целочисленное свойство `assert.length.min` сообщает о необходимости проверки при создании вершины длины списка `exprList`. В отличие от свойства `check.final`, это указание не анализатору описания структуры данных, а генератору кода.

Приведенный пример показывает, что задание дополнительной декларативной информации является мощным механизмом, который не только расширяет возможности системы типов языка описания графовых структур данных, но и может быть использован для генерации компонентов, осуществляющих

различную обработку данных.

4.6. Инструментальная поддержка языка TreeDL. Для обработки описаний графовых структур данных на языке TreeDL разработан инструмент `treedl`, реализованный на языке Java. Основным назначением этого инструмента является:

— анализ корректности описания структуры данных и операций над ними на языке TreeDL;

— генерация программного кода по описанию структуры данных на языке TreeDL, в частности, трансляция описания структуры данных и операций над ними в язык программирования, а также генерация реализации операций, которые зависят только от структуры данных.

Функциональность инструмента `treedl` может быть расширена подключаемыми модулями, реализующими дополнительные возможности анализа описания структуры графа и генерации программных компонентов.

Для языка TreeDL разработан модуль интеграции с популярной средой разработки на платформе Eclipse [29], поддерживающий создание описания структуры графа и его трансляцию в код на языке программирования.

5. Сравнение языка TreeDL с другими языками описания структуры данных. Актуальность задач описания структуры данных подтверждается большим количеством языков и инструментов, нацеленных на решение таких задач. Средства описания структуры данных имеются как в языках программирования и языках спецификации, так и в специализированных языках.

Важным частным случаем описания структуры данных является описание структуры деревьев абстрактного синтаксиса программ и структуры объектных моделей документов. Современные инструменты, автоматизирующие создание трансляторов, обеспечивают не только генерацию анализаторов языка, но и поддерживают работу с внутренним представлением программ, в частности, описание структуры внутреннего представления (деревьев абстрактного синтаксиса или объектных моделей) и определение операций над ним.

Выразительность языка описания структуры данных находится в прямой зависимости от его сложности. Языки, предоставляющие богатые возможности описания структуры данных, обладают значительной сложностью (fSDL [30], EXPRESS [31]). Сложность языка описания структуры данных и отличие принципов построения его типовой системы от принципов построения типовой системы используемого языка программирования могут затруднить изучение и использование языка описания структуры данных. Поэтому в языке TreeDL присутствуют лишь возможности декларативного задания типов, знакомые разработчикам по языкам программирования. Кроме того, в языке TreeDL присутствуют контейнеры (списки, отображения), поддержка которых в языках программирования для повышения производительности обычно осуществляется на уровне библиотеки. Транслятору TreeDL указание наиболее эффективной в каждом конкретном случае реализации контейнеров может быть задано с помощью механизма дополнительной декларативной информации.

Механизмы задания дополнительной декларативной информации существуют как в языках программирования общего назначения (аннотации в Java, атрибуты в C#), так и в специализированных языках (опции в ANTLR [32]).

Мультиметоды для определения операций над графами используются в инструменте `treecc` [33]. Управление модульностью в этом инструменте основано на файлах (аналогично языку C) — исходный код модуля состоит из всего содержимого файла, в который при помощи соответствующих директив может быть подставлено содержимое других файлов. Для определения нового модуля необходимо создать файл с дополнительными определениями типов и операций и включить в него файлы используемых модулей.

Определения не квалифицируются именами модулей, т.е. все определения находятся на глобальном уровне. Это может привести к конфликтам имен и невозможности переиспользования отдельных частей описания структуры данных, т.е. препятствует достижению одной из основных целей модульности (см. подраздел 3.3.3).

Несмотря на отмеченные недостатки, определение операций с помощью мультиметодов в языке `treecc` является мощным средством, отсутствующим в других языках описания данных.

Инструмент `treecc` поддерживает четыре целевых языка программирования: C, C++, Java и C#. Однако для этого инструмента не выполнены описанные в подразделе 3.4 требования — затруднено расширение функциональности инструмента сторонними разработчиками (в том числе в связи с отсутствием в языке возможности задания дополнительной декларативной информации), а также отсутствуют средства интеграции со средами разработки.

6. Заключение. В статье рассматриваются методы представления и обработки графов. Проанализированы необходимые возможности языка и инструментальной поддержки для решения этой задачи и

их присутствие в объектно-ориентированных языках общего назначения.

Представлен разработанный автором язык TreeDL и его инструментальная поддержка, обладающие необходимыми возможностями. Проведено сравнение языка TreeDL с существующими решениями. Это сравнение показало, что набор возможностей языка TreeDL не покрывается ни одним из существующих средств описания структуры графов.

Различные версии описания абстрактного синтаксиса, послужившие основой для языка TreeDL, были использованы в Институте системного программирования РАН в 1995–2006 гг. в следующих проектах.

1. При реализации трансляторов языка спецификации RSL и спецификационных расширений языков C, Java и C# язык TreeDL использовался для описания структуры абстрактного синтаксиса входного языка трансляторов [34–36].

2. При реализации модуля интеграции спецификационного расширения языка C# в среду разработки Microsoft® Visual Studio .NET 2003 язык TreeDL использовался для описания модели данных, создаваемой с помощью графического интерфейса.

3. В проекте по разработке генератора тестов на основе моделей язык TreeDL использовался в качестве языка описания моделей [37, 38].

4. В инструменте генерации тестов для языков программирования [39].

5. При реализации инструмента обработки самого языка TreeDL.

Результаты, описанные в этой статье, являются развитием результатов, опубликованных в работе [40]. Реализация инструментов для языка TreeDL свободно доступна на странице проекта [41].

СПИСОК ЛИТЕРАТУРЫ

1. *Кнут Д.* Искусство программирования. Т. 1. Основные алгоритмы. М.: Вильямс, 2000.
2. *Ахо А., Хопкрофт Дж., Ульман Дж.* Структуры данных и алгоритмы. М.: Вильямс, 2001.
3. *Касьянов В.Н., Евстигнеев В.А.* Графы в программировании: обработка, визуализация и применение. СПб.: БХВ-Петербург, 2003.
4. *Емеличев В.А., Мельников О.И., Сарванов В.И., Тышкевич Р.И.* Лекции по теории графов. М.: Наука, 1990.
5. *Дистель Р.* Теория графов. Новосибирск: Изд-во ИМ СО РАН, 2002.
6. *Gosling J., Joy B., Steele G., Bracha G.* The Java^{TM} language specification. Reading: Addison–Wesley, 2005.
7. Standard ECMA-334 C# Language Specification. Third edition. June 2005 (http://www.ecma-international.org/publications/standards/Ecma-334.htm).
8. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. М.: Бином, 2001.
9. *Мейер Б.* Объектно-ориентированное конструирование программных систем. М.: Интуит.ру, 2005.
10. *Грэхем И.* Объектно-ориентированные методы. Принципы и практика. М.: Вильямс, 2004.
11. *Влиссидес Дж., Джонсон Р., Хелм Р., Гамма Э.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2003.
12. *Ingalls D.H.H.* A simple technique for handling multiple polymorphism // Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86). Portland, 1986 (http://portal.acm.org/citation.cfm?id=28732).
13. *Visser J.* Visitor combination and traversal control // OOPSLA'01. Tampa, 2001 (http://portal.acm.org/citation.cfm?id=504302).
14. *Fahndrich M.K., Leino R.M.* Declaring and checking non-null types in an object-oriented language // OOPSLA'03. Anaheim: 2003 (http://research.microsoft.com/ leino/papers/krml109.pdf).
15. *Birka A., Ernst M.D.* A practical type system and language for reference immutability // OOPSLA'04. Vancouver, 2004 (http://www.sct.ethz.ch/teaching/ws2004/semspecver/papers/ernst-ref-immutability-oopsla2004.pdf).
16. *Tschantz M.S., Ernst M.D.* Javari: Adding reference immutability to Java // OOPSLA'05. San Diego, 2005 (http://pag.csail.mit.edu/ mernst/pubs/ref-immutability-oopsla2005.pdf).
17. *Mugridge W.B., Hamer J., Hosking J.G.* Multimethods in a statically-typed programming language // OOPSLA'91. New York, 1991 (http://portal.acm.org/citation.cfm?id=679202).
18. *Chambers C., Leavens G.T.* Typechecking and modules for multimethods // OOPSLA'94. Portland, 1994 (ftp://ftp.cs.washington.edu/homes/chambers/mmtc.ps.Z).
19. *Millstein T., Chambers C.* Modular statically typed multimethods // ECOOP'99. Lisbon, 1999 (http://www.ifs.uni-linz.ac.at/ ecoop/cd/papers/1628/16280279.pdf).
20. *Gabriel R.P., DeMichiel L.* The Common Lisp Object System: An overview // ECOOP'87. Paris, 1987.
21. *Chambers C.* Object-oriented multimethods in Cecil // ECOOP'92. Utrecht, 1992 (ftp://ftp.cs.washington.edu/homes/chambers/cecil-oo-mm.ps.Z).
22. The MultiJava Project (http://multijava.sourceforge.net).
23. The Nice programming language (http://nice.sourceforge.net).

24. *Ernst E.* Family polymorphism // ECOOP 2001-Object-Oriented Programming. Heidelberg: Springer-Verlag, 2001. 303–326 (http://portal.acm.org/citation.cfm?id=680013).
25. *Bruce K.B.* Some challenging typing issues in object-oriented languages // Electronic Notes in Theoretical Computer Science, 2003 (http://citeseer.ist.psu.edu/bruce03some.html).
26. *Jolly P., Drossopoulou S., Anderson C., Ostermann K.* Simple dependent types: Concord (FTfJP accepted version). April 2004
(http://myitcv.org.uk/papers/concord04.html, http://citeseer.ist.psu.edu/article/jolly04simple.html).
27. *Nystrom N., Chong S., Myers A.C.* Scalable extensibility via nested inheritance // OOPSLA'04. Vancouver, 2004 (http://citeseer.ist.psu.edu/nystrom04scalable.html).
28. *Фаулер М.* Рефакторинг: улучшение существующего кода. СПб.: Символ-плюс, 2005.
29. Eclipse platform (http://eclipse.org).
30. *Walters H.R., Kamperman J.F.Th., Dinesh T.B.* An extensible language for the generation of parallel data manipulation and control packages. Report CS-R9575. CWI SMC. Amsterdam, 1995
(http://www.cwi.nl/ftp/CWIreports/AP/CS-R9575.ps.Z).
31. Системы автоматизации производства и их интеграция. Представление данных об изделии и обмен этими данными. Часть 11. Методы описания. Справочное руководство по языку EXPRESS / Государственный стандарт РФ ГОСТ РИСО 10303–11–2000.
32. ANTLR: ANother Tool for Language Recognition (http://www.antlr.org).
33. *Weatherley R.* Treecc: An aspect-oriented approach to writing compilers // Free Software Magazine. Issue 2. 2001. (http://www.southern-storm.com.au/treecc.html).
34. *Демаков А.* Исполнимое подмножество языка спецификации и его трансляция // Приложения системного программирования: Вопросы кибернетики. Научный совет по комплексной проблеме "Кибернетика" РАН. М., 1998. Вып. 4. 17–28.
35. *Bourdonov I.B., Demakov A.V., Jarov A.A., Kossatchev A.S., Kuliamin V.V., Petrenko A.K., Zelenov S.V.* Java specification extension for automated test development // Proc. of PSI'01. Berlin: Springer-Verlag, 2001. 301–307.
36. *Демаков А.* Применение технологии UniTesK для тестирования NET-компонентов // Труды конференции "Технологии Microsoft в научных исследованиях и высшем образовании". М., 2003.
37. *Зеленов С.В., Зеленова С.А., Косачев А.С., Петренко А.К.* Генерация тестов для компиляторов и других текстовых процессоров // Программирование. **29**, № 2. 2003. 59–69.
38. *Kuliamin V., Petrenko A.K.* Applying model based testing in different contexts // Proc. of Seminar on Perspectives on Model Based Testing. Dagstuhl, 2004.
39. *Архипова М.* Генерация тестов для семантических анализаторов. Препринт ИСП РАН № 9. М., 2005.
40. *Демаков А.* Язык описания абстрактного синтаксиса TreeDL и его использование. Препринт ИСП РАН № 17. М., 2006.
41. Проект TreeDL (http://treedl.sourceforge.net).

Поступила в редакцию
25.09.2006