IDL C++ меппинг. CORBA и возможная альтернатива

К. В. Дышлевой

Эта статья затрагивает аспект организации передачи данных в рамках технологии CORBA. Рассматривается предусматриваемое стандартом отображение (меппинг) языка IDL в C++ в той его части, которая касается передачи параметров и результатов методов объектов. Также описывается предлагаемый альтернативный меппинг, рассчитанный на преодоление описанных здесь проблем CORBA меппинга.

Введение

В процессе разработки и отладки прототипа C++ ORB, осуществленных в ИСП РАН в течение 1996 г., был получен немалый опыт в создании и использовании технологии организации взаимодействия объектов в неоднородной распределенной среде. Многие достоинства и недостатки технологии CORBA оказалось возможным оценить как извне, т.е. в качестве пользователя предлагаемых этой технологией механизмов, так и изнутри, т.е. в качестве разработчика реализации этих механизмов. При этом, стали очевидными и те многочисленные проблемы, которые связаны с зафиксированным в стандарте CORBA меппингом IDL/C++. Пожалуй, основную массу этих проблем можно охарактеризовать так: от пользователя требуется довольно много чего знать и уметь, зафиксирована масса различных ограничений и требований, которым он должен следовать. Проверка со стороны системы (ORB) правильности действий пользователя не предусматривается. Однако, за ошибки, связанные с невыполнением *всех* требований меппинга, пользователю зачастую приходится платить довольно существенными потерями времени на кропотливую отладку программ, и, конечно же тем, что всегда остается возможность проявления новых подобных же ошибок, что неблагоприятно влияет на надежность программ.

Конечно же, возникает вопрос о возможности внесения некоторых изменений в меппинг C++, которые позволили бы решить многие проблемы пользователя. Неплохим источником идей здесь может служить техника оболочек ([3]) и интерфейсных объектов ([2]), используемая в еще одном проекте института, связанном с “метаобъектным контролем”. Механизм оболочек и интерфейсных объектов предназначен для организации работы объектов неоднородной распределенной среды таким образом, что любой пользователь объекта или группы объектов знает о том, с чем работает лишь то, что ему нужно. Этот механизм позволяет скрыть от пользователя “неинтересные” для него взаимосвязи между объектами, а также контролирует работу с доступными пользователю объектами. Использование аналогичного механизма на более низком уровне, т.е. на уровне работы с параметрами методов объектов может позволить пользователю работать с данными так, чтобы многое выполнялось за него системой программирования и, таким образом, упростить и процесс создания программ, и их отладку.

Итак, начнем с рассмотрения тех самых проблем CORBA IDL/C++ меппинга, которые и явились причиной появления предложений, более подробно описанных во втором разделе этой статьи.

1. Проблемы CORBA отображения из IDL в C++.

1.1. Одной из важнейших абстракций CORBA ([1]) является понятие объектной ссылки. Объекты, реализующие интерфейсы, описываемые на языке IDL (Interface Definition Language) ([1], гл. 3), могут указываться с помощью особого вида данных - объектных ссылок. При описании методов интерфейсов в качестве типа параметра метода может быть указано имя некоторого интерфейса, скажем A. Это обозначает, что через соответствующий параметр можно передавать указатель на объект, реализующий интерфейс A.

Для работы с такими данными, т.е. указателями на реализации, для каждого такого интерфейса A в CORBA меппинге ([1, 6-9]) из IDL в C++ ([4,5]) (далее будем его называть просто CORBA меппинг) предусмотрено два типа ссылок - A_ptr и A_var, совершенно различных по схеме работы с памятью (см. [1] 16.3.1.).

По сути, A_ptr является просто указателем, а A_var - специальный объект, имеющий конструктор и деструктор, владеющий памятью, на которую указывает. То есть, уничтожение объекта A_var обязательно влечет за собой и уничтожение памяти, на которую он ссылается. При этом, значение переменной любого из этих типов может быть присвоено переменной другого типа, определено преобразование типа A_var в A_ptr. Во всех этих случаях дублирование памяти не производится. Не предусмотрено никаких средств контроля подобных манипуляций с данными этих типов. Это и служит весьма распространенным источником проблем при работе с объектными ссылками. Так, например, следующая последовательность присваиваний является некорректной:

```
{
  A_var var1 = ...;     // некоторое заполнение переменной
  A_ptr ptr  = var1;
  A_var var2 = ptr;
}
```

Здесь ни в одном из трех присваиваний память не дублируется, все три переменные var1, var2, и ptr указывают на одни и те же данные. При выходе из блока, конечно же, возникнет ошибка работы с динамической памятью: при вызове деструкторов для var1 и var2 будет дважды уничтожаться одна и та же память, на которую они ссылаются.

Хоть стандарт CORBA и предупреждает о необходимости программисту (*самостоятельно*) избегать подобных ситуаций, такие ошибки, все равно, очень часто возникают даже при написании не очень сложных приложений. Ведь подобная схема требует при программировании постоянно держать в голове существующие взаимосвязи между *всеми* объектными ссылками в программе. Кроме того, всегда нужно помнить и о схеме работы с параметрами в той ее части, которая касается объектных ссылок (см. далее *1.3.*).

1.2. Особое внимание при работе с объектными ссылками CORBA меппинг требует от программиста еще и в связи с использованием операций duplicate и release ([1] 16.3.3.). Первая из них явным образом копирует ссылку, вторая - ее уничтожает.

Так, с помощью операции duplicate пример из *1.1*. можно сделать корректным:

```
{
  A_var var1 = ...;
  A_ptr ptr  = var1;
  A_var var2 = ptr->duplicate();
}
```

Однако, очевидно, что использование таких операций также заставляет программиста постоянно думать о взаимосвязи всех объектных ссылок. Любой лишний вызов duplicate или не поставленный в нужном месте вызов release может привести к тому, что память из под данных объектной ссылки не будет уничтожена. Подобная ошибка, например в циклически работающем мониторе, может вызвать постоянное увеличение размера используемой динамической памяти (далеко не сразу обнаруживаемая проблема). Любой же лишний вызов release или не произведенный вызов duplicate довольно быстро приводят к ошибке работы с памятью, как это было в примере из *1.1*.

Тип данных	**In**	**Inout**	**Out**	**Return**
simple	simple	simple &	simple &	simple
object ref. ptr	objref_ptr	objref_ptr&	objref_ptr&	objref_ptr
struct, fixed	const struct &	struct &	struct &	struct
struct, variable	const struct &	struct &	struct *&	struct *
union, fixed	const union &	union &	union &	union
union, variable	const union &	union &	union *&	union *
string	const char *	char *&	char *&	char *
sequence	const sequence&	sequence &	sequence *&	sequence *
array, fixed	const array	array	array	array slice*
array, variable	const array	array	array slice*&	array slice*
any	const any &	any &	any *&	any *

Рис. 1. Меппинг типов параметров и результатов в CORBA.

1.3. Пожалуй, самую сложную, трудно понимаемую и запоминаемую часть меппинга представляет собой набор правил передачи параметров и результатов. Во первых, для каждого типа данных языка IDL специфицированы отдельно типы C++ параметров всех четырех видов IN (входные), OUT (выходные), INOUT и результата ([1], таблица 24 на стр. 16-46 - 16-47). На

рис. 1 приводится таблица с этими соглашениями. Общую схему передачи параметров здесь можно выделить лишь для простых типов данных (short, long, unsigned short, unsigned long, float, double, boolean, char, octet, enum), которые на рис. 1 обозначены как simple.

Кроме того, предусмотрен набор из *6* вариантов требований к пользователю при организации передачи параметров ([1] таблицы 26 и 27 на стр. 16-48 - 16-49). Эти варианты отличаются и по типам данных и по виду параметра (IN, OUT, ...). Все эти соглашения *должны быть* хорошо усвоены и тем, кто пишет реализацию серверных объектов и тем, кто описывает только вызовы (клиентская часть).

1.4. Одним из наиболее спорных моментов в связи с меппингом типов данных, является разделение типов данных на данные фиксированной длины (fixed-length) и переменной (variable-length) ([1] 16.8). Типами данных переменной длины объявлены Any, строки (bounded и unbounded string), последовательности (bounded и unbounded sequence), объектные ссылки, а также структуры и объединения, хотя бы одно из полей (членов) которых имеет тип переменной длины, и массивы, элементы которых - типа переменной длины. Такое разделение типов по постоянности длины, как видно из рис. 1, существенно влияет на меппинг параметров структур, объединений и массивов. Такая схема заставляет учитывать подобное различие типов при написании сигнатур методов серверных объектов, при организации возврата данных из этих методов и при организации вызовов этих объектов.

Естественно, возможны ситуации, когда, например, при небольшом изменении спецификации IDL, из-за изменения типа одного поля некоторой структуры (структурного типа) меняется качество «fixed или variable» самой структуры. Такое изменение влечет за собой необходимость переписывания текста программ во многих местах, как для клиента, так и для сервера. Как видно, определение понятия типа переменной длины является рекурсивным, и такое одно изменение типа может повлечь за собой изменение качества «fixed или variable» для всех составных типов, элементом которых является измененная структура. Конечно же,

везде и в клиенте и в сервере, где используются OUT и RESULT параметры любого из этих типов, понадобится соответствующее переписывание текста программы даже там, где само изменившееся поле структуры никак не используется.

1.5. Для любого составного типа данных, скажем T, в меппинге вводится специальный тип указателей на данные этого типа - T_var. Схема работы с параметрами со стороны клиента с помощью такого посредника существенно проще и является одинаковой для всех таких типов. Но, поскольку использование T_var для передачи параметров *не является обязательным*, для клиента зачастую гораздо проще оказывается передать или получить данные при вызове метода без использования посредника. В этом случае у него есть значительный шанс допустить ошибку, забыв о каком-нибудь из аспектов предусмотренных меппингом схем работы с памятью (см. *1.3.*).

1.6. Для программиста же, который реализует методы серверного объекта, все параметры передаются без каких-либо посредников типа T_var. Здесь уже никуда не денешься и от необходимости четко знать меппинг типов параметров ([1], табл. 26) и от того, что нужно удовлетворить все требования по работе с памятью, отводимой для параметров. Например, меппинг обязывает программиста для *всех* параметров вида INOUT (входные-выходные) явным образом заботиться о том, чтобы пришедшая от клиента память была удалена в случае возврата через такой параметр других данных (не тех, которые были на входе метода) (см. [1], 16.18 8).

1.7. В [1] 16.18 10 описан целый ряд случаев, когда запрещается передавать нулевой указатель как клиенту (IN и INOUT параметры), так и серверу (OUT и RESULT). Таким образом, программист вынужден заботиться о заполнении чем-то таких параметров и в тех случаях, когда эти параметры не используются другой стороной. Например, в случае, когда при выполнении некоторого метода возникает исключительная ситуация (Exception), вообще ничего, кроме информации о Exception

клиенту не передается. Но даже в этом случае запрещено просто возвращать нулевую ссылку.

1.8. Стандартным источником ошибок является также схема заполнения объектов типа T_var (будем говорить просто T_var). Во всех таких типах есть и конструктор вида T_var (T *) и оператор присваивания вида T_var &operator=(T *) (см. [1] 16.8.1). В обоих случаях память, указываемая параметром T*, поглощается объектом T_var, то есть, это обязательно должна быть динамически выделенная память, и она будет удалена либо при присваивании в этот объект другого значения, либо в деструкторе объекта. Такая схема заставляет явным образом копировать данные, находящиеся не в динамической памяти, перед заполнением их в T_var. Ошибка здесь приводит в дальнейшем к попытке удаления динамически не выделенной памяти. Ярким примером того, как не разрешается заполнять T_var является заполнение String_var вида:
CORBA::String_var var = “some string”;

Вместо такой естественной формы присваивания требуется явно скопировать нужную строку в динамически выделенную память. Специально для такого копирования строк предусматривается функция string_dup:
CORBA::String_var var = CORBA::string_dup (“some string”);

Для всех остальных типов данных, правда, функции типа string_dup меппинг не предусматривает. Поэтому, и выделение динамической памяти и ее заполнение, пользователь должен делать сам.

Итак, ошибки при использовании такого меппинга очень часто приводят к ошибкам работы с динамической памятью. А именно такие ошибки, исходя из опыта работы с различными реализациями языка C++, являются одними из наиболее трудно отлаживаемых. Это связано с их поздним обнаружением, т.е. с тем, что такие ошибки проявляются чаще всего не в момент их совершения, а где-то в процессе дальнейшей работы программы при последующей работе с динамической памятью. Необходимость отыскания источников подобных ошибок, конечно же, существенно усложняет и удлиняет процесс отладки программ.

1.9. Как было отмечено в *1.8.*, при заполнении T_var на основании указателя на данные типа T, в T_var запоминается это значение указателя, никакого копирования данных не происходит. Это единственный предусмотренный способ заполнения T_var не через другой объект типа T_var, а на основании самих данных типа T (см. [1] 16.8.1). Таким образом, заполнение из const T* невозможно без явного копирования в динамически выделенную память ([1] 16.8.1. 16).

1.10. Заполненный по умолчанию T_var не может быть использован для доступа к данным типа T, поскольку в нем до первого явного присваивания значения T* (см. *1.8.*, *1.9.*) хранится нулевая ссылка. По этой же причине явно не заполненный T_var не может быть использован для возврата OUT параметров и результата метода.

1.11. Проблема с некорректным начальным значением существует и для самих типов данных. Так, для объединений (union) нет никакого начального значения и в принципе запрещено использовать дескриптор объединения без его явной предварительной инициализации. Правда, никакого механизма контроля некорректного доступа к незаполненному объединению не предусматривается (см. [1] 16.10 1).

1.12. Специально для заполнения массива некоторого типа A в объект типа Any введен тип объектов A_forany, не имеющий аналогов среди других типов данных. Как отмечено в [1] 16.12 5 A_forany обязательно отличается от A_var и имеет другую схему работы с памятью: деструктор A_forany не уничтожает тот массив, на который этот объект ссылается. Причина появления такого специального типа состоит в том, что, если элементы массива типа A имеют некоторый тип B и B - не массив, то в меппинге уже предусмотрена операция заполнения Any из B*. А как известно, в C++ (как и в C, конечно) переменная типа B[] (то есть A) используется также, как если бы она была типа B*. Таким образом, в смысле передачи параметра, компилятор не отличает массив от указателя на его первый элемент. И при попытке реализовать

операцию заполнения (<<=) Any из A возникает конфликт с такой же операцией для B*.

Кстати говоря, даже при наличии класса A_forany, пользователь может ошибочно воспользоваться операцией <<= заполнения Any для массива и *незаметно* для себя получить такой эффект, что вместо всего массива в Any окажется лишь его первый элемент.

2. Альтернативный меппинг IDL/C++

2.1. Использование оболочек

Если сравнить обсуждаемый здесь меппинг CORBA IDL/C++ с меппингом CORBA IDL/C, хотя бы только в части схемы передачи параметров (см. [1] 14.18 таблицы 20, 21, 22), становится очевидным, что меппинг IDL/C++ разрабатывался совместно с меппингом IDL/C и они содержат много общих решений. Так, меппинг типов параметров и соглашения о работе с памятью при передаче параметров в обоих меппингах практически совпадают (разница лишь в использовании оператора & в случае C++ меппинга вместо оператора *). Конечно же, возникает вопрос, можно ли избавиться от перечисленных выше проблем в меппинге C++, если не пытаться следовать «в курсе» меппинга C, а использовать возможности C++, которых нет в C.

Рассмотрим следующий вариант изменений в меппинге IDL/C++. Введем в использование *для всех составных типов* (структуры, объединения, массивы, последовательности, объектные ссылки, строки и Any) специальные классы - *оболочки* (*посредники*). Как и в случае с T_var в CORBA меппинге, для каждого составного типа данных имеется собственный класс таких оболочек. Но принцип работы с памятью в этом случае будет принципиально другим. Главное же в этом подходе то, что при работе с ORB'ом пользователю предлагается проводить все необходимые манипуляции с данными только через подобных посредников. В частности, только с их помощью организуется передача параметров методов и получении результатов для составных типов (это сразу решает проблему *1.5*).

При использовании такого меппинга пользователю не придется изучать и запоминать целое множество вариантов требований к работе с памятью (пп. *1.3.* и *1.6.*). С помощью оболочек акцент в работе с памятью переносится с организации передачи данных в методах (как клиентом так и сервером) на однотипную работу с оболочками, из заполнение и получение (просмотр) данных с их помощью. Аспекты работы с памятью, необходимой для параметров и результатов, при вызовах объектов скрываются внутренними механизмами оболочек и, таким образом, не затрагивают пользователя.

Итак, вместо приведенной ранее на рис. 1 таблицы с описанием CORBA меппинга передачи параметров и результатов для C++ здесь этот меппинг описывается гораздо проще (см. рис. 2). Это решает проблему *1.3* сложности меппинга и делает не нужным разделение данных на fixed и variable (решение проблемы *1.4.*).

Тип данных	**In**	**Inout**	**Out**	**Return**
Простой тип (S)	S	S &	S &	S
Составной тип (T)	const T_cvr &	T_cvr &	T_cvr &	T_cvr

Рис. 2. Предлагаемый меппинг типов параметров и результатов.

Перейдем теперь собственно к рассмотрению схемы работы с оболочками. Для каждого составного типа T вводится тип объектов-оболочек *T_cvr* (от cover). Предлагается два варианта восприятия клиентом схемы работы с такими посредниками. Первый из них рассчитан на пользователя, который хочет иметь максимально простой механизм для работы с данными, обеспечивающий возможность как можно меньше думать об особенностях работы с памятью. Такой пользователь выбирает простоту, но платит некоторой потерей эффективности. Второй вариант рассчитан на пользователя, которому важна эффективность работы программы, при этом он использует более сложный механизм работы с оболочками, позволяющий ему учитывать особенности работы с разными классами памяти языка C++ не только при работе с собственными данными, но и при работе с самими оболочками.

2.2. Первый подход. Базовый механизм

В первом подходе пользователь может себе представлять оболочки как «умные» контейнеры, *содержащие* данные соответствующего составного типа. Основой концепции работы с памятью здесь является тезис о том, что память, содержащаяся в оболочках *никогда* не доступна для управления пользователем. То есть, оболочки сами создают память под данные и уничтожают ее. Пользователь же лишь может указать, откуда скопировать данные в эту память, может посмотреть ее содержимое, и может изменять значения элементов составных данных, хранимых в оболочке.

Важной особенностью оболочек является то, что их не обязательно явным образом инициализировать. Изначально они уже содержат данные, заполненные по умолчанию: все элементы данных, имеющие конструктор, заполнены с помощью конструктора по умолчанию, остальные - просто обнулены. Это гарантирует, в частности, возможность возврата результатов (в том числе через INOUT и OUT параметры) методов без их явного заполнения (см. проблему *1.7.*). Проблема *1.10*. также решается с помощью этого механизма умолчания : оболочки (в отличие от T_var в CORBA меппинге) могут одинаково использоваться без явного заполнения и после такового.

Решается также и проблема *1.11.*, связанная с тем, что в соответствии с CORBA меппингом объединение изначально находится в некорректном состоянии. В случае использования до явного заполнения значения оболочки, содержащей данные типа объединения используется соглашение о том, что начальным значением дескриптора объединения является либо значение по умолчанию (default), указанное в IDL спецификации для этого типа, либо, если таковое умолчание не было определено, то используется первое определенное в IDL спецификации значение дескриптора (схема, используемая при инициализации объединений в C++ - заполняется именно первое поле). В качестве данных объединение содержит данные того типа, который соответствует выбранному указанным образом дескриптору, и заполнены также по умолчанию, используемому для этого типа.

Присвоить новое значение данным (типа T), хранимым оболочкой, можно с помощью операции присваивания. Источником

при этом может служить переменная типа T_cvr (другая оболочка этого же типа) или T (собственно данные). При этом, в обоих случаях происходит *глубокое копирование* данных на память, содержащуюся в оболочке, в которую производится присваивание. Глубокое копирование подразумевает создание полностью независимой от источника копии. То есть, например, при копировании строки, не зависимо от того, на каком уровне вложенности в составном типе она находится, не просто копируется указатель на строку, а выделяется новая память под копию строки. Таким образом, после операции присваивания оба операнда продолжают существовать независимо друг от друга, поскольку не ссылаются на общие данные. За счет этого достигается возможность присваивать значение оболочки из переменной *любого* класса памяти.

Кроме того, значение оболочки может быть присвоено в переменную типа T. При этом также происходит глубокое копирование данных. То есть, и в этом случае оболочки ведут себя так, будто они являются самими данными типа T.

Проиллюстрируем схему работы описанных операций присваивания на примере простейшего составного типа - структурного типа S с единственным полем типа long. Пусть имеются четыре переменные с указанными начальными значениями:

```
                              // начальное значение поля структуры:
S_cvr    Cvr;          //  1
S_cvr    AnotherCvr;   //  3
S        VarData;      //  3
const S  CnstData;     //  5
```

Начальное состояние переменных схематически изображено в левой части рис. 3. Оболочки типа S_cvr представлены в виде эллипсов, а данные типа S - в виде квадратов. Различный тип линий, которыми нарисованы указанные фигуры, используется для различения областей памяти, где расположены соответствующие данные. Рисунок 3 иллюстрирует изменения, происходящие с переменными после выполнения операции присваивания над различными парами этих переменных:

1. Cvr = VarData;
   Cvr = CnstData;

в обоих случаях данные (структура {3}) из VarData или CnstData *копируются* в память под структуру типа S, содержащуюся в оболочке Cvr;

2. VarData = Cvr;
   данные ({1}) из структуры, содержащейся в оболочке Cvr, *копируются* в переменную VarData;
3. Cvr = AnotherCvr;
   данные ({5}) *копируются* из структуры, содержащейся в оболочке AnotherCvr в структуру из оболочки Cvr.

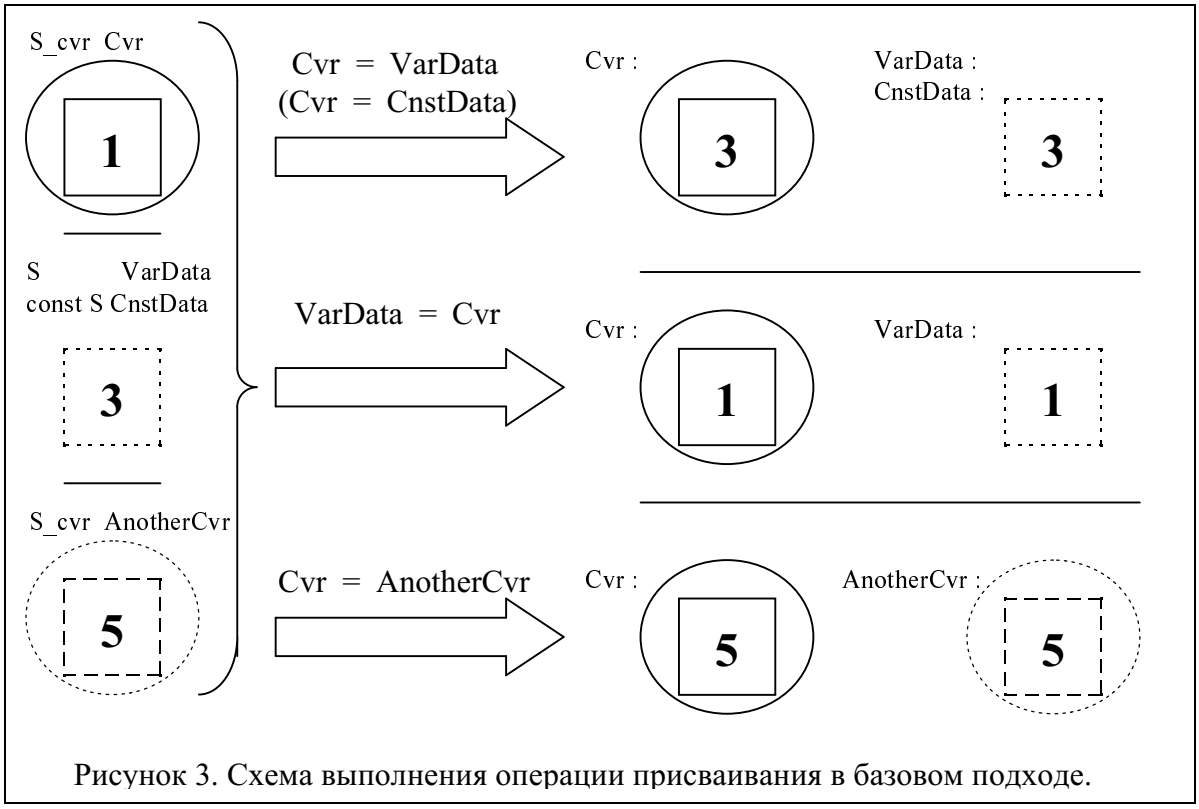

Рисунок 3. Схема выполнения операции присваивания в базовом подходе.

Кроме описанных операций присваивания в посреднике также, как и для T_var объектов CORBA меппинга, предусмотрены операторы для доступа к элементам указываемых им данных составного типа. То есть, в зависимости от типа могут использоваться операторы -> (структуры, объединения, последовательности, объектные ссылки, Any), [] (строки, последовательности, массивы), <<= и >>= (Any). Также для всех типов посредников определены операторы заполнения Any из посредника (Any <<= T_cvr) и посредника из Any (Any >>= T_cvr). После выполнения действий, предусмотренных этими

операторами, как обычно, память обоих операндов не пересекается. Таким образом, и для массивов необходимый механизм работы с Any реализуется посредником, т.е. нет необходимости в особом типе объектов вида T_forany (решена проблема *1.12*.).

Таким образом, пользователь может использовать оболочки не только для простой передачи параметров в методы, но и для манипуляций (изменения содержимого) над всеми составными типами данных. Очевидно, что при выполнении всех указанных действий пользователю не приходится заботиться о памяти, содержащейся в оболочке.

Нужно заметить, что в операции присваивания для оболочек вторым операндом никогда не может быть указатель на данные (T*). Напомним, что в CORBA для переменных типа T_var и T* подобные присваивания разрешены. Например:

```
T      *ptr = ...; // некоторая инициализация
T_var var;

var = ptr;
ptr = var;
```

Однако, как уже было отмечено ранее, после таких присваиваний обе переменные-операнды фактически указывают на одни и те же данные, что и является одним из источником проблем при работе с памятью. Итак, отсутствие возможности непосредственного запоминания указателя оболочкой в рамках этого подхода решает проблему *1.8*.

При необходимости же присвоить в оболочку (скажем, Cvr) значение, указываемое переменной (скажем, Ptr) типа T* пользователю достаточно написать Cvr = *Ptr. Очевидно, что такое присваивание сохраняет независимость переменных, поэтому не имеет значения класс памяти и самой переменной Ptr и данных, на которые она указывает (решение проблемы *1.9*.). Однако, ситуация с обратным присваиванием (из оболочки в указатель) обстоит сложнее. Существующую здесь проблему легко проиллюстрировать на примере использования пользователем оболочки в качестве фактического параметра.

Пользователь может захотеть передать данные оболочки, в некоторую *свою* функцию, не вызываемую через объектную ссылку, то есть когда работа идет не через ORB (Object Request Broker). При этом, если тип соответствующего параметра пользовательской функции - T*, оболочка может быть передана в качестве фактического параметра в том случае, если у нее предусмотрен оператор преобразования в T*. Но различные такие пользовательские функции могут иметь различные соглашения о схеме работы с параметрами. Например:

```
void f (T* dust) {
      delete dust; // удаление памяти по указателю
}

void g (T* info) {
      cout << info; // вывод содержимого параметра на экран
}
```

Очевидно, что невозможно зафиксировать какой-то конкретный механизм преобразования из T_cvr в T* так, чтобы в обе функции можно бы было передать оболочку Cvr (f (Cvr) и g (Cvr)) и это не вызвало бы никаких проблем. Здесь может быть два варианта:

1. При преобразовании T_cvr в T* выделяется динамическая память, в которую копируются данные из оболочки. В этом случае эта память никогда не будет уничтожена при вызове g (Cvr).
2. При преобразовании T_cvr в T* выдается указатель на память, содержащуюся в оболочке. В этом случае g (Cvr) проблем не вызывает, но f (Cvr) уничтожает память, содержащуюся в оболочке без ее ведома. Возможность такого действия противоречит тому, что лишь оболочка может управлять памятью, отведенной в ней под данные.

Таким образом, наличие оператора преобразования T_cvr в T* недопустимо, поскольку такой оператор в основном используется неявно (в том числе при присваивании из T_cvr в T*), но во многих случаях вызывает проблемы работы с памятью. А ведь именно для решения в первую очередь таких проблем и рассматривается техника использования оболочек.

Конечно же, отсутствие оператора преобразования T_cvr в T* можно обойти:

```
T *tmp_ptr = new T ((T)Cvr);// выделение динамической памяти,
                            // в которую копируются
                            // данные из оболочки
f (tmp_ptr);

T tmp_dt = Cvr; // копирование данных из оболочки
                // во временную переменную
g (&tmp);
```

То есть, пользователь, зная семантику указателя - параметра *своей* функции (здесь f или g) явным образом обеспечивает нужные действия для передачи данных из оболочки в функцию. Не выходя за рамки предлагаемой в этом подходе концепции работы с памятью, для упрощения передачи данных в пользовательские функции имеет смысл предусмотреть специальный метод самой оболочки - *T * **copy ( ).***

Этот метод выдает указатель на дополнительно выделяемую динамическую память, в которую скопированы данные из оболочки. *Явное* использование такого метода подразумевает, что пользователь берет на себя управление выдаваемой этим методом памяти. Теперь вызов функции f может быть произведен проще, без введения временных переменных:

```
f (Cvr . copy ());
```

Нужно отметить, что для подобного упрощения вызова функции g, нужно было бы предусмотреть метод, который бы не делал никакого копирования, а просто выдавал бы указатель на память, содержащуюся в оболочке (ведь f не модифицирует входные данные). Однако, подобная функция не может быть частью этого первого подхода, поскольку она нарушает принцип ортогональности (независимости) памяти, используемой пользователем и оболочками (такой метод - T * look () - предусмотрен во втором подходе).

Итак, все действия над оболочками, предусмотренные в этом подходе, обеспечивают такой механизм, при котором внутренний

механизм оболочек работы с памятью не известен пользователю, то есть является для него прозрачным. Это, разумеется, полностью исключает возможность возникновения у пользователя проблем при работе с памятью из-за необходимости работать с параметрами и результатами методов.

2.3. Второй подход. Механизм оптимизации

Однако, при использовании концепции оболочек из первого подхода приходится платить эффективностью за то, что оболочки берут на себя всю работу с памятью. Глубокое копирование, возникающее в операциях присваивания и copy(), и дополнительные операции выделения и удаления динамической памяти могут существенно отражаться на времени работы программы.

И именно поэтому для пользователей, которым важна эффективность программы, и которые свободно себя чувствуют в работе с разными видами памяти языка C++, предлагается второй подход в восприятии техники объектов-посредников.

Все изложенные ранее действия над оболочками доступны и во втором подходе - они являются ядром этого подхода. Однако, здесь оболочки должны восприниматься не как «умные» контейнеры, а как «умные» указатели на данные. Пользователь теперь понимает, что в пределах ядра (операции первого подхода) оболочки работают с динамической памятью. Изначально оболочка содержит нулевой указатель на данные. При первой же операции над ней, требующей наличия памяти под данные, для этих данных неявно выделяется динамическая память, которая заполняется по умолчанию описанным ранее образом. Это касается всех рассмотренных в первом подходе операций:

- операций присваивания, как в оболочку, так и из нее;
- операций доступа к элементам данных (->, [], <<=, >>=);
- метода copy ().

Эта динамическая память удаляется при уничтожении оболочки.

Основным содержанием второго подхода является добавление к интерфейсу оболочек ряда специальных методов. Эти методы позволяют избежать выделения и удаления динамической памяти

самими оболочками и глубокого копирования данных. Это достигается за счет использования оболочкой предоставляемой пользователем памяти. Таким образом, в этом подходе, в целях увеличения эффективности, допускается пересечение памяти пользователя и оболочек. Конечно же, при использовании этих дополнительных методов необходимо соблюдать осторожность и выполнять требования по работе с памятью, накладываемые каждым из методов.

Итак, в этом втором подходе для любого составного типа T соответствующий ему класс объектов-оболочек T_cvr имеет следующие дополнительные методы:

1. *void* ***assume*** *(T*)* - оболочка берет на себя управление выделенной пользователем динамической памятью. Здесь пользователь обязан предоставить в качестве параметра указатель именно на динамически выделенную память. После такой передачи памяти пользователь не должен сам пытаться уничтожить эту память.

2. *void* ***point*** *(T*)* - оболочка запоминает указанную ссылку на данные, но управление предоставленной памятью остается за пользователем. То есть, при уничтожении оболочки или замене данных, на которые она указывает (любым из предусмотренных способов), удаления памяти не происходит. Таким образом в оболочку можно заполнить данные любого вида, даже из стека (автоматическая память). Однако, пользователь должен обеспечить корректность данных, на которые после этой операции будет указывать оболочка. Так, например, если в оболочку с помощью этого метода передан указатель на локальную переменную некоторого блока, то эта оболочка не должен использоваться вне такого блока до изменения ее указателя на данные.

3. *T ** ***look*** *()* - этот метод выдает указатель на данные, указываемые оболочкой. Результат не может быть равен нулю. При выполнении этого метода над пустой (незаполненной) оболочкой предварительно происходит ее заполнение, т.е. заполнение по умолчанию динамически выделяемой памяти. Эта

операция, в частности, может быть использована для передачи в пользовательскую функцию указателя на данные из оболочки в том случае, если эта функция не удаляет данные по полученному указателю (как функция g() в приведенном ранее примере).

4. *T * **give_data** ()* - операция, обратная к assume (). Оболочка передает владение данными пользователю. После этой операции оболочка пуста (содержит нулевую ссылку), а пользователь владеет полученной памятью (т.е. должен будет ее удалить после использования). Это обеспечивается даже в том случае, если оболочка не владела памятью (получила ее по операции take ()): в таком случае данные дублируются.

5. *void **release** ()* - операция «чистки» данных: если оболочка владела памятью, то эта динамическая память освобождается. После этой операции оболочка находится в том же состоянии, что и сразу после ее создания до первого использования. То есть, после выполнения этой операции ссылка на данные принимает нулевое значение.

6. *void **assign** (const T_cvr &another_cvr)* - операция, альтернативная операции присваивания из другой оболочки, после которой оба операнда (оболочки) владеют собственной памятью и существуют далее независимо друг от друга. Операция assign() дает возможность произвести операцию присваивания из одной оболочки в другую так, будто они представляют собой обычные указатели на данные. После такой операции обе оболочки ссылаются на одну и ту же память и, таким образом, изменения в одной из них отражаются на обеих. Оболочки совместно владеют памятью таким образом, что она будет уничтожена лишь после того, как обе оболочки перестанут ею владеть. Такой механизм обеспечивается с помощью стандартного механизма счетчиков ссылок (refcounters).

С помощью использованных ранее в рис. 3 обозначений смысл приведенных здесь операций поясняется также схематически на рис. 4. Дополнительно к уже описанным для рис. 3 соглашениям здесь используются также следующие:

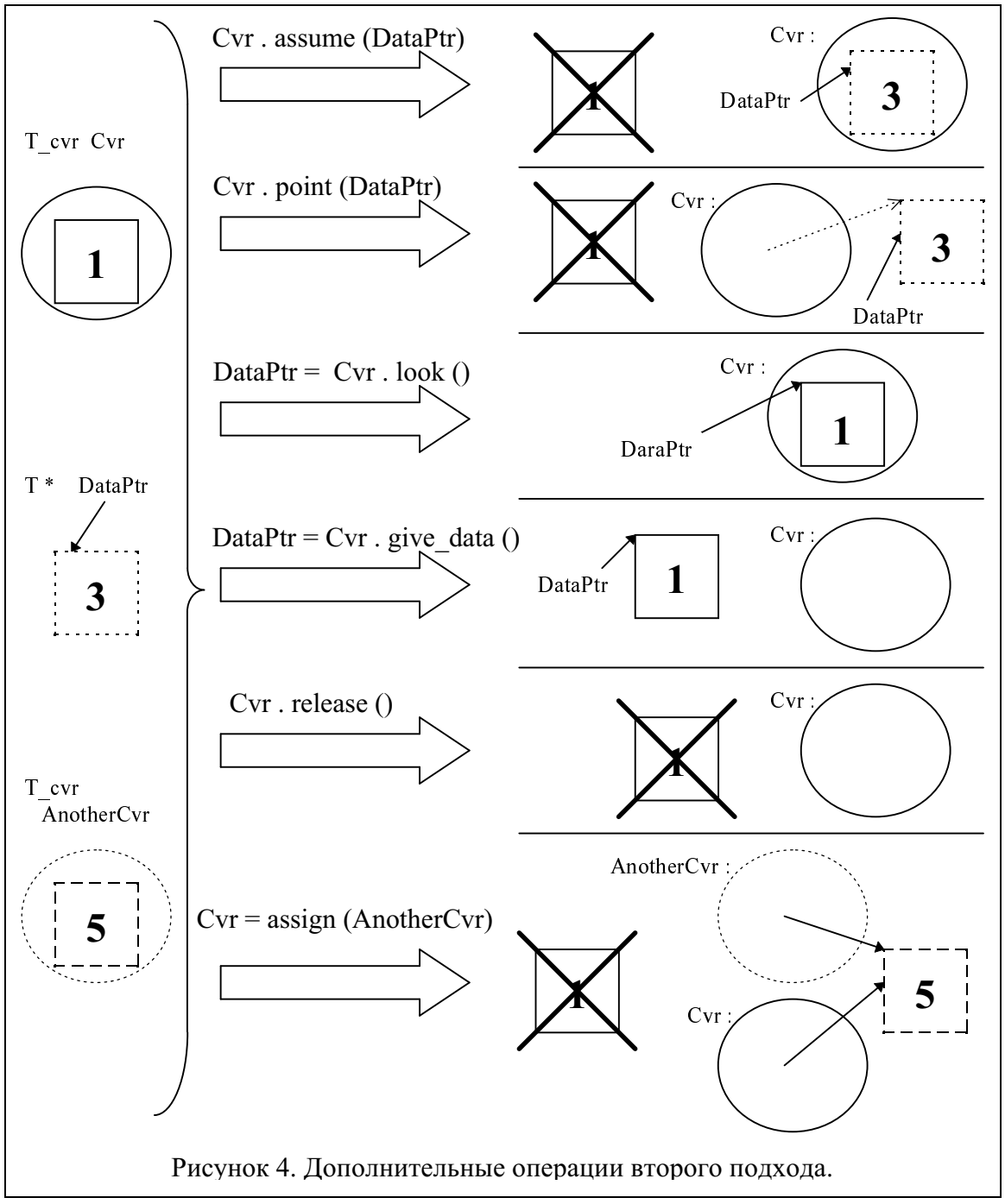

Рисунок 4. Дополнительные операции второго подхода.

- уничтожение памяти обозначается двумя скрещивающимися отрезками;
- квадрат внутри эллипса обозначает (как и на рис. 3) то, что оболочка «единолично» владеет памятью под структуру;
- квадрат вне эллипса, указываемый стрелкой, исходящей изнутри эллипса обозначает то, что оболочка либо вообще не владеет памятью (стрелка нарисована прерывистой линией), либо она

владеет памятью совместно с другой оболочкой (сплошная линия стрелки).

Важно отметить, что указанные операции, расширяя возможности работы с памятью, управляемой объектами-посредниками, дают возможность использовать различные типы памяти для передачи параметров в методы. В частности, все способы работы с памятью при передаче параметров, предусмотренные в CORBA, могут быть организованы с помощью этих операций и здесь. Но передача параметров может быть организована иногда более эффективно, чем это позволяет сделать CORBA меппинг. Так, например для OUT параметров - последовательностей, массивов, структур и объединений переменной длины (см. *1.4.*) и типа Any CORBA меппинг не дает возможности клиенту выделять память под выдаваемый через такой параметр результат. Через такой OUT параметр будет возвращена динамически выделенная (либо в вызванном серверном объекте при локальном взаимодействии, либо в ORB'е при удаленном) память, которую клиенту еще нужно будет не забыть удалить.

Предлагаемый же механизм оболочек позволяет для всех указанных типов OUT параметров клиенту предоставить собственную память и, таким образом, избежать накладных расходов, связанных с выделением и дальнейшим уничтожением динамической памяти. Пример:

```
MyStruct str;
MyStruct_cvr cvr;

cvr . point (str);
some_object -> some_method (cvr);  // OUT параметр
cout << cvr->field1 << cvr->field2; // распечатка результата
```

Конечно же, и серверная программа может пользоваться приведенными операциями оболочек для оптимизации. Например, если серверному объекту необходимо вернуть в качестве значения OUT или INOUT параметра значение, хранимое объектом не в автоматической памяти (т.е. не в стеке), CORBA меппинг в этом случае требует выделить динамическую память и произвести глубокое копирование данных в эту память. И все это лишь для

передачи данных, которые и без копирования доступны и тогда, когда объект не активен. С помощью рассматриваемого механизма оболочек такие данные можно возвращать и без всех этих весьма «дорогих» по времени действий:

```
some_object_realization::some_method (MyStruct_cvr cvr) {
        static MyStruct static_var;     // статические данные
        MyStruct auto_var;              // автоматические данные
        …
        cvr . point (static_var);       // запоминание указателя на
                                        //статические данные
        …
        // или :
        cvr = auto_var;                 // копирование из стека
        …
}
```

Если сопоставить приведенные примеры частей программ клиента и сервера, видно, что клиент и сервер могут обеспечить механизм взаимодействия без использования динамической памяти. При этом, клиент «на всякий случай» предоставляет свою память под результат. Она используется ORB'ом при удаленном взаимодействии (ORB заполняет эту память результатом, полученным по сети из серверного процесса) и сервером при локальном вызове (в этом случае клиент видит непосредственно результат работы серверного объекта), если OUT параметр заполнен с помощью операции присваивания (как в случае с переменной auto_var). Конечно, серверный объект мог воспользоваться статической (ситуация с переменной static_var) или динамической памятью. В этом случае при локальном взаимодействии клиент будет видеть через оболочку не свою память, а предоставленную сервером. Именно поэтому, если клиенту четко не известна стратегия работы сервера с возвращаемыми параметрами, полученный результат нужно просматривать именно через оболочку (cvr), а не с помощью непосредственной работы с предоставленной памятью (переменная str). Но, еще раз напомним, что думать о таких вещах стоит лишь при острой необходимости повысить эффективность программы.

До сих пор речь шла лишь о повышении эффективности передачи параметров методов. Передача результата не затрагивалась. Рассмотрим теперь следующий простой пример:

```
// клиент
MyStruct_cvr cli_cvr;
cli_cvr = some_object -> some_method_with_result ();

// сервер
MyStruct_cvr
some_object_realization::some_method_with_result () {
      MyStruct serv_cvr;
      …
      return serv_cvr;
}
```

В этом примере данные из локальной автоматической переменной serv_cvr серверного объекта должны попасть в переменную cli_cvr клиента. Однако, как выяснилось при тестировании программ, откомпилированных доступными современными компиляторами языка C++, даже при непосредственном вызове метода без посредничества ORB'а реально компилятор не производит прямой операции присваивания из serv_cvr в cli_cvr. При выходе из метода порождается временный объект типа MyStruct_cvr с помощью конструктора копирования (из переменной serv_cvr). Этот конструктор выделяет динамическую память для сохранения данных в создаваемом объекте. И уже лишь после выхода из метода происходит присваивание из временного объекта в cli_cvr, то есть осуществляется глубокое копирование в память из cli_cvr. После присваивания временный объект уничтожается: удаляется выделенная для него динамическая память.

Также реализуется часто необходимая передача результата «транзитом» вида return method (). В этом случае компилятор C++ тоже использует механизм создания временного объекта и элементарная на вид операция на самом деле является источником замедления работы программы.

Реально корнем проблемы здесь является то, что в целях обеспечения четкого и ясного механизма работы с памятью, зафиксирован механизм, при котором после копирования данных из одной оболочки в другую они остаются независимыми друг от друга, то есть не ссылаются на общую память. Механизм же, при котором один из объектов будет после операции копирования указывать точно на те же данные, что и второй, реализован в явной операции *assign* (). Конечно же, если бы именно такой механизм использовался при возврате результата, то не было бы практически никаких лишних накладных расходов. Но в рамках типа T_cvr это не осуществимо, т.к. один и тот же конструктор копирования используется и при явной инициализации переменных (например, T_sh cvr = another_cvr) и при заведении временного объекта для возврата результата.

Возможным решением проблемы является введение в рамках второго подхода альтернативного к T_cvr типа, скажем T_res. Этот новый тип объектов-оболочек идентичен T_cvr, за исключением всего того, что связано с копированием из других объектов-оболочек, как типа T_cvr, так и типа T_res. Если в операции присваивания (в том числе, в конструкторе копирования) участвует объект типа T_res, то копирование производится с минимальными затратами: оба операнда после операции будут ссылаться на одну и ту же память. То есть, это такая же схема, как и в методе *assign* интерфейса T_cvr. Конечно, для симметрии метод *assign* в T_res следует заменить на другой метод, скажем void dupl_to (T_res &), который позволяет явным образом скопировать данные из данной оболочки типа T_res в *память,* указываемую другим посредником. То есть, оба типа оболочек предоставляют возможность копирования двумя способами. Отличаются эти типы тем, какой из этих двух способов используется явно, а какой по умолчанию (в операции присваивания и конструкторе копирования).

Теперь, при написании методов серверных объектов, если есть необходимость оптимизировать возврат результата типа MyStruct, в качестве типа результата можно использовать второй тип оболочек T_res. Например:

```
MyStruct_res
some_object_realization::some_method_with_result () {
        MyStruct serv_cvr;
        …
        return serv_cvr;
}
```

Как видно, изменилась лишь сигнатура метода (тип результата), тело реализации осталось прежним, однако теперь компилятор C++ создаст для возврата результата временный объект типа MyStruct_res, который будет просто указывать на ту же память, что и переменная serv_cvr. С помощью использования счетчика ссылок, в результате такого присваивания и удаления автоматической переменной serv_cvr реально происходит передача владения памятью от serv_cvr к временному объекту. Важно отметить, что механизм возврата результата через T_cvr и через T_res обеспечивает возможность не запоминать результат метода, то есть, как это разрешено в C++, можно вызывать метод без сохранения его результата. При этом, возвращаемая память будет удалена при удалении временного объекта-оболочки, если переменная serv_cvr владела этой памятью и притом единовластно. В CORBA меппинге, если не сохранить результат «переменной» длины, то память из под него останется не освобожденной.

Конечно же, как клиент, так и серверный объект могут равноправно работать с обоими типами оболочек, в зависимости от необходимой схемы присваиваний.

Нужно отметить, что до сих пор подразумевалось, что пользователю доступны порождение и удаление как самих оболочек, так и данных тех типов, на которые они могут указывать. Однако, объектные ссылки представляют собой единственное исключение из этого правила. Дело в том, что понятие объектной ссылки напрямую связано с ORB'ом и вне его контекста не имеет смысла. Механизм работы объектных ссылок и данные, в них содержащиеся, необходимые для организации взаимодействия с объектом, прозрачны для пользователя. Более того, пользователь не может создавать и уничтожать данные типа «объектная ссылка». Таким образом, пользователь работает только с оболочками в виде

контейнеров из первого подхода. Единственный метод, который имеет смысл оставить для таких посредников - метод *release* (нет и метода copy из первого подхода). Типы T_cvr и T_res здесь совпадают.

Итак, описанные в самом начале статьи проблемы работы с объектными ссылками (п. *1.1*, *1.2*.) также решены с помощью оболочек. Пользователь теперь просто не имеет возможности запутаться при работе с объектными ссылками, теперь не нужно различать типы A_ptr и A_var, а также думать, где нужно использовать метод duplicate и следить за сбалансированное его использование с методом release. Механизм оболочки выполняет сам все необходимые действия даже без ведома пользователя. Метод же release остался лишь для того, чтобы пользователь мог отказаться от текущей ассоциации оболочки с объектной ссылкой еще до присваивания новой ассоциации, либо удаления посредника. Наличие одной этой операции, конечно, не вызовет проблем, аналогичных описанным.

Итак, использование первого из двух предложенных подходов к работе с оболочками полностью решает описанные в начале статьи проблемы CORBA IDL/C++ меппинга. Второй же подход предоставляет более гибкий и эффективный механизм работы с данными, однако требует от пользователя хорошего знания схемы работы с разными видами памяти языка C++. Выбор более подходящего варианта целиком лежит на самом пользователе.

Заключение

Конечно, после рассмотрения предложенной альтернативы меппинга возникает закономерный вопрос: каково может быть место такого нестандартного меппинга с учетом того, что CORBA меппинг сейчас фактически уже воспринимается как стандарт в области организации распределенных объектных систем.

Разумеется, описанный в этой статье подход не стоит воспринимать как претензию на изменение стандарта. Нет необходимости переписывать в соответствии с описанной технологией уже отлаженные приложения, ориентированные на CORBA меппинг. Однако, при создании «с нуля» новых

приложений описанный здесь вариант меппинга может оказаться более удобным, как для написания программ, так и для их отладки. В частности, этот механизм представляется более удобным и надежным при переработке ранее написанных приложений на языке C++, не ориентированных на технологию CORBA (проблема унаследованных систем - legacy), с целью организации из этих приложений сервисов, доступных через ORB. Ведь сама технология использования ORB'ов позволяет различным элементам (процессам) общего объектного пространства быть реализованным и на разных языках, и работать через ORB'ы различных производителей. Так, почему же нельзя использовать и несколько разных версий меппинга одного и того же языка?

Необходимо заметить также, что стандарт CORBA реально носит рекомендательный характер. Каждая из реализаций ORB разных производителей даже на одном и том же языке программирования имеет собственный меппинг, до той или иной степени соответствующий CORBA, но и зачастую эти реализации содержат собственные проработки в области технологии организации взаимодействия объектов. Таким образом, и описанные в этой статье предложения могут быть реализованы как дополнение к меппингу C++ CORBA. В частности, базовую часть описанной альтернативы меппинга (первый подход), не затрагивающую вопросы эффективности, представляется возможным реализовать в качестве надстройки над C++ ORB'ом, совместимой с реализацией стандарта CORBA любого производителя. В такой универсальной надстройке могли бы быть отражены и многие возможности из второго из рассмотренных подходов. Правда, без базиса в виде некоторой конкретной реализации ORB'а нельзя рассчитывать на особую эффективность. Зато в таком виде, когда описанный подход реализуется не как альтернатива, а как дополнение к существующему стандарту, конечно же, существенно расширяется круг потенциальных пользователей описанной технологии.

Литература

1. The Common Object Request Broker: Architecture and Specification. Revision 2.0, July 1995
2. В.П. Иванников, К.В. Дышлевой, В.И. Задорожный. Спецификация метанаращиваний для эффективного метаобъектного контроля. Программирование, N 4, 1997 г.
3. V.Ivannikov, R.Kossman, and other. An Architectural Framework for Semantic Inter-Operability in Real-Time Distributed Object Systems, ISP/Nortel technical report. Moscow, November 1995
4. Б. Страуструп. Язык программирования C++. М., “Радио и Связь”, 1991
5. Bjarne Stroustrup. The C++ Programming Language, Second Edition. Murray Hill, NJ: AT&T Bell Laboratories, reprinted 1992
6. Steve Vinosky. A review of the IDL C++ Mapping Submission. OMG TC Document 93.12.19
7. Peter B. Kessler. Iona-NEC-SunSoft. IDL C++ Language Mapping: Ease of Use Layering. OMG TC Document 93.12.2
8. Michael L. Powell. Iona-NEC-SunSoft. IDL C++ Language Mapping: Rationale and Trade-offs. OMG TC Document 93.12.1
9. OMG RFP Submission. IDL C++ Language Mapping Specification. IONA Technologies Ltd., NEC Corporation, SunSoft Inc., OMG TC Document 93.11.6, November 15, 1993