Применение стандарта CORBA для унаследованных систем[1]

В. Е. Каменский, А. В. Климов, С. Г. Манжелей, Л. Б. Соловская

Тенденция распределенной обработки данных в современном компьютерном мире становится все заметнее. Использование индустриальных стандартов для создания распределенных систем, таких как Object Management Group (OMG) CORBA 2.0, позволяет делать системы “открытыми” в условиях гетерогенного распределенного окружения. Хотя стандарт CORBA 2.0 рассчитан на создание новых объектно-ориентированных приложений, он может быть успешно использован и для распределения существующих объектных систем (legacy systems). Одной из задач при распределении таких систем является задача получения спецификации некоторой части системы на языке OMG Interface Definition Language (IDL).

В статье рассматриваются вопросы автоматической генерации IDL спецификации по коду унаследованной системы, а также вопросы использования полученной спецификации. Предлагается расширение языка IDL для поддержки унаследованных систем, а также “обратный” компилятор с используемого языка программирования в IDL.

1. Введение

В последнее время в компьютерном мире все заметнее становится тенденция использования распределенной обработки данных. Распределенность помогает сделать систему открытой для взаимодействия с другими системами, дает возможность повысить эффективность вычислений, наконец, позволяет использовать разные машины, операционные системы и языки программирования вместе. Интересной представляется инициатива Object Management Group (OMG), которая предложила стандарт архитектуры распределенных объектных систем - Object Management Architecture (OMA) [1] и стандарт взаимодействия объектов между собой - Common Object Request Broker Architecture, version 2.0 (CORBA) [2, 3]. В этих документах вводится понятия языка OMG Interface Definition Language (IDL), использующегося для описания интерфейсов систем, а также отображение IDL на конкретные языки программирования, такие, как C, C++, Smalltalk. Кроме этого, дается определение Object Request Broker (ORB). Это программная компонента, находящаяся между сетевым программным обеспечением и прикладными программами и призванная обеспечить “прозрачное”[2] взаимодействие объектов пользователя независимо от того, где они находятся. CORBA позволяет также разным объектным моделям сосуществовать в рамках одной системы, предоставляя разработчику возможность использовать разные языки программирования и разные окружения. Такое свойство называется прозрачностью по объектной модели [4, 5].

Перечисленные свойства CORBA позволяют не только легко создавать новые объектно-ориентированные распределенные системы, но дают возможность интегрировать такие системы между собой, а также

---

[1] Работа поддержана грантом РФФИ № 96-01-01278 “Исследование и разработка архитектурных принципов организации интегрированных распределенных неоднородных объектных систем”.

[2] “Прозрачность” в данном случае означает, что пользователь может вызывать методы объекта одинаковым образом - и в том случае, если вызываемый объект находится на той же машине, и в том случае, если объект удаленный.

модернизировать существующие объектные унаследованные системы (legacy systems). Под термином “модернизация” мы будем понимать расширение системы в сторону открытости: сторонние объекты могут использовать или быть используемы новой системой. Аналогичная задача использования унаследованного кода решается, например, в Distributed Computing Environment Remote Procedure Call (DCE RPC) [8]. Этот стандарт предлагает специальные средства для распределения существующих систем в языках DCE IDL и DCE Attribute Configuration Source (ACS). Хотя CORBA и не поддерживает явно модернизацию унаследованного кода, как показано в статье, такая поддержка может быть добавлена путем расширения языка OMG IDL.

Обычным стилем создания системы, удовлетворяющей стандарту CORBA, является разработка ее “с нуля” - создание спецификации, а затем реализации. Для унаследованных систем реализация уже существует. Поэтому при модернизации унаследованной системы самое важное - это суметь обеспечить взаимодействие ее объектной модели с моделью расширяющих объектов. Устроив такое взаимодействие с использованием технологии CORBA, мы сделаем унаследованную систему распределимой, оставив, однако, ее без изменений. Распределенная система может быть получена из распределимой путем переноса некоторых ее частей в другое окружение, что требует изменения исходных текстов последней.

Итак, чтобы модернизировать унаследованную объектно-ориентированную систему, необходимо выполнить, по крайней мере, следующие шаги:

- выделить из всей системы некоторый объектный контекст[3], который предполагается делать открытым, будь то для взаимодействия с совершенно другими системами или для взаимодействия с “вынесенными” частями существовавшей системы. Вообще говоря, этот контекст может совпадать с интерфейсом унаследованной системы, но чаще всего является его подмножеством. В этой статье мы будем рассматривать тот случай, когда система делается “открытой”, что подразумевает как экспорт ее интерфейсов, так и импорт интерфейсов других систем. Если стоит задача сделать унаследованную систему распределенной, то возникающие при этом проблемы будут аналогичными;

- создать тем или иным способом IDL спецификации для “открываемого” контекста. Однако, как мы увидим дальше, это возможно не для всех объектов;

- наконец, порожденные IDL интерфейсы нужно реализовать, используя обычную технологию написания программ в стандарте CORBA: используя IDL компилятор, получить описание интерфейсов на используемом языке программирования и реализовать эти интерфейсы, опираясь на код существующей системы. Только на этом этапе система может подвергаться изменениям.

Первый из перечисленных шагов является, пожалуй, самым трудным и под силу только разработчику, то есть человеку. Любые автоматические средства могут лишь помочь в решении этой проблемы, но не могут обеспечить полное автоматическое выделение объектных контекстов из реальных систем. Мы будем считать, что тем или иным способом, методом проб и ошибок, для унаследованной объектно-ориентированной системы выделены экспортируемые и/или импортируемые контексты, которые необходимо специфицировать на IDL.

---

[3] Под объектным контекстом мы будем понимать набор интерфейсов объектов.

Второй шаг представляется нам наиболее интересным с точки зрения возможности его автоматизации. Рассмотрению различных способов получения IDL спецификаций по коду существующей системы, возникающих при этом проблем, а также рассмотрению возможных оптимизаций и посвящена эта статья. При рассмотрении примеров мы будем пользоваться языком программирования Protel-2 [7]. Этот язык распространен мало, но имеет много общего с другими процедурными объектно-ориентированными языками программирования, поэтому проблемы, возникающие при работе с системой, написанной на Protel-2, будут общими и для других широко распространенных языков, таких как C и C++.

Последний, третий шаг, в общем случае необходим, но мы будем стремиться если не исключить, то хотя бы минимизировать его, используя написанный код унаследованной системы.

Последующие разделы статьи организованы следующим образом. Часть 2 “Как получить IDL спецификацию” посвящена ручной генерации IDL спецификации. Раздел 2.1 “Ручное получение IDL спецификации” иллюстрирует некоторые из возникающих проблем, а раздел 2.2 “Использование прагм в IDL” предлагает их решение с использованием стандартного средства языка IDL - директив pragma или просто прагм. Здесь же проводится классификация прагм, введенных для Protel-2, а также даются примеры их применения. Часть 3 “Компилятор с языка программирования в IDL” предлагает решение той же проблемы с помощью некоторого автоматического средства - компилятора с используемого языка программирования в IDL. В разделе 3.1 “Автоматическое порождение IDL спецификаций с прагмами” излагаются идеи, как должен работать такой компилятор, и обсуждаются проблемы отображения языка программирования (на примере Protel-2) в IDL. Приводимый список неоднозначностей покрывает, практически, все разумные случаи для Protel-2. Наконец, раздел 3.2 “Долой прагмы!” предлагает следующий шаг в решении проблемы, когда мы совмещаем компиляторы с IDL в язык программирования и с языка программирования в IDL в одном. Это дает возможность избежать промежуточного этапа получения IDL спецификации, если мы уже имеем некоторую реализацию “открываемого” объектного контекста. Последнее, в частности, справедливо для унаследованных систем.

2. Как получить IDL спецификацию

Прежде чем автоматизировать решение поставленной задачи, мы попытаемся решить ее вручную. То есть так, как это сделал бы человек, столкнувшийся с необходимостью “открыть” свою систему, но не имеющий под рукой никаких подручных средств.

*2.1 Ручное получение **IDL** спецификации*

Наиболее простым способом получения IDL спецификации для некоторого существующего кода с точки зрения организации, является генерация его человеком. Для разработчика системы, очевидно, это самый трудоемкий и полный ошибок путь. Рассмотрим пример.

Пусть в унаследованной системе объявлены следующие типы (приведен код на языке Protel-2):

```
TYPE ProtelInfo STRUCT
      importance $longint,
      valid      bool
ENDSTRUCT;
TYPE legacy_class_ptr PTR TO ANY legacy_class;
TYPE legacy_class CLASS REFINES $OBJECT
            field ProtelInfo
      OPERATIONS
            field_set METHOD (UPDATES; REF x ProtelInfo),
            field_get METHOD (REF) RETURNS ProtelInfo
ENDCLASS legacy_class;
```

Как уже было сказано, задача выбора объектных контекстов подлежащих “открытию”, является нетривиальной. Не все объекты унаследованной системы можно экспортировать. Например, во многих системах, особенно работающих в реальном времени, существуют группы тесно связанных между собой объектов. Вынесение одного объекта из группы нарушило бы связи (например, временные) между ними и потребовало бы изменений в архитектуре системы. Невозможность экспорта объекта может следовать также из способа его реализации на языке программирования (например, когда все поля и методы объекта объявлены доступными только ему и/или его наследникам). Здесь и далее, мы считаем, что некоторый “открываемый” объектный контекст уже выбран.

Соответствующая примеру IDL спецификация может выглядеть так:

```
struct IDLInfo {
      long        importance;
      boolean     valid;
};
interface legacy {
      void        field_set(in IDLInfo info);
      IDLInfo     field_get();
};
```

Написав спецификацию и использовав стандартный IDL компилятор для порождения необходимого кода на целевом языке программирования, разработчик столкнется с некоторыми проблемами. А именно:

1. Типы **IDLInfo** и **legacy** будут объявлены как в IDL спецификации (а, следовательно, и в порожденном коде), так и в унаследованной системе. Это одни и те же типы. Поэтому человек вынужден будет вручную удалить дублирующиеся объявления.
2. Логично оставить объявления существовавших в унаследованной системе типов, а в порожденном коде вставить ссылки на них. Очевидно, это придется делать вручную. В данном случае в список `uses` порожденного IDL компилятором модуля нужно вставить имена модулей с объявлениями типов.
3. Наконец, имена сгенерированных из IDL типов будут другими, отличными от существующих. Поэтому предстоит большая работа по ручному исправлению этих имен во всех процедурах и типах порождаемых IDL компилятором.

Для программиста может оказаться удобным, объявить тот или иной тип на IDL под другим именем. В этом случае ему предстоит переименовать сгенерированные IDL компилятором имена типов (или объявить синонимы) как в местах объявления, так и в местах использования.

Пример также демонстрирует одну из неоднозначностей, которая возникает при обратном отображении языка программирования в IDL. В данном случае доступ к атрибуту `field` класса `legacy_class` на Protel-2 осуществляется двумя методами - `field_set` и `field_get`, видимыми пользователю. В IDL спецификации описаны такие же методы. Но, так как с точки зрения IDL, атрибут есть просто пара методов, то можно было бы в IDL спецификации объявить один атрибут `field`, и эффект был бы тем же самым[4].

Конечно, все перечисленные трудности решаются ручным редактированием генерируемого IDL компилятором кода. Но нам хотелось бы избавить человека от этой рутинной работы, поручив ее машине. Тем более что исправлять приходиться генерируемый код, то есть все сделанные правки будут потеряны, если код нужно будет перегенерировать. Логично было бы редактировать только исходный текст, полагаясь в остальном на автоматические средства. Для этого мы делаем первый шаг.

2.2 Использование прагм в IDL

Недостатки, связанные с использованием IDL для существующего кода, могут быть устранены. Язык IDL имеет встроенное средство для указания системно-зависимой информации компилятору - прагмы [2]. В директивах pragma, программист мог бы ссылаться на определения существующих в унаследованной системе типов, производить замену имен, использовать другие, уже скомпилированные, IDL спецификации. Прагмы имеют смысл только для того компилятора, который их умеет интерпретировать. Стандартный компилятор с IDL просто игнорирует неизвестные ему прагмы.

В общем случае прагма в IDL представляется одной строкой текста, начинающейся с ключевого слова **#pragma**. Для решения поставленной задачи -

---

[4] Не считая того, что пришлось бы в генерируемом IDL компилятором коде заменять имена методов, порожденных по умолчанию, на существующие имена.

использования существующего в унаследованной системе кода, мы ввели следующие классы прагм [6] (далее приводятся классы прагм для конкретного языка программирования - Protel-2, однако аналогичным образом можно было бы расширить IDL компилятор с любого другого объектно-ориентированного языка):

- прагмы, предназначенные для включения текста на IDL или Protel-2 в IDL спецификацию или в порождаемые Protel-2 модули соответственно. Прагмы этого класса называются includeProtel, includes, inline, useProtel, uses;
- прагмы, позволяющие изменять имена, генерируемые IDL компилятором по умолчанию: prefix, replaceid, filename;
- прагмы, указывающие, что IDL спецификация для данного интерфейса будет использоваться только как клиентская или только как серверная: noclient, noserver;
- наконец, прагмы, дающие возможность подменить порождаемый IDL компилятором тип, класс или атрибут класса на существующие (например, в унаследованной системе) объявления: maptype, mapinterface, mapattribute.

Наиболее важным с точки зрения использования написанного кода является последний класс прагм. Все входящие в него прагмы действуют аналогичным образом. Вставленные в IDL спецификацию, они заставляют IDL компилятор не генерировать тип, класс или атрибут по умолчанию, а использовать вместо него указанный Protel-2 тип, класс или атрибут соответственно. Для нашего примера можно было бы написать такую IDL спецификацию:

```
#pragma Protel useProtel legmod
#pragma Protel maptype IDLInfo as ProtelInfo
struct IDLInfo {
        long            importance;
        boolean         valid;
};
#pragma Protel mapinterface legacy as legacy_class with legacy_class_ptr
interface legacy {
        #pragma Protel mapattribute field asmethods field_get field_set
        attribute IDLInfo field;
};
```

Как можно видеть, были добавлены четыре прагмы. Все прагмы имеют общий синтаксис:

- ключевое слово #pragma;
- слово Protel (указывающее, что прагма относится к компилятору с IDL в Protel-2);
- тип прагмы - одно слово (в примере это useProtel, maptype, и так далее);

- и, наконец, параметры прагмы - свои для каждого типа.

Первая из прагм (**useProtel**) заставляет IDL компилятор добавить указанные имена в список `uses` генерируемого модуля. Тогда, объявления из указанных модулей становятся доступными в Protel[5].

Вторая прагма (**maptype**) говорит компилятору, что тип **IDLInfo** объявлен в Protel как `ProtelInfo`. Таким образом, компилятор не должен генерировать определение типа, а должен лишь породить необходимые для работы с ним процедуры. При этом во всех вспомогательных типах и процедурах будет стоять тип, указанный в прагме (а не порождаемый компилятором по умолчанию). Тип `ProtelInfo` должен быть объявлен или быть видим (по правилам видимости Protel-2) в модуле `legmod`, для этого и производилось подключение последнего. Структура определяемого типа, конечно, должна соответствовать существующему. Прагму **maptype** можно применять только для типов, которые не являются классами.

Для отображения IDL классов на существующие Protel-2 классы применяется прагма **mapinterface**. Прагма имеет три параметра. Во-первых, имя IDL класса. Во-вторых, имя Protel-2 класса, который будет соответствовать классу в IDL. И, в-третьих, имя типа - указателя на объявленный класс. Очень часто вместе с объектами данного класса используются указатели на класс (а в генерируемом IDL компилятором коде используются, в основном, указатели), и нередко соответствующие типы объявлены в унаследованной системе[6]. Поэтому в третьем параметре прагмы программист должен сослаться на существующий или объявленный тип указателя. Если для некоторого интерфейса в IDL указана прагма **mapinterface**, то это означает, что в порождаемом IDL компилятором модуле не будет сгенерирован соответствующий класс, а во всех вспомогательных типах и процедурах будет использоваться указанный в прагме Protel-2 класс и тип указателя.

На использование прагмы **mapinterface** накладываются определенные ограничения. Пусть, например, объявлены два IDL интерфейса **A** и **B**, причем **B** наследует из **A**. Прагма **mapinterface** применена к интерфейсу **B**. Если интерфейс **A** отображается тем или иным способом в Protel-2 класс `Ap`, то интерфейс **B** должен отображаться в Protel-2 класс `Bp`, который наследует из `Ap`. Применение прагмы **mapinterface** для интерфейса IDL, который не наследует из другого интерфейса, влечет за собой требование обязательного наследования соответствующим Protel-2 классом из `CORBA_Object` (базового класса для всех объектов, объявленных в IDL).

Последняя прагма (**mapattribute**) указывает IDL компилятору, что для доступа к атрибуту `field` будут использоваться методы `field_get` и `field_set`. Применение этой прагмы обосновано тогда, когда в существующем коде уже есть методы доступа к атрибуту и их нужно использовать. Прагма **mapattribute** может быть применена и без прагмы **mapinterface**. Это приведет к использованию в порождаемом коде других имен и, даже, других атрибутов, вместо стандартным образом генерируемых методов.

---

[5] В данном случае, в порождаемом компиляторе модуле будут доступны декларации из модуля `legmod`.

[6] Если же тип указатель не объявлен, то можно воспользоваться прагмой **inline**. Она позволяет вставить в генерируемый компилятором файл любой фрагмент кода на Protel-2.

Кроме приведенных прагм были введены и другие, облегчающие использование унаследованного кода. Перечислим их с краткими комментариями:

- includeProtel - включает файл на Protel-2 в интерфейсную часть генерируемого модуля. Прагма удобна для вставки объявляемых пользователем типов и/или процедур;
- includes - указывает на включение в IDL спецификацию файла на IDL. Эта прагма применяется для выбора одной из возможных интерпретаций директивы #include IDL и распространяется на указанный в ней IDL файл;
- uses - другая интерпретация #include в IDL. В прагме, как аргумент, указывается имя Protel-2 файла со скомпилированной IDL спецификацией. При этом во время компиляции одной IDL спецификации может использоваться другая, но исходный текст последней не нужен. Такая схема работы соответствует принятой в Protel для разработки программ;
- inline - позволяет вставить небольшой фрагмент кода (в пределах одной строки) на Protel-2 в интерфейсную часть модуля генерируемого IDL компилятором;
- prefix - позволяет изменить префикс для вложенных генерируемых имен в Protel-2;
- replaceid - заменяет IDL имя указанным Protel-2 именем. В отличие от прагм maptype или mapinterface происходит только подстановка имени. Все типы, которые IDL компилятор должен был породить - генерируются;
- filename - указывает имена генерируемых IDL компилятором файлов;
- noclient и noserver - говорят IDL компилятору, что спецификация интерфейса будет использоваться только как клиентская или только как серверная. Поэтому компилятор может не генерировать некоторый код.

Расширив IDL компилятор пониманием таких прагм, мы можем облегчить жизнь программисту, когда ему нужно использовать существующий код. Те проблемы, которые были перечислены в разделе 2.1 “Ручное получение IDL спецификации” фактически решаются на уровне составления IDL спецификации. Преимущества и недостатки такого похода очевидны - не нужна ручная правка генерируемого компилятором кода, с одной стороны. С другой стороны, IDL спецификация должна быть расширена прагмами, что требует некоторой работы от программиста.

Следующий шаг в решении поставленной задачи состоит в том, чтобы избавить человека от ручного создания IDL спецификаций.

3. Компилятор с языка программирования в IDL

Рассмотренный способ генерации IDL спецификаций очень трудоемок. Ту же задачу можно было бы решить с помощью некоторого автоматического средства, способного по заданному описанию интерфейсов на некотором языке программирования, сгенерировать соответствующую IDL спецификацию. Логично такое средство назвать компилятором с языка программирования в язык IDL (далее просто компилятор).

3.1 Автоматическое порождение IDL спецификаций с прагмами

Входом компилятора служит набор файлов с текстом на используемом языке программирования, и содержащих описание “открываемого” объектного контекста. То есть, набор деклараций констант, типов, классов. Все остальные конструкции во входных файлах просто игнорируются, так как не могут найти своего отражения в IDL спецификации. Результатом работы компилятора является спецификация того же самого объектного контекста, но уже на IDL, расширенном прагмами из раздела 2.2 “Использование прагм в IDL”. Использование прагм необходимо, так как мы должны в IDL спецификации ссылаться на существующий код унаследованной системы.

Общая идея компилятора проста. Конструкции, объявленной на языке программирования, ставится в соответствие конструкция на IDL: структуре - структура, массиву - массив и так далее.

Для того чтобы компилятор порождал код верный не только синтаксически, но и семантически, он должен разрешать возникающие неоднозначности. Дело в том, что по заданной спецификации на некотором языке программирования можно породить, вообще говоря, несколько соответствующих IDL спецификаций. И объясняется это не только более богатыми выразительными способностями практически любого языка программирования по сравнению с IDL, но и неоднозначностями, возникающими при обратном отображении. Например, в IDL тип union требует указания дискриминанта, на основе которого делаются выводы о типе содержимого. В языке Protel-2 тип “`struct with ovly`” (структура с оверлеем - аналог union) не содержит дискриминанта. Поэтому перед компилятором встает проблема его выбора. Ясно - что бы ни выбрал компилятор, его вывод может оказаться семантически неверным и только в силах программиста направить его на верный путь.

Другой аспект, важный для разработчика системы - прагматический. Предложенный компилятором вариант при разрешении некоторой неоднозначности, может быть правильным как синтаксически, так и семантически, но не удовлетворять человека по условиям производительности или каким-либо другим.

Далее рассматриваются наиболее важные и типичные проблемы обратного отображения - языка программирования в IDL на примере Protel-2.

3.1.1 Проблема выбора дискриминанта для IDL типа union

Одна из часто используемых конструкций в языках программирования - это тип union или аналогичный ему. В языке Protel-2, например, ему соответствует структура с оверлеем:

```
TYPE U STRUCT
      OVLY {0 TO 2}
      {0}: x     $longint
      {1}: y     {0 TO 255}
      {2}: z     bool
      ENDOVLY
ENDSTRUCT;
```

В IDL тоже есть такой тип, но в отличие от некоторых языков программирования (типа C или Protel), он содержит дискриминант - тег, на основании значения которого можно сделать вывод о типе содержимого. Например, приведенный на Protel тип мог бы быть описан на IDL следующим образом:

```
union U switch (short) {
      case 0: long        x;
      case 1: char        y;
      case 2: boolean     z;
};
```

Хотя описания кажутся на первый взгляд идентичными, их семантика различна. В Protel указывается только тип дискриминанта и программист не имеет возможности использовать его значение в своей программе - этого значения просто не существует. Таким образом, при отображении Protel в IDL возникает проблема выбора дискриминанта. Компилятор должен выбрать переменную или процедуру, которая будет вычислять значение дискриминанта. В случае, когда вычисление невозможно, делается вывод, что тип не подлежит экспорту.

Приведенный пример структуры с оверлеем на Protel, иллюстрирует еще одну особенность языка. Оверлей может быть объявлен только внутри структуры, но не на верхнем уровне, к тому же оверлеев может быть несколько. В этом случае все они суть анонимные типы. В IDL структуре с несколькими оверлеями должна быть поставлена в соответствие структура с несколькими полями типа `union`. Но так как в IDL нельзя объявлять одни типы при объявлении других (например, `union` внутри `struct`), то необходимо создать вспомогательные типы. Конечно, если имена этих типов не использовались на Protel, то в IDL для них можно сгенерировать некоторые имена по умолчанию. С другой стороны, можно предоставить пользователю возможность указать, какие конкретно имена использовать для этих типов.

3.1.2 Проблема указателей в Protel-2

В языке Protel-2, как и во многих других, программист может пользоваться указателями. С точки зрения реализации, в Protel-2 указатель представляет собой адрес памяти, содержащей значение указываемого типа. Это справедливо как для указателей на базовые или конструируемые типы, так и для указателей на классы. Очевидно, значение самого указателя имеет смысл только в том контексте, в том адресном пространстве, в котором он создан.

В следующем примере параметр метода объявлен как указатель:

```
m METHOD (REF; p PTR TO T)
```

Язык IDL не имеет такой конструкции как указатель, поэтому возникает проблема отображения объектов и типов унаследованной системы, содержащих указатели.

Общий подход к решению проблемы состоит в замене указателя на какой-либо тип интерфейса IDL. Для приведенного примера компилятор с Protel-2 в IDL может сгенерировать следующий вспомогательный интерфейс и объявление метода:

```
interface T_wrapper {
        attribute T value;
};
…
void m (in T_wrapper p);
```

В вызываемый метод передается объект типа `T_wrapper` (и тем самым объектная ссылка) и каждое обращение к указываемому значению будет приводить к удаленному запросу. Атрибут `value` (фактически пара методов) предоставляет доступ к значению переменной. Тип `T`, конечно, должен быть экспортирован. Если `T` - это класс, то компилятору не нужно генерировать вспомогательный интерфейс, а соответствующий метод в IDL может быть объявлен следующим образом:

```
void m (in T p);
```

Проблема с указателями для типов, объявленных как указатели, а также для составных типов, внутри которых содержатся указатели, решается аналогично. Для каждого объявления указателя порождается вспомогательный интерфейс (таким образом, интерфейсы могут быть вложенными). Выбор имен таких интерфейсов можно возложить как на компилятор, который будет порождать имена по умолчанию, так и на пользователя.

В некоторых случаях для вспомогательного интерфейса необходим также метод явного создания объектов типа. Обычно метод создания объектов является методом класса. Проблема экспорта классовых методов рассмотрена в разделе 3.1.3 “Классовые методы в Protel-2”.

Рассмотренный способ решения проблемы с указателями хотя и корректен, но в некоторых случаях может оказаться неэффективным. Это объясняется тем, что для доступа к значениям указываемого типа приходиться использовать ORB. Возможны случаи, когда компилятор и/или разработчик могут оптимизировать генерируемый код. То есть указатели могут быть устранены без ущерба для семантики путем замены их в IDL указываемым типом. (Например, когда полем структуры является подструктура, представленная указателем.) В общем случае, вывод о возможности устранения указателя может сделать только разработчик системы.

3.1.3 Классовые методы в Protel-2

Многие объектно-ориентированные языки программирования позволяют описывать в классах так называемые “классовые” методы - методы, которые могут быть вызваны не на экземпляре класса, а на самом классе. К таким языкам относится, в частности, и Protel-2. IDL не поддерживает классовые методы и в IDL спецификации они могут быть объявлены только обычным образом. Для вызова любого метода (при стандартном отображении в используемый язык [6]) нужно иметь экземпляр объекта, и тем самым объектную ссылку. Если класс содержит классовые методы, но ни одного объекта этого класса еще не создано, то получается, что невозможно вызвать метод класса.

Для унаследованных систем можно предложить следующее решение этой проблемы. Компилятор может создать в IDL спецификации вспомогательный класс, содержащий только классовые методы исходного. В реализационную часть генерируемого IDL компилятором модуля должна быть добавлена процедура создания и инициализации ровно одного объекта вспомогательного класса. Тем самым на ORB будет возложена работа по порождению объектной ссылки. А пользователь сможет вызывать классовые методы на вспомогательном классе, даже если объектов исходного класса еще нет.

3.1.4 Updates параметры процедур

Язык Protel-2, как впрочем, и многие другие, предлагает два способа передачи параметров в процедуры: по значению и по ссылке. При передаче параметра по ссылке нужно указать - меняется ли в процедуре этот параметр или нет. В Protel-2 это делается с помощью ключевых слов `updates` и `ref` соответственно. Если в унаследованной системе в какой-либо процедуре параметр передается как `updates`, то возникает вопрос: какой тип передачи должен соответствовать ему в IDL?

Как известно IDL поддерживает три типа передачи параметров процедур: **in**, **out** и **inout**. Можно привести примеры, когда последние два типа могут быть использованы для Protel `updates` параметра.

Логично было бы предположить, что для всех `updates` параметров можно использовать **inout** тип передачи в IDL. Это наиболее типичный случай, но следующий пример показывает, что иногда это опасно. В Protel-2 есть такой тип, как дескриптор. Фактически, это массив с переменной границей. Дескриптор содержит указатель на данные и число элементов. Пусть есть некоторая процедура с `updates` параметром типа дескриптор. Для этого параметра в IDL спецификации указан тип передачи **inout**. Процедура не использует входное значение параметра, а просто переписывает его выходным значением. Программист, зная это, передает в процедуру не проинициализированный дескриптор. Тогда, при упаковке параметров для передачи по сети ORB будет пытаться разыменовать указатель в дескрипторе и возникнет исключительная ситуация.

Таким образом, компилятор должен производить анализ исходного текста (а значит, он должен иметь доступ к нему) на предмет использования значений параметров типа `updates` внутри тел процедур. Если нет уверенности ни в одном из случаев, то компилятор должен завершать работу выдав сообщение об ошибке. Применение варианта по умолчанию (всегда генерировать **inout**) возможно, но довольно опасно и тогда пользователь должен проверять соответствие сгенерированной IDL спецификации существующему коду.

Другое решение проблемы использования как `ref` так и `updates` параметров в исходной программе состоит в замене декларации вида:

```
m1 METHOD (REF; REF P T1, UPDATES Q T2)
```

на

```
m1 METHOD (REF; P PTR TO VAL T1, Q PTR TO T2)
```

При этом во всех вызовах перед фактическими параметрами должно быть поставлено ключевое слово `PTR`. Тем самым, задача сводится к уже рассмотренной выше задаче отображения указателей (см. раздел 3.1.2 ”Проблема указателей в Protel-2”). Такое решение является более надежным (хотя и менее эффективным) и может быть использовано компилятором в автоматическом режиме. Понятно, что выбрать наиболее подходящее решение может только человек.

3.1.5 Проблема атрибутов в Protel-2

Типичной ситуацией является тот случай, когда класс содержит не только общедоступные методы, но и общедоступные атрибуты. Для унаследованной системы стоит задача породить соответствующую IDL спецификацию, то есть экспортировать класс, в том числе и эти атрибуты. Для этого можно воспользоваться прагмой **mapattribute** в Protel-2 (см. 2.2 “Использование прагм в IDL”). Но тогда IDL компилятор не сможет сгенерировать соответствующий спецификации код на Protel. Дело в том, что для каждого интерфейса IDL генерируется stub - некоторый посредник между клиентом этого класса и реализацией [2]. Stub - это класс, который наследует от абстрактного класса, соответствующего объявленному в IDL, и переопределяет все его методы для обработки удаленных вызовов [6]. Однако язык Protel-2 не поддерживает переопределение атрибутов в классах и тем самым IDL компилятор не сможет создать stub.

Решение проблемы состоит в обязательной генерации компилятором с Protel-2 в IDL прагмы **noclient**, отменяющей порождение клиентского кода. То есть, для тех классов унаследованной системы, которые содержат общедоступные атрибуты, возможно использование только в качестве сервера.

3.1.6 Неоднозначности отображения некоторых типов

Для некоторых типов языка программирования можно указать несколько способов отображения в IDL. Выбор того или иного способа может диктоваться удобством работы, эффективностью, другими соображениями. Мы приведем несколько примеров наиболее типичных неоднозначностей, возникающих при применении языка Protel-2.

На Protel программист может объявить тип-интервал со значениями из указанного целочисленного диапазона. В IDL такому типу можно поставить в соответствие любой целочисленный тип. Достаточно лишь, чтобы все значения Protel типа были представимы. Таким образом, для интервала `{0 TO 2}` в Protel можно использовать типы **octet**, **char**, **short**, **unsigned short** и так далее в IDL. Выбор остается за программистом.

Другой пример связан с типом `table` (массив) Protel. В IDL логично представить его также массивом, но возможен и другой вариант. Если мы хотим передавать этот массив не по значению (копированием всех элементов со стороны вызывающего объекта на сторону вызываемого), а по ссылке, то в IDL ему можно поставить в соответствие интерфейс[7]. Тогда при вызове метода будет передаваться объектная ссылка. А при доступе к элементам массива будут возникать удаленные запросы. Такой подход целесообразен в случае больших массивов, когда копирование слишком дорого, а вызываемый объект делает доступ к небольшому числу элементов. Вообще говоря, практически любой тип языка программирования можно отобразить на наиболее подходящий ему в IDL или на интерфейс.

Наконец, последний пример - это неоднозначное отображения классов используемого языка программирования в IDL. Типичный вариант состоит в представлении классов интерфейсами на IDL. Как альтернативу, можно предложить использовать для этой цели структуры IDL. В случае, когда класс на используемом языке содержит только атрибуты и ни одного метода, такой способ отображения позволяет передавать объекты не по ссылке, а по значению, что может оказаться полезным.

3.1.7 Диалог с компилятором

Рассмотренный список проблем дает понять, что не все объектные контексты, которые подаются на вход компилятору с языка программирования в IDL, могут быть открыты. При работе компилятора могут возникать как предупреждения, так и ошибки (см., например, раздел 3.1.1 “Проблема выбора дискриминанта для IDL типа union”). В таких случаях человек должен пересмотреть исходную спецификацию.

Разрешение неоднозначностей компилятором требует глубокого анализа исходного текста. А способ разрешения может основываться на выборе варианта по умолчанию. Однако удобно, а в некоторых случаях необходимо, давать пользователю возможность самому принять решение. Для этого можно использовать два подхода.

Первый состоит в интерактивности компилятора, когда необходимый диалог между программой и человеком происходит во время компиляции (и может существенно опираться на графический интерфейс пользователя). Компилятор мог бы поддерживать разные уровни интерактивности. От полностью автоматической работы, при которой все решения принимаются без вмешательства человека, до сугубо интерактивной, когда любая проблема (разрешение неоднозначности, выбор имени для вспомогательного типа и так далее) вызывает останов процесса компиляции и вмешательство человека. В любом случае, программист после использования компилятора должен проверить полученную IDL спецификацию на соответствие своему объектному контексту. И сделано это может быть только вручную.

Второй подход состоит в реализации пакетного режима работы. Пользователь должен каким-либо способом специфицировать исходный объектный контекст, а также свои решения по разрешению неоднозначностей.

---

[7] В нем должна быть объявлена операция доступа к элементам массива.

Эта проблема должна решаться по-разному для разных языков. Для Protel-2 можно было бы поместить всю информацию в дополнительный файл (написанный на Protel). В него же программист мог бы добавить какой-то дополнительный код.

Каждый из предложенных подходов имеет свои преимущества и недостатки. Однако в силу того факта, что интерактивные компиляторы трудны в использовании и обычно не приживаются, мы остановили свой выбор на втором подходе.

3.2 Долой прагмы!

После того, как процесс получения IDL спецификации автоматизирован, можно пытаться его оптимизировать. Давайте рассмотрим типичную последовательность действий человека, использующего стандарт CORBA для унаследованной системы.

На первом шаге после выделения объектного контекста в унаследованной системе, разработчик применяет компилятор с языка программирования в IDL и получает IDL спецификацию. Затем, эта спецификация должна быть скомпилирована IDL компилятором. Результатом будут модули на целевом языке программирования. Если целевой язык и окружение отлично от присущих унаследованной системе, такая цепочка действий является единственно возможной. В том случае, когда нужно породить код на языке реализации унаследованной системы, последовательность действий явно избыточна. Наше предложение состоит в совмещении двух компиляторов - с языка программирования в IDL и с IDL в язык программирования, в одном. Конечно, должна оставаться возможность воспользоваться только одним из них.

При таком подходе отпадает необходимость пользоваться прагмами (раздел 2.2 “Использование прагм в IDL”). Действительно, для компиляторов с IDL в другие языки прагмы окажутся бесполезными и будут проигнорированы, для используемого языка необходимый код будет порожден напрямую (например, из унаследованного кода на Protel-2 - все необходимые типы, процедуры, классы и их реализации - на Protel-2).

Итак, на входе совмещенный компилятор будет получать спецификацию объектного контекста на используемом языке программирования, а на выходе выдавать IDL спецификацию[8] этого контекста и те модули, которые должен порождать стандартный IDL компилятор. Информация, необходимая компилятору для представления исходного контекста, разрешения неоднозначностей и так далее, может подаваться как в диалоге (в интерактивной форме), так и в виде дополнительных файлов (см. раздел 3.1.7 “Диалог с компилятором”). Для Protel-2 файлы могут быть, например, написаны на Protel (как интерфейсные секции) и содержать псевдокомментарии со всей необходимой информацией. Модуль, порождаемый IDL компилятором [6] может расширять эти файлы, и все они целиком будут составлять единый модуль. Тем самым программисту будет удобно использовать как свой код, относящийся к спецификации объектного контекста, так и код, порождаемый IDL компилятором

---

[8] Она необходима для тех частей системы, которые будут вынесены из унаследованного окружения, а также для внешних по отношению к системе объектов.

- нужно вставить имя модуля в списки `uses` использующих модулей. При необходимости расширения реализации, в генерируемый модуль может быть добавлена реализационная секция.

4. Заключение

В статье рассматриваются задачи, возникающие при применении стандарта CORBA к унаследованным системам - автоматическое получения IDL спецификации по существующему коду и использование последнего в реализации новой, модернизированной системы. Появляющиеся при этом проблемы могут быть решены расширением языка OMG IDL и созданием "обратного" компилятора - с языка программирования в IDL. Расширение использует стандартное средство IDL - прагмы. Кроме того, как оптимизация, предложен компилятор, совмещающий функции стандартного IDL компилятора и "обратного" компилятора.

В настоящее время проделана работа по расширению IDL набором прагм для поддержки унаследованного кода. Кроме этого, реализован компилятор с расширенного IDL. "Обратный" компилятор и связанные с ним проблемы находятся на стадии разработки.

Благодарности

Выражаем свою искреннюю признательность Виктору Петровичу Иванникову, который прочитал рукопись и сделал ценные замечания. Мы благодарны ему также за настойчивость без которой статья вряд ли увидела бы свет.

Список литературы

1. The Object Management Architecture Guide. Object Management Group. 1990. Revised 1995.
2. The Common Object Request Broker: Architecture and Specification. Revision 2.0. Object Management Group. 1995.
3. J. Siegel. CORBA - Fundamentals and Programming. John Wiley & Sons. 1996.
4. S. Danforth, I. R. Forman. Reflections on Metaclasses Programming in SOM. OOPSLA'94 Conference Proceedings, 1994.
5. I. R. Forman, S. Danforth, and H. Madduri. Composition of Before/After Metaclasses in SOM. OOPSLA'94 Conference Proceedings, 1994.
6. A. V. Klimov, M. R. Kovtun, and S. G. Manzheley. Protel-2 ORB System Guide. Technical Documentation. Institute for System Programming, Nortel. 1996.
7. M. Turnbull. Protel-2 Reference Manual. For Compiler Version P2BRxx. Bell Northern Research. 1995.
8. AES/Distributed Computing Remote Procedure Call. Revision B. Open Software Foundation. 1995.