Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

DPL Programmers' Manual

Nicola Howarth, Dave Otway, Owen Rees, Andrew Watson

Abstract

As computer networks become ever-larger, it is becoming clearer that writing distributed programs to exploit their potential is difficult and error-prone. Techniques for addressing this problem via use of programming tools are described in APM.1020 “Abstract and Automate”; DPL is an experimental notation for validating this approach. It is envisioned that the lessons learned in designing and using DPL will be carried forward into future distributed programming systems.

DPL combines the functions of Interface Definition Language and application code wrapper for specifying how distributed program objects conforming to the ANSA Computational Model (see APM.1001) invoke one another. DPL is self-sufficient and independent of specific data types, making it a truly abstract notation usable with a spectrum of programming languages.

This is the draft programmers’ manual for DPL. This manual is aimed at the applications programmer who is familiar with a high level language (such C or Pascal) and an operating system such as Unix or VMS. DPL is more formally defined in APM.1015 “DPL reference manual”.

APM.1014.01 **Approved** 29th November 1995
Technical Report

Distribution:
Supersedes:
Superseded by:

DPL Programmers' Manual

DPL Programmers' Manual

Nicola Howarth, Dave Otway, Owen Rees, Andrew Watson

APM.1014.01

29th November 1995

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Contents

1	**1**	**Overview**
3	**2**	**Introduction**
3	2.1	The problem space
4	2.2	Basic concepts
5	2.3	ANSAware and DPL
5	2.4	The manual
7	**3**	**A distributed applications primer**
7	3.1	Abstraction and programming models
7	3.1.1	The shared memory problem
9	3.1.2	Partial failure in distributed systems
9	3.2	Anatomy of a service
11	3.3	Objects: controlling co-location of services
12	3.3.1	Relationship to object oriented languages
14	3.4	Characteristics of services
14	3.4.1	First Class status
14	3.4.2	Service transparencies
15	3.4.3	Use of service references
16	3.5	Configuring distributed systems
16	3.5.1	Dynamic Binding: support for reconfiguration
18	3.5.2	Trading: support for service reuse
19	3.6	Types: describing the way a service is used
19	3.6.1	Interface types
20	3.6.2	Service types
20	3.6.3	Types and trading
21	3.7	Activities: handling concurrency and communication
23	3.8	Separation mechanisms
25	3.9	Summary
27	**4**	**DPL Syntax and Semantics**
27	4.1	Overall structure of a DPL program
28	4.2	Interface lists and Expressions
31	4.3	Interface constructors
31	4.4	Operations and terminations
33	4.5	Type constructors
34	4.6	Object constructors
34	4.7	Fixed and variable bindings
34	4.7.1	Fixed bindings
35	4.7.2	Variable bindings
36	4.7.3	Fixed vs Variable bindings
36	4.8	Invocations
36	4.8.1	Making invocations

37 4.8.2 Failure semantics of invocations
39 4.8.3 Behaviour of the invoked operation
39 4.9 Activity constructors
39 4.10 A brief introduction to scope and closure
41 4.11 The bank account example
43 4.12 Factories
45 4.13 Making lists
48 4.14 Making list factories
48 4.15 Terminations
49 4.15.1 Handling terminations
52 4.15.2 Using terminations to generate recursion
53 4.16 Embedded code
53 4.17 Names and scopes
54 4.17.1 Introduction of terms
55 4.17.2 Name spaces
55 4.17.3 Bindings
57 4.17.4 Scopes and Ranges
61 4.17.5 Extents
61 4.18 The type model
61 4.18.1 Informal statement of type conformance
62 4.18.2 Full type conformance rules

65 5 DPL library
65 5.1 Basic types
65 5.1.1 Integer
67 5.1.2 String
67 5.1.3 Literals
68 5.1.4 OutputStream
69 5.2 Library types
69 5.2.1 Common Operations
69 5.2.2 The type constructor protocol
70 5.2.3 The list, set and stack types
71 5.2.4 The Mapping Abstract Supertype

75 A Formal DPL Syntax
75 A.1 Notation
75 A.1.1 Meta-syntax
75 A.1.2 Lexical Meta-Syntax
75 A.2 Lexical syntax
76 A.2.1 Separators
76 A.2.2 Lexical tokens
76 A.2.3 Embedded blocks
76 A.2.4 String literals
77 A.3 Language syntax
77 A.3.1 Terminal symbols
77 A.3.2 Reserved words
78 A.3.3 Semantic Indicators
78 A.3.4 Syntax rules

1 Overview

Purpose

This document describes the ANSA view of distributed applications programming and how to use the ANSA distributed programming language DPL.

Audience

The intended audience are application programmers who are familiar with high level programming languages and commercial operation systems used in open systems.

Stability

The bulk of the DPL language is regarded as stable. The main areas of uncertainty are the type system, attributes and engineering terminations.

Related documents

APM.1001: *The ANSA Computational Model*

DPL was designed as a concrete syntax for writing programs that conform to these abstract semantics.

APM.1004: *The ANSA Atomic Activity Model and Infrastructure*

This is an architectural framework for the execution of concurrent distributed computations which maintain and preserve the integrity of distributed object state in the face of failures. As such it places extra requirements on the computational model. DPL does not yet take all of these requirements into account.

APM.1015: *DPL Reference Manual*

This defines the syntax of the DPL and gives an informal definition of its semantics.

RC.339: *Revising the DPL Type System*

This discusses the shortcomings of the DPL type system and describes the proposed solutions.

APM.1020: *Abstract and Automate*

This presents the arguments that constrained the semantics of DPL.

2 Introduction

This manual is about programming distributed applications of ANSA.

ANSA is an architecture supporting the design and construction of distributed systems that operate as a unified whole, so that their distribution is transparent to application programmers and users. It comprises a set of components, specifications and rules, complemented by recipes, guidelines and examples, which are used to create systems in a consistent, coherent and open manner.

The ANSA approach is fundamentally different to simply networking single systems together. It allows full advantage to be taken of the inherent concurrency and separation of distributed systems, in order to increase performance, decentralisation and reliability, while better masking their disadvantages: communication errors and partial failures. It produces systems that can be managed as co-ordinated sub-systems appropriate to the enterprises they serve, rather than as a random collection of boxes.

These first two chapters focus on architectural concerns and the general principles of writing distributed applications in ANSA; the remainder of the manual describes how to use the DPL preprocessor and ANSAware tools for writing distributed programs in C.

2.1 The problem space

When building a distributed system, a number of assumptions which are commonly made when constructing systems for single hosts are not merely invalidated, but must be reversed. The most important of these are:

- *local → remote*
  Because of the possibility of communications failure, remote interactions in a distributed system have more failure modes than local ones. A program component which assumes that all other components with which it is interacting are remote (and hence subject to unpredictable failures) will cope with any failures that do occur, whereas a system which implicitly assumes that all components are local will not.
- *direct → indirect binding*
  Continuously-running distributed systems with large numbers of nodes must be reconfigured or upgraded dynamically. This requires support for binding at execution time.
- *sequential → concurrent execution*
  Sequential execution (with its implicit synchronisation) is the default in single-processor, non-distributed systems, and concurrency must be generated explicitly when needed. The multiple processors in a distributed system inevitably provide concurrency, and synchronisation must be engineered where sequencing of actions is required.

- *synchronous → asynchronous interaction*
  Interactions between components of non-distributed systems are synchronous with small communication delays. Interactions between distributed system components are naturally asynchronous, and can suffer from large delays.
- *homogeneous → heterogeneous environment*
  Non-distributed programs usually only have one representation for any particular piece of data; distributed programs must cope with multiple simultaneous representations of the same data.
- *single instance → replicated group*
  Replication of system components which are then distributed over a system can be used to increase its availability and/or dependability.
- *fixed location → migration*
  The locations of distributed system components may not be permanent; for example, a service provided in one location may be transferred to another after hardware failure.
- *global name space → federated name spaces*
  Non-distributed systems usually fall entirely within one administrative domain. Distributed systems often span administrative boundaries, and hence need flexible naming schemes that reflect this.
- *shared memory → disjoint memory*
  Shared memory mechanisms cannot operate successfully on a large scale.

ANSA is designed to cater for interactions between distributed system components that are remote from each other; given sufficient knowledge, particular optimizations can be engineered back when the full generality of the remote interaction is not required. In this way the architecture copes with distribution without requiring a diversity of separate programming interfaces to cater for local and remote interactions, and without sacrificing execution efficiency.

2.2 Basic concepts

The structuring concepts provided by ANSA for application programmers to use when designing distributed programs are collectively termed the ANSA Computational Model. The major concepts in this model are:

- *service:* an abstraction of data and the programs that operate upon them
- *operation:* a service primitive
- *object:* an entity that encapsulates the data and programs of one or more services
- *interface:* an object's service provision point
- *invocation:* the mechanism by which operations are executed
- *interface type:* a description of an interface in terms of the set of operations that it can perform
- *activity:* the thread of execution that makes a sequence of operation invocations
- *type conformance:* the relation between any pair of interface types which determines whether one can legally be used when the other was requested

(Note that in ANSA the term "object" has a slightly different connotation to its use in object-oriented languages - see §3.3.1)

2.3 ANSAware and DPL

ANSAware is a software package providing an infrastructure for writing distributed systems, based upon the ANSA architecture. DPL is a notation for defining the interfaces between components of a distributed system implemented using ANSA. It is a simple language which embodies all the concepts needed for distribution, and which can be used with conventional programming languages to provide them with the capability to support distributed applications programming.

The design goals of DPL are to provide the means to:

- write distributed programs independently of particular computer systems and networks
- logically partition system components into distributable objects
- separate control over logical and physical modularity
- exercise control over the granularity of distribution
- provide a minimum set of additional constructs for the distribution of programs written using conventional programming languages
- provide, where possible, declarative control of mappings of application programs to ANSA engineering mechanisms
- encapsulate existing programming languages
- check types at compile-time
- dynamically bind application components together in a type-safe manner
- transparently distribute:
  - naming and binding
  - invocation mechanisms
  - parameter passing
- provide support for selective transparencies, thus allowing the bulk of ANSA's engineering model (including the transparency mechanisms themselves) to be implemented using DPL

Because ANSA is an architecture, there can be many different implementations, each conforming to the specification laid down by the architecture. ANSAware and DPL are examples of implementations of parts of ANSA, and do not represent a conformance test against which other implementations should be measured.

The current version of ANSAware is implemented in C, and the DPL translator produces C code as its output.

2.4 The manual

Chapter 1 of the manual provides an introduction to the problems associated with distributed processing.

Chapter 2 provides an overview of the principles of programming distributed systems in ANSA.

Chapter 3 describes the facilities of DPL in detail.

Chapter 4 describes the DPL library.

Appendix A gives a summary of DPL syntax.

3 A distributed applications primer

This chapter discusses the principles of distributed application programming in ANSA.

ANSA provides a small, complete set of primitives that can be used to structure distributed applications. The form of these primitives is dictated by ANSA architectural principles, which in turn are derived from the requirements of distributed systems, coupled with an architectural requirement that separate concerns should be separately addressed.

Statements of the ANSA architectural principles are scattered throughout the text of this chapter.

3.1 Abstraction and programming models

A primary aim of ANSA is to enable programmers to construct distributed applications without having to take account of the potential diversity of hardware, operating systems and communications mechanisms in the underlying computer network. The ANSA computational model also hides the actual degree of distribution of an application from its programmer, thereby ensuring that application programs contain no deep-seated assumptions about which of their components are co-located and which are separated. Because of this, the configuration and degree of distribution of the hardware on which ANSA applications are run can easily be altered without having a major impact on application software.

This desirable characteristic is called *distribution transparency*.

Ideally, this goal could be achieved by taking some programming system based on an existing language (such as C) and somehow re-engineering it (for example, by changing the libraries or writing a new compiler) to allow existing programs to be distributed across a network without modification. Unfortunately this is impossible: C (like Pascal, Algol and other comparable languages) is designed upon deeply-ingrained single-machine assumptions. It is for this reason that ANSA uses a new programming model for constructing distributed systems; the ANSA computational model.

The ANSA computational model does not, however, preclude the use of software for centralised systems in a distributed environment, but allows the encapsulation of existing applications as (non-distributed) components of a larger, distributed application. This permits an evolutionary approach to the provision of distribution, thereby protecting investment in existing software.

3.1.1 The shared memory problem

The obstacle to truly distributed versions of procedure-oriented languages is the way in which they separate programs and data. Languages like C, Pascal, Algol and Ada use arrays and records to build composite data structures, with procedures to manipulate those data. Many recent languages that innovate in

other areas (such as logic and functional programming languages) nevertheless continue to use this separation of data from the code that operates upon them.

Separating code and data in this way necessitates rapid transmission of small units of information between the two; in a single computer this is easily achieved by making procedures directly manipulate the data stored in memory. However this environment cannot feasibly be provided should the code and its associated data be located on separate nodes in a large network.

Principle: Large scale distributed shared memory is not practical.

The assumption implicit in the use of shared memory is that complex data structures can be represented using a “lowest common denominator” representation - typically a 32-bit word or an 8-bit byte. Procedures interact with data structures by fetching, manipulating and storing data at this fine grain. Attempts to provide distributed implementations of this mechanism encounter problems that stem from the overhead and latency implicit in communication.

The communication overhead springs from the need to associate routing and error-checking information with any data shipped over a network. This overhead is usually independent of the quantity of data concerned, and is often several tens of bytes per data item. Hence using a computer network to transmit data as individual bytes or words would be very wasteful of available bandwidth.

Although it is always possible to provide more communications bandwidth, the second problem, latency, cannot be circumvented. No matter how high the bandwidth of a communications link, information cannot travel along it faster than the speed of light. Using local storage it might typically take about a hundred nanoseconds to read, manipulate and store a word, and so a procedure can sequentially execute some tens of millions of such operations per second. In a distributed shared memory system the data and the procedure might be separated by (say) 100km. This means that the delay between a procedure requesting and receiving a piece of data could not be less than one millisecond, allowing an ordinary sequential procedure to perform only about 1,000 sequential manipulations per second, some four orders of magnitude fewer than in the local case.

It is important to stress that these performance limitations of distributed shared memory are not technological, but stem from basic physics. As computers get faster the disparity between the speed of access to local and remote memory will get larger, not smaller. Local caching can only help if data is written much less frequently than it is read, since protocols for maintaining the consistency of a number of cached copies of the same data are subject to the same effects.

This granularity problem may be circumvented by tailoring the unit of data interchange for each particular application, rather than attempting to use the same, lowest-common-denominator unit (be it a byte, word or whatever) for every application. Although this will not reduce the delay between requesting and receiving remote information, it does mean that useful quantities will be moved in a single round trip. The ANSA computational model is built on this approach; data and the operations that manipulate them are grouped together to provide a *service* that processes information in chunks of a size appropriate to the job at hand, rather than as individual words. Because communication delays often greatly exceed the time required to process information, a single

service request can return a useful quantity of data in about the same time that a distributed shared memory can retrieve a single word. This is a major reason why ANSA has its own service-based computational model rather than adopting an existing model implicitly based on the assumption of shared memory.

3.1.2 Partial failure in distributed systems

A model of computation based on services has other advantages for distributed systems. For instance, it can provide control over the visibility of partial failure.

Principle: The possibility of partial failure is implicit in distributed computation.

If a single computer supporting a non-distributed application fails then all parts of the computation stop together - there is no need to cater for the possibility that (for example) a procedure can continue to execute after its data ceases to be available. On the other hand, where there is a network of inter-linked but independent computers, it is possible that a computation on one node might discover that some failure had isolated it from a cooperating activity at another node.

Partial failure in distributed systems cannot be ignored. Although the infrastructure can mask transient failures such as noisy communications lines, hard failures (for example, the crash of a node or a total loss of communication) must be reported back to the client application so that it can take steps to continue or fail gracefully as appropriate. For this reason failure is made explicit as a possible outcome from the use of an ANSA service. An application can specify what action to take in the case of a service's failure, or it can elect to ignore the possibility, in which case an unrecoverable service failure will cause the application to fail as well, thus propagating the failure to any clients of services provided by the application. In this way hard failures are always either dealt with or propagated; a user of a service cannot overlook its failure and continue to run with no indication that something went wrong.

3.2 Anatomy of a service

The component parts of a service are its *operations.*

Consider a bank account as a service. Associated with the account is some state, such as a current balance and the list of standing orders that are paid out every month. The bank dictates the set of actions that can be performed on this state; for example, creating a new standing order or debiting the balance by writing a cheque. These actions are the primitives for manipulating a bank account; the transactions that appear on the customer's monthly statement are drawn from this restricted set of possibilities. In ANSA they are represented as operations.

The ways that a user can interact with a service are completely defined by the set of operations that the service supports. For the sake of this explanation, consider a simplified version of a bank account service that only supports integer balances and three possible actions: debit or credit the account, and find the current balance. In ANSA the specification of each operation has two parts: an *operation signature* which is written in DPL and defines how the operation is invoked by a client, and the *operation body*, which is the piece of program code executed when that operation is invoked.

The operation signature in turn has a number of well-defined components. Although the syntax of DPL will not be introduced until the next chapter, it is convenient to use the syntax of operation signatures here in order to describe this banking service example.

An operation signature has three parts:

- The *operation name* is an intrinsic part of the operation: when the client wishes to invoke an operation in a particular service it identifies it by its name within that service. To ensure that there is no ambiguity, no two operations in the same service may have the same name.
- The *parameter part* of an operation specifies the number and types of the parameters that are passed to the operation when it is invoked.
- The *result part* of an operation specifies the number and types of result for each possible outcome from the operation.

In DPL notation the signature of the credit operation on the bank account service would be:

```
Credit(x:Integer) ->()
```

This states that the operation named `Credit` takes a single parameter (the integer amount to be credited to the account) and returns no results. There is no parameter to say which account to credit; the operation is performed *on* the appropriate instance of the bank account service. The DPL syntax for using the `Credit` operation to increase the balance of account `MyAccount` by 17 units would be:

```
MyAccount.Credit(17)
```

The Debit operation is similar to `Credit`:

```
Debit(x:Integer) ->() ->InsufficientFunds()
```

This signature indicates that the `Debit` operation has two possible outcomes: the normal one, and one which indicates a lack of funds in the account to cover the requested withdrawal.

Principle: Operations have distinct outcomes, each of which can convey different numbers and types of results

Although this principle applies to operations on many kinds of data, few programming systems support it. In ANSA an operation's possible outcomes are called *terminations*, and are distinguished by their names (such as `InsufficientFunds`). For convenience one outcome from each operation can be left unnamed; this is called the anonymous termination, and is conventionally used to represent the normal or expected outcome, while named terminations are often used to represent unusual or unexpected results.

As discussed in §3.1.2, failure is a possible outcome from any attempt to invoke an operation on a service, since any service may (in principle) be remote from the invoker, and any remote invocation may fail. This failure is represented as a termination, which, since it may result from any invocation, is automatically written into the signature of all operations by the translator. This allows the programmer to detect and handle invocation failures if desired.

Note: The name and signature of this engineering failure termination has not been finalised.

The third operation on this example bank account service provides information about the state of the account:

```
List() ->(Integer String)
```

In this case there are no parameters, and the single possible outcome is the anonymous termination delivering two results: the account balance and a string giving the time of last access to the account. In fact an operation can be defined to return any number of results in any or all of its terminations, as well as taking any number of parameters as arguments.

3.3 Objects: controlling co-location of services

All the operations in one service have access to the data associated with the service (such as the account balance in the example in the last section). On occasion it may also be desirable to share state between different services, or many instances of the same service. As an example, consider implementing a bidirectional queue of integers as a pair of services. Each end of the queue would be represented as a service supporting operations to push and pop integers:

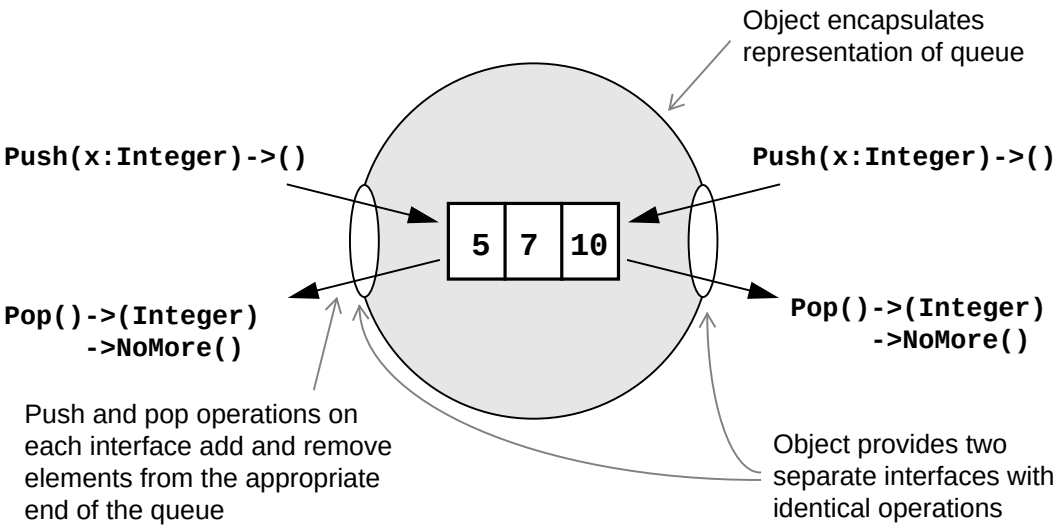
```
Push(x:Integer) ->()

Pop() ->(Integer) ->NoMore()
```

If two services are to share a single queue they must both have access to a common representation of that queue. This is arranged in ANSA by allowing service provision points (known as *interfaces*) and their associated data to be grouped together into *objects*.

Principle: Encapsulation and service provision are separate concerns

The bidirectional queue can be modelled as an object that presents two interfaces with identical operations, each providing access to one end of the shared queue implemented within the object. Figure 3.1 illustrates this.

Figure 3.1: The bidirectional queue object



The object provides an encapsulation boundary within which single-machine design assumptions apply; programs running inside the same object can share memory and internal data representations and can assume that there will be no communication failure between them, and in the event of external failure they will all terminate simultaneously. Hence an existing, single-machine application can be integrated into a distributed system simply by enclosing it in an object, and providing interfaces that permit the (potentially remote) clients of the application to access its services. Of course, in this case the application has not itself become distributed, since the object that it inhabits cannot be distributed across more than one address space.

It is important to stress that objects exist solely to encapsulate the state behind one or more interfaces. It is not possible for the application programmer to refer directly to the object, nor to manipulate the state that it holds other than by using the operations in its interfaces. ANSA does not even allow the user of several services to determine whether they are provided by the same object; this is so as to prevent a client program which uses a number of services from making assumptions about the structure of the objects that provide those services. Such assumptions are likely to be invalidated as the implementation of a distributed system evolves.

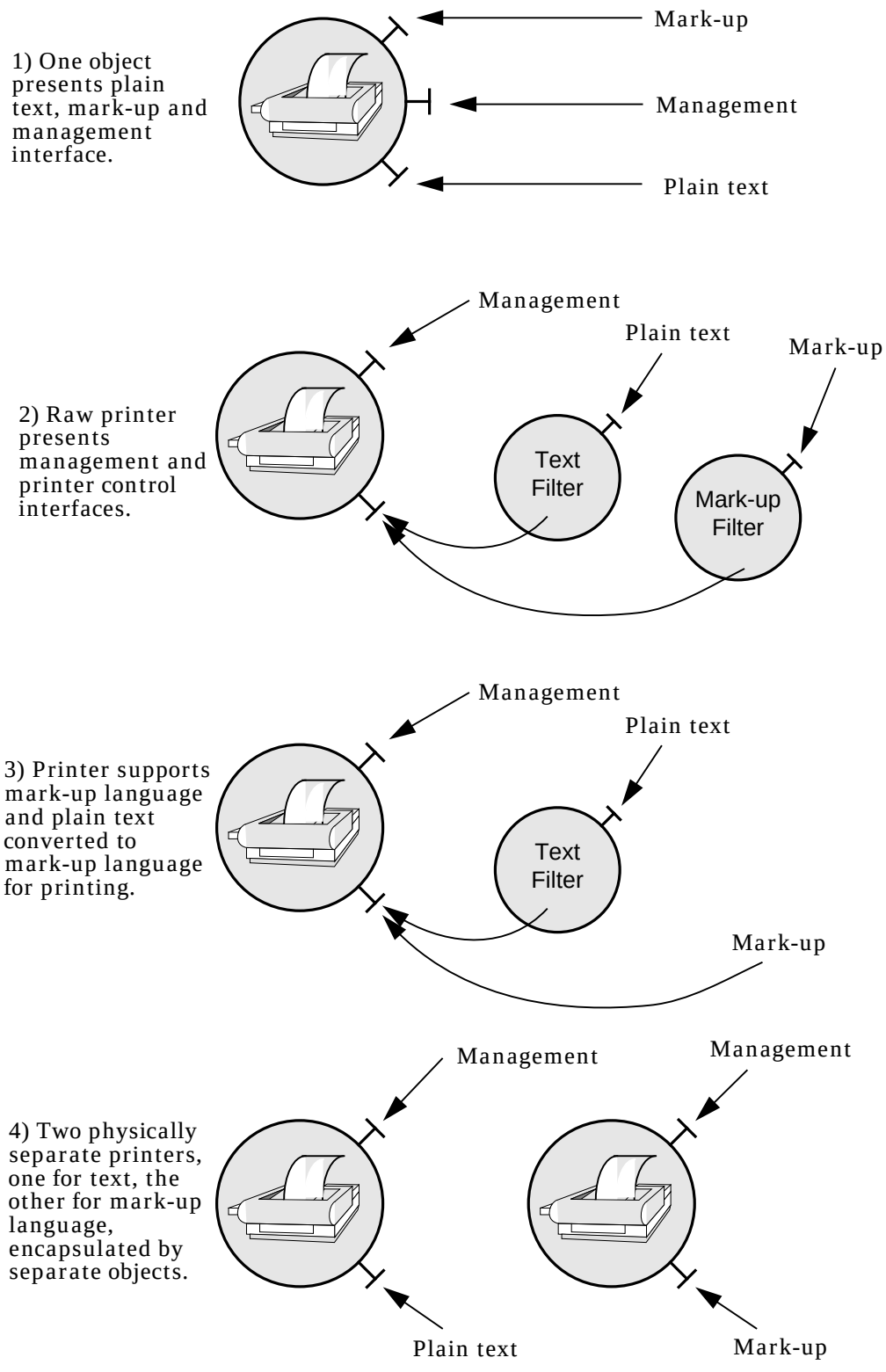
Figure 3.2 provides a specific example to illustrate this point. It shows how three logically-related services (two for printing and one for print service administration) might be provided using any one of four different object configurations. Only in the first configuration is the management service provided by the same object as both print services. If the computational model incorporated a mechanism to allow clients to determine whether two services were provided by the same object then there would be a danger that a programmer might incorporate the assumption of a particular configuration into a client program, which would then not work with any other configuration. Because the relationships between services (such as which management service is responsible for a particular print service) is independent of the configuration of objects used to provide them, the only reliable way to determine the relationships is to build appropriate operations into the services themselves.

3.3.1 Relationship to object oriented languages

Those who have encountered object-oriented programming languages (OOLs) will notice that they have much in common with ANSA; similar requirements for encapsulation have independently led to similar designs. One difference is that OOLs use a single mechanism for both service provision and encapsulation, whereas ANSA has separated these functions. This means that (for example) OOLs cannot model the bidirectional queue example using a single OOL object, since it would be incapable of simultaneously providing two different instances of the same service.

The other major difference between the ANSA model and OOLs is that ANSA makes no mention of the implementation inheritance mechanism that provides code reuse in many OOLs. Inheritance comes in two forms: *reactive* and *non-reactive*. Reactive inheritance uses the assumed presence of shared memory to allow a changed definition to be propagated automatically to all inherited copies, whereas in a non-reactive system, definitions are copied when inherited and don't subsequently change. Because of the shared memory assumption, reactive inheritance cannot be distributed. Non-reactive inheritance poses no such problems, but since it is a language design issue

Figure 3.2: Possible structures for printing services



which affects how programs are written rather than the way they are structured, it does not have any impact on the computational model.

3.4 Characteristics of services

3.4.1 First Class status

Some important principles follow from the status of the service as the basic means of representation in ANSA systems.

Principle: Services should be dynamic entities, created and destroyed at will.

It must be possible to create new services easily when a new instance of an abstraction is required. In the bank account example creating a new account requires the creation of a new service to represent it. Equally, when an account is closed, the now-redundant service must be deleted.

Principle: It should be possible to pass the ability to use a service between clients.

The reasoning here is the same: in order to use services to represent information, it must be possible to pass the use of a service from one point in the system to another. In ANSA the ability to use a service is conferred by holding a reference to it - the original reference is manufactured when the service provider is created, and can be copied and distributed to many clients by being passed as a parameter to, or returned as a result from, an operation.

3.4.2 Service transparencies

Distribution transparency was introduced in §3.1 as a desirable property that gives distributed programs independence from the details of their support environment. However there are some occasions when transparency is less desirable. For instance, the programmer who wishes to implement a resilient service by distributing the individual components over a network will want to exert enough control over the locations of the services to ensure that they are not all to be found on the same machine.

Principle: Distribution transparency should be under the control of the programmer.

In order to provide this sort of fine-grained control, ANSA subdivides distribution transparency into a number of specific transparencies which the programmer can control individually if desired. This control is known as *selective transparency*. Among others, ANSA provides:

- *Access transparency*, which hides differences in data representation between client and server, so that a client can use a service located on a machine with a foreign data format without needing to detect and cater for this situation. Access transparency is fundamental to writing distributed programs using ANSA, and few users will wish to disable it.
- *Location transparency*, which makes the location of a service invisible to the client, and the manner of use of local and remote services indistinguishable. Optimisations of services which are found to be co-located with their clients can be made by support tools (such as the DPL preprocessor) without altering the program code at all.
- *Migration transparency*, a dynamic form of location transparency, which hides the effect of a service moving from one location to another, even if a client is using it at the time.

- *Replication transparency*, which hides the effects of providing a single service using a number of duplicated interfaces working in concert. Where the interfaces are supported by separate objects in separate locations this can be used to increase the service's resilience in the face of individual failures.
- *Concurrency transparency*, to prevent multiple simultaneous users of a service from suffering any interference from each other.

Another motivation for making transparencies selective springs from the observation that most of them can be implemented independently. For example, a mechanism for replication transparency could use an access- and location-transparent invocation mechanism to make each of the replicated invocations. This independence allows some transparency mechanisms to be implemented as source-to-source transformations on DPL programs. Each transformer implements a transparency by taking a DPL program that assumes the existence of that transparency and producing one that achieves the same effect in its absence. In this way the larger part of the distribution transparency provided to the DPL programmer can be implemented by DPL code.

3.4.3 Use of service references

Service references in ANSA are completely opaque; the only actions that the application program may perform on a service reference are to copy it (by duplicating it or passing it as an argument to or result from an operation), or use it to invoke a service's operations.

Principle: Service references can only be used to invoke service operations.

Many prospective users are at first puzzled by the absence of a user-level identity test that can be used to establish if a pair of service references refer to the "same" service without actually invoking either of them. There are two fundamental problems with providing this sort of facility; one practical and one philosophical.

The philosophical question concerns what is meant by “same”. This is rather a slippery word, and there are several possible definitions that we could apply in different circumstances; in particular we need to distinguish clearly between equality (“sameness”) of abstraction and equality of implementation. To see the difference, consider a pair of computers each providing a very simple service which calculates the area of a circle from its radius. So far as the user is concerned these two services are indistinguishable; a reference to one may be substituted for a reference to the other in any program without affecting its correctness. To put it another way, the services represent the same abstraction. On the other hand, the implementations of the two services may well differ - the two computers could have different architectures and use different floating-point number formats to represent the numbers involved. The two programs could have been compiled by different compilers from different source programs. The purpose of the service transparency mechanisms discussed in §3.4.2 is precisely to hide these incidental differences, making the two services indistinguishable from the user's point of view. By this token the two services are the same, yet it would be impossible to arrange that a comparison of the service references could reveal this, since it would involve examining the behaviour of the service providers, rather than the structure of the service references. A service reference comparison would

simply tell us that these otherwise-identical services differ in some way that is not visible to the service user. This would not be particularly helpful.

If the philosophical problem lies in trying to make a service reference test to indicate when a pair of services will exhibit the same behaviour, the practical problem is that it is very difficult to implement a test that will always provide even the (apparently trivial) ability to tell whether one reference is a copy of the other, or that they share a common ancestor in a tree of copies. This stems from the concurrency and chains of indirection present in distributed systems.

To see this, consider the example in figure 3.3 , where migration transparency can prevent a client from detecting that it holds two references to the same service, even when one reference was originally a copy of the other. The migration transparency mechanism outlined in this example is purely illustrative, and is not necessarily the one employed by any particular ANSA implementation; however it does demonstrate how the use of indirection (which is unavoidable in the implementation of distributed systems) masks service reference equality.

Hence, in the cases when we cannot say with certainty that the two references are related by copying, we cannot assert that they are certainly not related - it may simply be that the various optimisations and implementation techniques have hidden the relationship. The best answer that can be given to the “same reference” question is either “yes” or “maybe”.

Although ANSA does not provide a test to determine whether two service references denote the same piece of implementation, it certainly does not preclude tests to discover if two services denote the same abstraction. This will typically be implemented by an operation (e.g. `"equal"`) on a service that takes as a parameter a reference to another service and returns a Boolean result to say if the two represent equal abstractions. Such a test must in turn be implemented using equality tests on more primitive data types. Within the ANSA computational model the most primitive such test is the ability to distinguish terminations by their distinct names; the caller of an operation can detect which of the possible named terminations the callee has returned, and act accordingly.

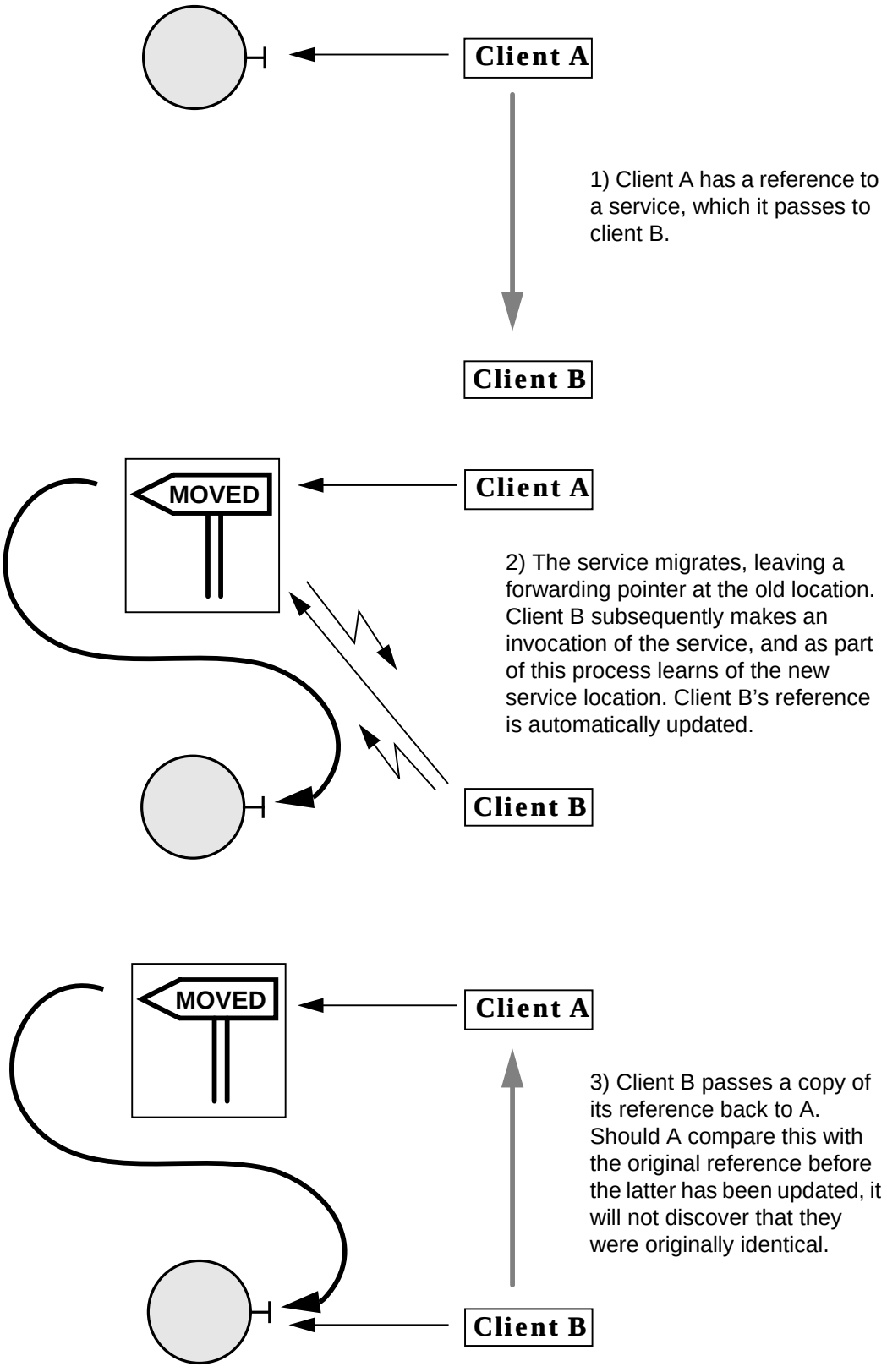
3.5 Configuring distributed systems

3.5.1 Dynamic Binding: support for reconfiguration

In building applications, whether of the distributed or non-distributed variety, *binding* refers to the action of making one component of the application accessible from another. For example, when a C programmer writes a procedure call in a program he assumes that the desired procedure will have been bound to the appropriate name by the time the program is executed. C is an example of a *statically-bound* language, where all bindings (of procedures) are resolved before the program is run. In fact the utility that does this is sometimes referred to as a “binder”, although it is more commonly called a “linker”.

Should any change need to be made to the configuration of a statically-bound application (such as substituting an improved version of some procedure), the entire application must be re-linked and then re-executed, replacing the running version and probably causing some disruption.

Figure 3.3: The problem with service reference equality



The alternative style of binding, used in interactive programming languages such as Lisp and interpreted BASIC, is *dynamic binding*. This technique permits bindings to be altered while a program is running, allowing the configuration to be changed in response to changing circumstances. Although this is useful for prototyping, debugging and exploratory programming, it suffers from a potential loss of efficiency, and of safety compared to static binding. While static binding allows static type checking, where the correct use of bound components can be checked at compile-time, dynamically bound systems are typically dynamically-typed. Since bindings can be changed at arbitrary times, every use of a binding is accompanied by a check to see that the bound component will still support the intended use. It is this type-checking overhead that is partly responsible for the reputation for inefficiency held by dynamically-bound languages. Worse, the failure of a run-time check, often resulting from a programming error, is another potential source of failure in an application.

The same distinction between static and dynamic binding applies to distributed applications. If the relationships between components of the application never change then service references can be compiled into applications to give static configurations. As with non-distributed applications, changing such a configuration requires the application to be aborted and restarted. However, this procedure becomes less and less practical as the degree of distribution of the application increases.

Principle: Large-scale distributed systems do not have fixed configurations, and cannot be stopped and started as a single unit.

Although ANSA can be used to construct statically-configured and statically-bound applications, it has a mechanism for efficient dynamic binding of clients to services which supports the reconfiguration necessary to support highly-distributed, long-lived applications. This mechanism is based upon performing type compatibility checks at bind-time, rather than invocation time, so that invocation of a service can proceed with the same safety that is achievable in a statically-bound system.

3.5.2 Trading: support for service reuse

Large applications, whether distributed or not, invariably have many common support requirements. To avoid having repeatedly to re-implement support functions, programmers use (and reuse) libraries of system components. Some libraries are used by almost all applications (such as I/O libraries), while others are more esoteric, being used by smaller groups of users (for example, graph plotting routines).

While it is possible to use a statically-linked library mechanism in building distributed applications, dynamic binding opens up the possibility of replacing compile-time libraries by generally-useful services to which applications can bind at run-time to perform particular, widely-needed tasks.

To allow clients to locate these services, many distributed system architectures provide so-called *name services*, which map the names of services onto addresses, allowing prospective clients to look up their desired service by name. ANSA supports an extension of this idea, a *trading service*, based on the observation that the client usually wants a service that performs a specified function rather than one with a particular name. For a useful analogy, consider searching for a service using the telephone directory; the Yellow Pages allow you to look up a service by its function (e.g. “plumber”),

while the ordinary directory is only useful if you know the name of the service provider (“John Smith”).

A trading service keeps a database containing offers of services of interest to particular communities (such as those who require a printing service, or a password authentication service). Clients wishing to use a service contact the trader, giving it a specification of what they want. This specification is the *service type* of the desired service.

In order to understand the way that a trading service operates, it is necessary to explain the use of types in ANSA.

3.6 Types: describing the way a service is used

Types in programming systems are generally used for two purposes:

- To describe how values can be used. For example, the C integers support the ++ operation (increment), but not the * operation (pointer dereference), while pointers support both.
- To express classification ("is-a") relationships. Type systems are sometimes used as knowledge classification systems to encode beliefs about the relationships between different classes of entity. For example, making Truck a subtype of Vehicle encodes the belief that all trucks are vehicles, but not vice-versa.

These two rôles for types are both important, but in keeping with its philosophy of separation of concerns, ANSA handles them in different ways: with *interface types* and *service types*.

3.6.1 Interface types

The interface type contains information on conducting structurally correct (although not necessarily meaningful) interaction with a service; it gives the potential client information about the mechanics of using the interface:

- The names of the operations supported
- The number and interface types of the arguments to any given operation
- The names of all the possible terminations that might result from invoking an operation
- The number and types of parameters associated with each termination.

In short, the interface type alone gives all the information necessary for a client to conduct a correct interaction with a service. It is mechanically checked at bind time to ensure that the client can only invoke legal operations on the service interface, passing parameters of compatible types. It does *not* attempt to say anything about what actions operations perform, nor does it hold any information about what the service represents - this is the job of the service type.

Interface type checking in ANSA is done not by exact matching, but by a *conformance* check. Informally stated, type A conforms to type B if a service could safely provide an interface of type A where its clients expected one of type B. This is precisely the condition that is required for in-service upgrade of a service - an administrator can replace a server that has a type B interface with one with a type A interface, with conformance providing the guarantee that all interactions between the client and the server will be correct. In

particular, the conformance relation allows the new service to provide an enlarged set of operations, which can be used by new clients without the original clients being disrupted.

3.6.2 Service types

Service type is a broader, but less precise, concept that reflects the semantic relationships between services in a distributed system.

Many languages (such as C++ and Common Lisp) support the creation of explicit subtype relationships, allowing programmers to state which types are subtypes of others, and thus express semantic relationships between them. Unfortunately, this explicit-subtype approach has two problems, one particular to distributed systems, and one more general.

- *Separate development*. Explicit subtype systems require all subtypes to be defined in relation to, and inserted into, a single universal type graph. This causes problems if different development teams make incompatible changes to separate type graphs which must subsequently be merged. It also makes it very difficult to discard a type if other types have been defined in relation to it.
- *Insufficient expressive power*. The only semantic information that an explicit subtype relationship can convey is the "is-a" classification. This is insufficient to convey ideas such as the cost per page of a printing service. Such information is very application-specific, and it is in fact rather difficult to construct *any* framework capable of expressing all possible kinds of semantic information about any particular service in a form that can be mechanically checked.

For these reasons ANSA does not use explicit subtyping to express the relationships between services. Indeed, because the format of the relevant information (and the criteria used for judging an offer's suitability to fulfil a request) vary from application to application, ANSA does not attempt to formalise service type information at all, but instead leaves the task of storing and manipulating service types to the trading service.

3.6.3 Types and trading

Service types need only be explicitly represented when a prospective client is looking for a service with some particular set of semantics; in other words, when trading. Since the trading service is a user-level application, rather than a low level facility in ANSA's support infrastructure, it can be tailored to support a specific concept of service type if desired. However, the general-purpose trading service provided with ANSAware has been used with many applications - it associates the following service type information with each offer:

- Interface type.
- Context - to which of a number of hierarchically arranged categories the service belongs. These categories are used to group related services, such as all those belonging to one administration, or services provided at a specific location.
- Properties - a collection of (name, value) pairs giving information about this service instance.

When requesting a service from a trading service, the prospective client supplies:

- The interface type to which the offered service must conform (in order to guarantee successful binding and correct interactions).
- A specification of a sub-tree of the context tree in which the selected offer must be registered.
- A constraint expression specifying what properties are acceptable.

The trading service selects an offer that satisfies these conditions (if one exists) and supplies it to the client. On receiving the offer the client can communicate directly with the service, and need not communicate with the trading service again until it needs to obtain a service with different properties.

3.7 Activities: handling concurrency and communication

Principle: In a distributed system there is inherent parallelism.

A distributed system, by virtue of having many processing sites, has inherent parallelism. The challenge for the system architect is to determine how best to map the parallelism requirements of multiple applications onto the parallelism that the system provides, and how to make the inherent parallelism available within a single application.

There are many possible mechanisms for communicating between parts of distributed applications, for controlling the concurrency between them and for synchronising separate components. ANSA's approach is based on the following principle:

Principle: Concurrency generation and communication are separate concerns, and should be separately addressed.

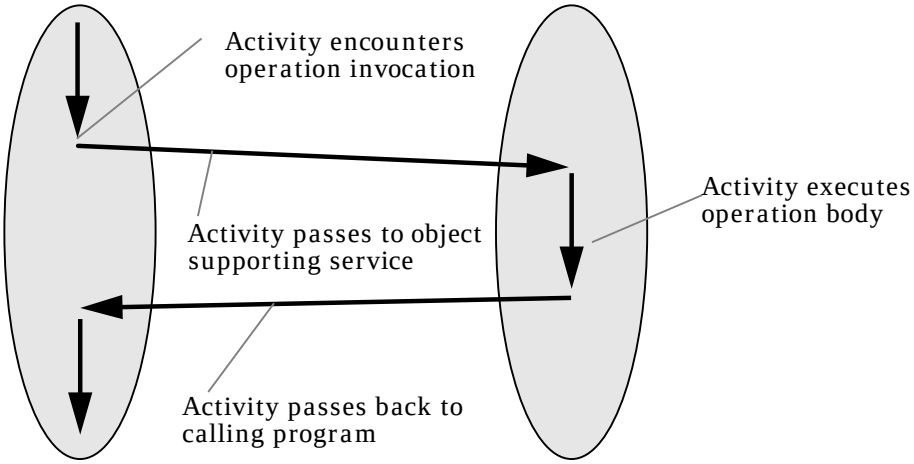
Application of this principle leads to a communication mechanism that is concurrency-preserving (i.e. neither creates nor destroys concurrency within or between objects), and a concurrency generation mechanism that does not duplicate the functions of the communication mechanism. In addition, distribution transparency requires that the communication and concurrency mechanisms operate independently of the degree of distribution of the application.

The ANSA realisation of these principles is centred around the *activity*. An activity is a unit of concurrency; a collection of ANSA objects may have any number of activities threading through them, of which one or more may actually be executing on a processor at any one instant, depending on the number of processors available. An activity encountering an invocation statement while executing the code of one ANSA object migrates to the object whose interface has been invoked. There it executes the operation's code, then returns to the calling object. This makes service invocation concurrency-preserving.

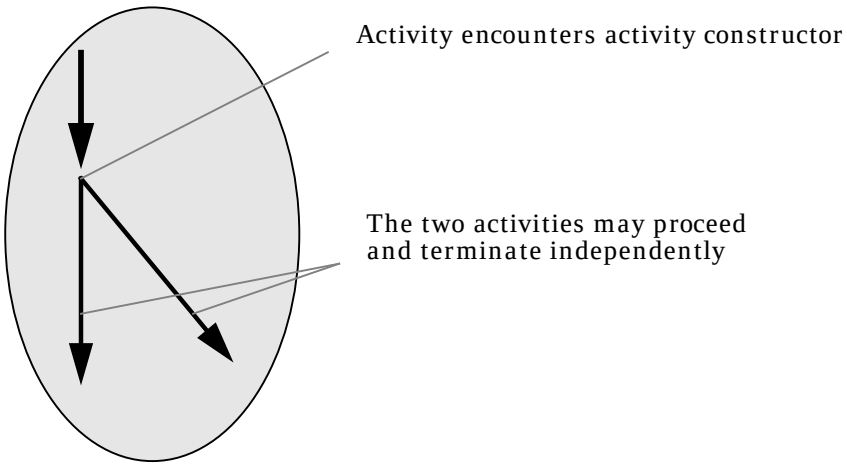
Concurrency generation in distributed applications comes in two varieties, which are illustrated in figure 3.4, along with the activity structure in an invocation. In the first variety an activity proceeds as several parallel subactivities, each of which perform some computation within the same object before they all merge once again. This is discussed further in the next section.

In the second variety, provided by the ANSA *activity* constructor, a new subsidiary activity is split off from the parent and both parent and child proceed, and finish, independently.

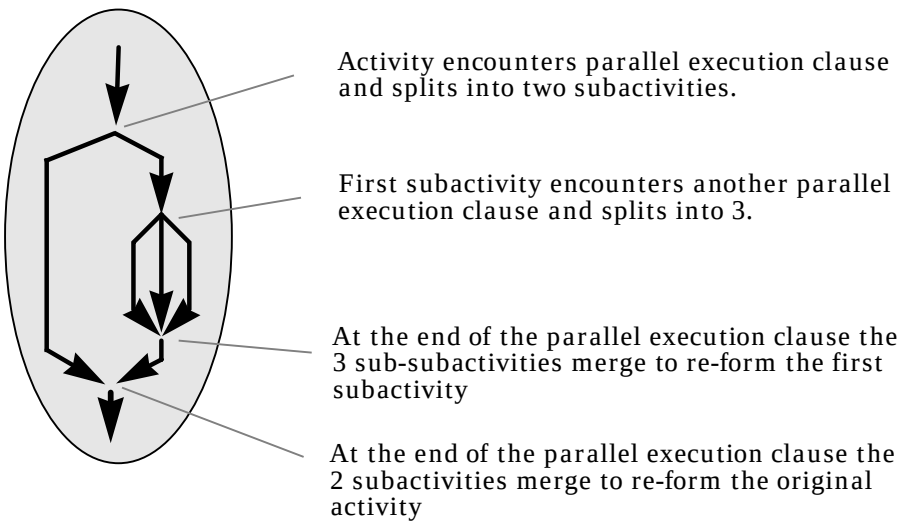
Figure 3.4: Activity structures



Activity structure in operation invocation



New Activity construction



Activity structure for parallel execution

The activity mechanism allows maximum use to be made of the inherent parallelism in distributed systems. Separate activities encountering an invocation of the same operation in the same service can all use the service, possibly simultaneously. The upper limit on the concurrency and throughput that can be achieved while several activities run the code of that operation is determined only by the processor(s) available at that node - and if improved hardware adds more processors, then the objects at that node will be able to use the increased parallelism without software changes.

Object state in ANSA systems was introduced in §3.3; to this must now be added the concept of activity state. Activities carry the state of their computation with them - when an activity passes into an operation it carries the parameters for that invocation, and returns carrying the results. It is this association of state with activities that provides concurrency transparency in ANSA, allowing several activities to execute the same operation at the same time.

The state that objects encapsulate is not owned by any one activity, but rather can be used by any activity executing an operation in an interface to that object. In fact there are two kinds of state that an object can encapsulate: per-object state and per-interface state. The former is potentially accessible to any operation in any interface to the object. The latter is attached to one interface, and is only accessible to operations in that interface; this allows multiple instances of the same interface type on one object each to keep their own state without having to have separate implementations.

State encapsulated by objects provides the means for communication between activities, but also opens up the possibility of independent activities causing conflicting object updates, violating concurrency transparency. To prevent this the implementor of the operation must be able to specify how activities should be separated when executing critical parts of operations. ANSA supports two main separation mechanisms: nested transactions and ordering predicates.

3.8 Separation mechanisms

A large distributed system will, almost by definition, simultaneously support multiple applications. Where these applications manipulate object state there is the possibility that concurrency transparency will be violated; one application could be caused to fail by encountering inconsistent object state left behind by another application, either because the two were accessing the state simultaneously, or because one suffered some catastrophic failure while manipulating that state.

Enforcing concurrency transparency and failure transparency between multiple applications is termed *separation* in ANSA, and the mechanism provided for achieving separation is the *transaction*.[1]

The ANSA transaction mechanism is based on a transformer technology, in which programming tools are used to automatically and safely transform the behaviour of non-transaction applications to transactions. These transformations are driven by parameters of an *atomic* attribute declared in the signatures of object specifications. The transformation technique is of

1. This is a complex subject, and only a brief overview is given here. For more details, consult APM/AR.004 *ANSA Atomic Activity Model and Infrastructure.*

sufficient generality to facilitate a wide range of transaction control policies and protocols.

With the transaction mechanism in place, all operation invocations on objects are executed by atomic activities which have the properties of (i) atomicity, (ii) consistency, (iii) independence, and (iv) durability. These four properties are collectively known as the ACID properties of transactions.

The first property - *atomicity* - ensures that the execution of an application can either be terminated normally (committed), producing the intended results, or can be terminated abnormally (reverted), producing no effect, as if the application had never executed.

The second property - *consistency* - ensures that the concurrent applications which access common objects are free of interference from each other (i.e., a concurrent execution of the applications can be shown to be equivalent to some serial order of execution).

The third property of transactions - *independence* - is essentially a refinement of the preceding two properties. Because transactions can either commit or revert, it is necessary to prevent concurrent transactions from sharing each others partial results until the final outcome of each transaction is known. Maintaining independence between concurrent transactions thus avoids the problem which would otherwise arise if one transaction were to use the partial results of another transaction which then became a subject of voluntary reversion or involuntary failure.

It is reasonable to assume that once the execution of an application has completed normally, the results produced are not destroyed by subsequent processor crashes. It is the fourth property of transactions - *durability* - which ensures this requirement by recording the committed results of each application on stable storage which can survive processor crashes with high probability.

Separation between independent applications is not the only area where concurrency needs to be controlled. The concurrency generation facilities of ANSA necessitate mechanisms for specifying the allowable concurrency between different parts of the same application. This is a less stringent requirement than concurrency transparency - although there may be several activities involved, they are all part of one application, and each part has been designed with knowledge of the other activities in the same application. Failure transparency constraints can also be relaxed, since it not necessary to guard against failure at every possible interaction between pairs of activities.

ANSA provides an *ordering* mechanism to allow the application programmer to control the concurrency within applications without the overhead entailed by using transactions. There are two aspects to ordering; declarations of potential concurrency between the evaluation of expressions within an activity (called source ordering) and declarations of allowable concurrency and ordering between the operations within one interface (called destination ordering). Source ordering is provided by the mechanisms for performing concurrent computations within one activity, while destination ordering declarations may be placed on the operations of an interface to control the order and concurrency with which they may be executed relative to one another, preventing patterns of operation invocation that would lead to inconsistent results.

3.9 Summary

- *Large scale distributed shared memory is not practical.*
- *The possibility of partial failure is implicit in distributed computation.*
- *Operations have distinct outcomes, each of which can convey different numbers and types of results.*
- *Encapsulation and service provision are separate concerns.*
- *Services should be dynamic entities, easily created and destroyed.*
- *It should be possible to pass the ability to use a service between clients.*
- *Distribution transparencies should be under the control of the programmer.*
- *Service references can only be used to invoke service operations.*
- *Large-scale distributed systems do not have fixed configurations, and cannot be stopped and started as a single unit.*
- *In a distributed system there is inherent parallelism.*
- *Concurrency generation and communication are separate concerns, and should be separately addressed.*

4 DPL Syntax and Semantics

The last chapter introduced the main issues in distributed programming, and presented the ANSA perspective on a programming system that takes account of them. This chapter will fill in this outline by introducing DPL, the specification language tailored for ANSA systems.

DPL is used to specify, configure and distribute embedded applications, and to implement transparencies. Since many existing applications are written in C or other programming languages, DPL provides a framework within which these applications can be used in a distributed system. Applications can also be written in DPL itself.

This chapter presents the basic kernel of DPL, explaining how the language expresses the ANSA model of computation introduced in the last chapter. The kernel is small and self-sufficient, and can be used on its own to provide examples of these concepts. The text is interspersed with tutorial examples to illustrate the points raised. Later sections show how to embed other language code within a DPL program, and give full details on scope rules, the type model, and the syntax of the language. Chapter 5 covers the type library and will introduce the use of syntactic sugars which make the language look more familiar.

Note: The syntactic sugars have not yet been implemented; the explanatory material will be added in due course.

4.1 Overall structure of a DPL program

The top level of a DPL program is an object constructor, which causes the DPL compiler to generate code to create a new object instance and execute the initialisation code it contains. In fact, this is all that a DPL program does, so all the work of the program is performed (either directly or indirectly) by this “initialisation” code.

Figure 4.1 is an example of a very simple DPL program which uses the facilities of the DPL library to print the message “Hello World”. Execution of the top-level expression `object(...)` causes a new object instance to be created and initialised by evaluating the expression list within it. The first expression in this list (i.e. `x=interface...`) creates a new instance of an interface to the object and binds it to the name `x`. This interface provides a single operation called `hello`, which takes no parameters and returns no results. When invoked, this operation will create a reference to a string of characters “Hello World” (terminated by a new line) and then immediately invoke the print operation provided by the string, causing it to print itself. The print operation takes a single parameter which specifies the stream on which this output will appear, in this case the value of the binding `output` from the DPL library, which will send the characters to a terminal or some other output medium in an implementation-dependent way.

Figure 4.1: Hello World

```
object          % the complete program object
 ( x = interface% make an interface bound to x
 ( hello() ->()% with operation called hello
      [
        (“Hello World\n”).print(output)
      ]
 )
 ; x.hello()    % invoke operation hello on x
 )              % end of object constructor
```

Interface constructors merely make interfaces - none of the code inside the constructor is executed until one of the interface's operations is actually invoked. In this example this happens in the second, and last, expression in the object constructor, which fetches the interface from the binding of `x` and invokes the `hello` operation on it. As indicated by the operation's declaration in the interface constructor, the invocation is made with no arguments, and will return no results. When the invocation finishes the flow of control leaves the object, and hence the program terminates.

Figure 4.2 gives a rather more complex example in which two objects are created, one of them bearing two interfaces (figure 4.3 shows the structure of objects and interfaces created by the program). These two interfaces have operations with the same names, but different behaviours; they represent the two boolean values true and false. Although small, this example illustrates many of the salient features of the language, and will be explained and used as a source of examples over the next few sections.

4.2 Interface lists and Expressions

Services are used as the basic means of abstraction in the ANSA computational model, and the ability to use services is passed between clients by passing references to the service provision points, which are called interfaces. It will therefore come as no surprise that interfaces and interface references have a central rôle in the DPL language, where every value is modelled as a reference to an interface providing an appropriate service.

In order conveniently to handle the sequences of interface references that make up argument and result lists, DPL uses *Interface lists*. Unlike interface references, these are not first-class entities in DPL; they cannot themselves be assigned to variables or passed as parameters. Instead, DPL expressions yield interface lists as their results, and assignment of names to interfaces is used to de-structure interface lists in order to pick out the individual interface references.

Most syntactic constructions in DPL are expressions. The evaluation of an expression yields an interface list. Probably the most important expression is the *Block*, which is used to group other expressions and provide close control over the construction of interface lists. Syntactically a block is just a list of expressions with separator characters between them, delimited by matching pairs of brackets.

Blocks come in two basic varieties, distinguished by their delimiting brackets:

Figure 4.2: The Boolean example

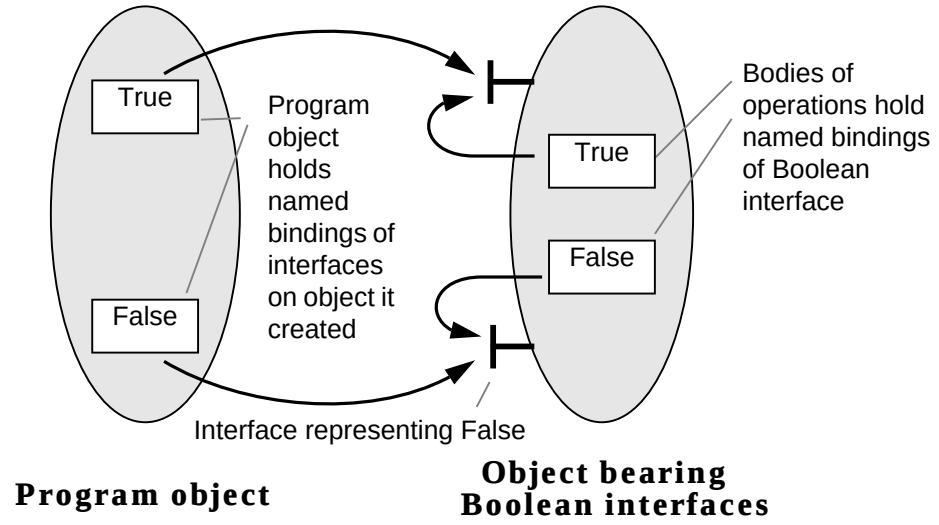
```
object          % the complete program object
   ( Boolean =
       type ( not() ->(Boolean)% The Boolean type
              and(b:Boolean) ->(Boolean)
              or(b:Boolean) ->(Boolean)
              print(s:OutputStream) ->()
              if() ->true() ->false()
          )
   ; true false =
       object
           ( interface% the true interface
               ( not() ->(Boolean)
                    [false]% not true=false
                 and(b:Boolean) ->(Boolean)
                    [b]% true and b=b
                 or(b:Boolean) ->(Boolean)
                    [ true ]% true or b=true
                 print(s:OutputStream) ->()
                    [ t = "true"
                    ; t.print(s)
                    ]
                 if() ->true()
                    [ ->true() ]
               )
           |% No dependency between generation of interfaces
             interface                % the false interface
               ( not() ->(Boolean)
                    [ true ]% not false=true
                 and(b:Boolean) ->(Boolean)
                    [ false ]% false and b=false
                 or(b:Boolean) ->(Boolean)
                    [ b ]% false or b=b
                 print(s:OutputStream) ->()
                    [ t = "false"
                    ; t.print(s)
                    ]
                 if() ->false()
                    [ ->false() ]
               )
           )
    % do the invocations (fig 4.3 shows structure at this point)
    ; j = true.not().or(false.not())
    ; j.print(output)
    )
```

(…) returns the concatenation of the interface lists from its constituent expressions; `(B,C)` delivers the interface list concatenated from the results of expressions `B` and `C`.

[...] throws away the results of all its constituent expressions except the last, which is returned as the result; `[A,B]` delivers the same result as expression `B`

Apart from their use as general expressions, and to control the construction of interface lists, blocks form the bodies of operations (which will be discussed in

Figure 4.3: Object structure in the Boolean example



§4.4) and object constructors (to be discussed in §4.6). For the moment, suffice it to say that in DPL the body of an operation is a block that is executed when the operation is invoked. The bodies of the print operations in the two interfaces constructed in figure 4.2 are each blocks containing two expressions.

Since blocks are themselves expressions, one block may be a constituent expression of another. This nesting is often used to control the form of the interface list generated by the outer block:

`[A,(B,C)]` the result is that of the final expression in [...], here the result of `(B,C)`.

`([A,B],C)` the interface list concatenated from all expressions in (...) is delivered, but [...] delivers only `B`, so that the result is once again equivalent to that of `(B,C)`.

The order and degree of concurrency of evaluation of expressions within a block is be controlled by using one of four different ordering separators. Each block may contain only one kind of separator, but its ordering is independent of any other block, including other blocks nested within it, which can therefore use a different ordering. The intention is to permit the programmer to specify declaratively the ordering and concurrency constraints on the execution of his program, giving the implementation the freedom to choose the most suitable order and concurrency within the specified limits. This permits implementations on parallel machines or making remote invocations to use the available concurrency to the full without forcing the implementation to provide concurrency if it is neither available nor required.

The four orderings and their operators are:

“;” sequential blocks have their expressions evaluated one at a time from left to right in textual order, i.e. sequential blocks execute their expressions with a concurrency of one and in a fixed order.

“,” exclusive blocks have their expressions evaluated one at a time, but in an order selected by the language implementation and not under the

control of the programmer. Specifying exclusive evaluation allows the implementation to choose the evaluation order that suits it best, while guaranteeing a concurrency of one.

"||" concurrent blocks have their expressions evaluated with at least enough concurrency to resolve dependencies between them, so one expression becoming blocked does not prevent execution of other expressions. This gives the implementation latitude to use as much concurrency as is available, but guarantees the minimum required to prevent deadlock.

"|" unconstrained blocks have their expressions evaluated in an order and with a degree of concurrency that is chosen by the implementation, i.e. with a concurrency that may be anywhere between one and the number of expressions. There are assumed to be no dependencies between the expressions, and if this is not the case then deadlock may occur.

In the Boolean example, the two interfaces representing `true` and `false` are constructed independently, and so unconstrained evaluation order has been specified using the "|" separator. The two expressions in the bodies of the print operations must be evaluated sequentially (since the second depends on the results of the first being available), and so the sequential symbol ";" is used. It is important to recognise that these symbols are more than just separators, since they dictate an evaluation order.

Figure 4.4 provides further examples of evaluation orders.

4.3 Interface constructors

The last chapter introduced the ANSA concept of the interface as the point of provision of a service. As was explained there, interfaces in ANSA are created dynamically, making it possible to add new interfaces to existing objects under program control.

This idea of dynamically-created interfaces is realised in DPL by an interface constructor expression:

```
interface (
          % Declarations of operations in the interface
           )
```

The body of an interface constructor is a list of operation declarations, as described in the next section. Each time the interface constructor is executed within an object a new interface to that object is created. The result of the interface constructor is an interface list whose single member is the sole reference to the newly-created interface.

The larger part of the example in figure 4.2 is a pair of interface constructors that make the `true` and `false` interfaces.

4.4 Operations and terminations

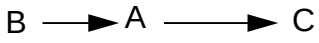
An instance of an interface comprises a number of operations, each of which has a name that is unique within that interface. An operation is specified within an interface constructor by declaring its signature and body. The signature sets out the following details of the operation:

- its name

Figure 4.4: Examples of evaluation orders

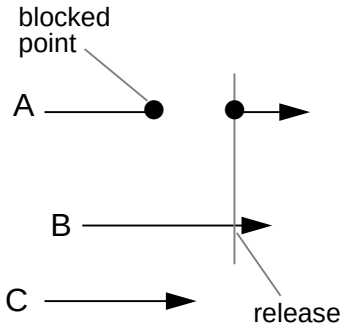
[B; A; C]

The expressions are evaluated one after the other, in the order B, A, C.



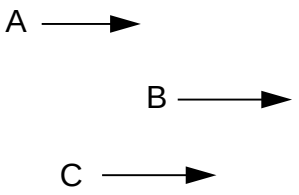
[B || A || C]

The three expressions are evaluated concurrently. Take the case where A is unable to complete until some part of B is completed. At some point, A is blocked pending this synchronisation. B and C are not blocked, however, and may run to completion. At some point B will release A, which can then complete.



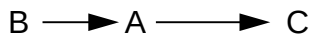
[B | C | A]

The three expressions will be evaluated in any order, with any degree of concurrency, and it is assumed that there are no dependencies between them. If there are in fact dependencies then the result will be unpredictable.



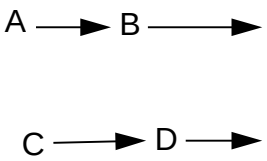
[B, C, A]

The expressions are evaluated one after the other, but in some order chosen by the implementation.



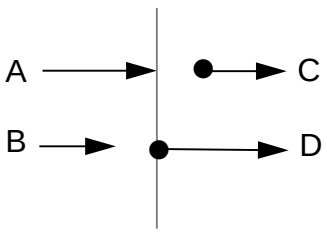
[[A; B] | [C; D]]

A must execute before B, and C before D, but the two pairs A;B and C;D independent of



[[A | B] ; [C | D]]

A is independent of B, so A and B can be executed in any order (including concurrently), similarly with C and D. However A and B must both complete before either of C and D can commence.



- the names and types of its arguments
- the result types of its (optional) anonymous termination
- the termination names and result types of its named terminations (if any)

Every operation must have at least one termination, either anonymous or named. Operations that return no useful information should be defined to return an anonymous termination with no results: the print operation defined in figure 4.2 does this.

Operations may take zero or more arguments and may return zero or more results in each of their terminations. For example:

```
complicated(a:Boolean b:Boolean c:Boolean) ->(Boolean Integer)
                        ->failed(Boolean)
```

This is the signature of an operation called complicated that takes three Boolean arguments `a`, `b` and `c`; and may return two results, a `Boolean` and an `Integer`, via its anonymous termination. Alternatively it may return with the named termination `failed` with a `Boolean` result.

4.5 Type constructors

An ANSA interface type specifies the operations that can be requested of a service provided at an interface, as was described in §3.6.1. DPL has a type constructor `type(…)` which, when evaluated, generates a representation of an interface type[1]. The type required is specified by giving the signatures of its operations. The interface type declaration at the beginning of figure 4.2 has this form; it is analysed in figure 4.5.

Figure 4.5: The Boolean interface type

```
Boolean =
     type ( not() ->(Boolean)
            and(b:Boolean) ->(Boolean)
            or (b:Boolean) ->(Boolean)
            print(s:OutputStream) ->()
            if() ->true() ->false()
          )


name          argument          termination signatures
              name   type         anonymous           named
not             -     -            Boolean              -
and             b   Boolean        Boolean              -
or              b   Boolean        Boolean              -
print           s   OutputStream                        -
if              -     -              -           true() false()
```

1. Further discussion of the workings of the type system, and in particular what exactly is generated by the `type(...)` constructor, can be found in APM/RC.339 *Revising the DPL type system*.

4.6 Object constructors

Each execution of a DPL object constructor causes a new object to be created. The body of the constructor is a block whose evaluation initialises the new object, and whose value provides the value of the constructor.

For example:

`object(e1,e2)` Constructs a new object and evaluates expressions `e1` and `e2` to initialise it. The result of this evaluation (and therefore of the constructor) is the interface list formed by concatenation the results of expressions `e1` and `e2`.

`object[e1;e2]` Constructs a new object and sequentially evaluates `e1` and `e2` to initialise it, returning the result of expression `e2`.

The new object is completely distinct from that in which the constructor was executed; the only connection between them is that the creating object is given the result of evaluating the object constructor block. This block will typically contain an interface constructor, so that the creator is given a reference to an interface on the new object. The following idiom for creating new objects is very common:

```
object
  [ % expressions that initialise the object
    % go here, followed by the interface constructor:
    interface(...)
  ]
```

I n this way the sole reference to the interface on the new object is passed back to the creator of the object. A variation on this idiom is used in figure 4.2 to create a new object with a pair of interfaces representing `true` and `false`.

4.7 Fixed and variable bindings

The example in figure 4.2 uses both kinds of binding supported in DPL: fixed and variable. Each associates a name with an interface. The value of a fixed binding cannot subsequently be changed, although the binding may cease to be accessible (“go out of scope”), and the same name may then be bound to another interface reference. On the other hand, the contents of a variable binding can be changed by assignment. The terms fixed and variable therefore relate to the binding itself, and not to the interface to which the name is bound.

4.7.1 Fixed bindings

Fixed bindings are created in two ways; by executing a definition expression, and by passing parameters to an operation or a termination handler.

4.7.1.1 Definition expressions

Definition expressions have the form;

name {name} **=** expression

(This notation indicates that one or more names, separated by white space, are followed by the “=” character, and then by the expression. This is a simple example of the Extended BNF notation used to define the syntax of DPL - the full EBNF definition of the language can be found in appendix A).

The number of names being bound must equal the number of interface references delivered by the expression. For example;

`a=[x;y]` Evaluate expressions `x` and `y` (in that order), then discard the value of `x` and bind the value of `y` to the name `a`.

`a b=(x,y)` Evaluate expressions `x` and `y` in either order, then bind `x`'s result to `a` and `y`'s result to `b`.

Similarly, in figure 4.2, the names `true` and `false` were bound to references to the two newly-created interfaces to an object:

```
true false =
     object  (
           interface (% the interface representing true
                     . . . . .
           ) |
           interface (% the interface representing false
                     . . . . .
                     )
           )
```

4.7.1.2 Binding by passing parameters

The action of passing parameters when invoking an operation or handling a termination causes the formal parameters in the appropriate operation or handler declaration to be given fixed bindings to the actual parameters in the corresponding positions in the operation invocation or termination signalling form. This will be discussed in grater depth in §4.8 (invocations) and §4.15 (terminations).

Fixed bindings created by passing parameters behave in the same way as those created by definition expressions.

4.7.2 Variable bindings

A variable binding is created by an initialisation expression, which has the form:

declaration {declaration} **:=** expression

Where a declaration has the form:

name {name} **:** typeExpression

Hence a typical initialisation expression might look like this:

```
a b c:Integer  d:Boolean  e f:Complex := thing.op()
```

In this example the operation `op` on the interface thing returns six interface references, the first three of type `Integer`, the fourth of type `Boolean`, and the last two of type `Complex`. These six interfaces are bound to the names `a` to `f` respectively.

An initialisation expression is similar to a definition expression:

- The number of names must equal the number of interface references delivered by the expression on the right hand side
- Each name is bound to the interface reference in the corresponding position in the interface list

From the programmer's point of view there are two significant differences between fixed and variable bindings:

- The interface reference held in a variable binding can be reassigned, using an assignment expression.
- The type to which the type of the interface held in a variable binding must conform has to be declared when the binding is initialised (further details about this requirement and its ramifications can be found in §4.17.3.2)

Here is an example of the syntax used to create and then reassign a variable binding of the name c:

```
c :Integer := a.plus(b)% initialisation

c := a.minus(b)    % reassignment
```

Multiple and repeated assignments can also be made. For example,

```
r i := d.complexCartesianCoords()% operation has 2 results

x := y := a.plus(b)     % x and y both assigned to
                        % the result of a.plus(b)
```

4.7.3 Fixed vs Variable bindings

DPL has two kinds of binding in order to improve the lot of the programmer and to facilitate translation.

Variable bindings correspond to the “variables” of C and other procedural languages. However, “variables” in these languages are often not variable at all, but instead are used by the programmer to name intermediate results, either to aid program readability or because the same value must be used several times over. Faced with the same need, a DPL programmer can use a fixed binding instead. This not only improves program clarity (by clearly distinguishing bindings whose value may change from those that will not) but also enables the DPL translator to produce more efficient code for the creation and use of fixed bindings.

4.8 Invocations

4.8.1 Making invocations

Operation invocations are specified using the syntax:

unit **.** operationName block

The unit is usually the name of an appropriate interface binding, although it may also be a block, an object or another invocation. In each case evaluating the unit must yield as an anonymous termination an interface list containing a single interface (this is automatically true when an interface binding is used). The operation name is a text string in the program source that names the operation in that interface which will be invoked. The block, when evaluated, yields the interface list forming the arguments to the invocation.

The example in figure 4.2 included the following invocations:

```
t.print(s)          % Occurs in two places
j.print(output)
j = true.not().or(false.not())
```

The first invocation is rather simple; it invokes the print operation on the interface bound to `t`, with the interface bound to `s` as the single argument. In the original program `t` is bound to an interface representing a string, and `s` to an interface representing an output stream, so this invocation causes the string to be printed out on the stream.

The second invocation does something rather similar for the interface `j`.

The third example actually contains three invocations. An invocation of the `not` operation on an interface called `true` is used to generate the interface to which is subsequently to be invoked;

```
true.not()
```

The resulting interface has its `or` operation invoked, using the result of the invocation of the `not` operation on the `false` interface as the argument;

```
false.not()
```

This is illustrated in figure 4.6.

This expression could be rewritten equivalently as three separate invocations, with the two intermediate results being bound as constants:

```
j = [ a = true.not()
    ; b = false.not()
    ; a.or(b)
    ]
```

H ere the block on the right hand side of the initialisation of `j` yields an interface list of a single value, the result of invoking the `or` operation on the interface bound to `a`, passing as argument the interface bound to `b`; `a` and `b` in their turn have been bound to the results of invocations of the `not` operation on the interfaces `true` and `false` respectively.

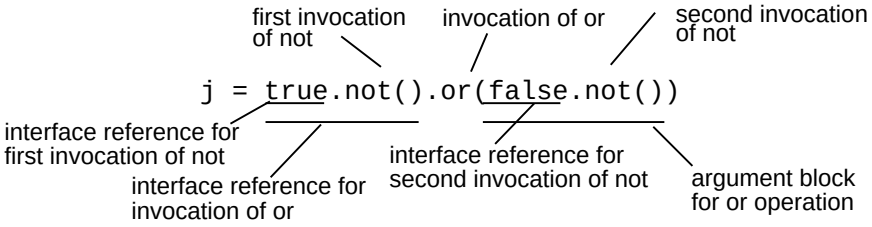
In making an invocation, the unit that returns the invoked interface and the block that provides the arguments are evaluated using exclusive evaluation order - that is to say, one at a time, but in an indeterminate order. The DPL translator checks when it processes the program that each will return an anonymous termination with the right number of interfaces of the right types, and issues a warning if they do not. However, it may be possible for either the unit or the block to produce a named termination at run-time. Should this occur, the evaluation of the entire invocation expression will cease at that point, without the invocation having been performed, and the named termination that caused this will be returned as the overall result of the expression.

Once the interface to be invoked and the interface list that forms the invocation arguments have been successfully computed, the invocation itself can proceed. If successful it will yield one of the results given in the operation's signature; in the case of the Boolean `not` operation, for example, the only outcome given by the signature is an anonymous termination returning a single Boolean interface.

4.8.2 Failure semantics of invocations

Note: This next paragraph is slightly speculative, since the exact structure of engineering terminations has not yet been fixed.

Figure 4.6: Invocation evaluation



Two example evaluations:

Evaluation 1	Result
interface expression for 1st invocation of `or` evaluated	
(interface expression for 1st invocation of `not` evaluated	reference to true* interface
argument block for 1st invocation of `not` evaluated	empty list
operation `not` invoked	result *false*
)	reference to *false* interface
argument block for invocation of `or` evaluated:	
(interface expression for 2nd invocation of `not` evaluated	reference to *false* interface
argument block for 2nd invocation of `not` evaluated	empty list
operation `not` invoked	result *true*
)	reference to *true* interface
operation `or` invoked	result *true*
`j` bound to reference to interface *true*	
end of expression evaluation	result *true*

Evaluation 2	Result
interface expression for 1st invocation of `or` evaluated	
(interface expression for 1st invocation of `not` evaluated	reference to *true* interface
argument block for 1st invocation of `not` evaluated	empty list
operation `not` invoked	result false
)	reference to *false* interface
argument block for invocation of `or` evaluated:	
(interface expression for 2nd invocation of `not` evaluated	reference to *false* interface
argument block for 2nd invocation of `not` evaluated	empty list
attempt to invoke `not` suffers engineering failure	`->invocationFailed()`
)	`->invocationFailed()`
or invocation terminated	`->invocationFailed()`
binding does not occur - result of entire expression	`->invocationFailed()`

* Although interfaces themselves do not have names, for convenience here the two interfaces are referred to as *true* and *false*.

As discussed in §3.2, every operation on every DPL interface also has an extra, implicit named termination not written by the programmer, namely an engineering failure termination. This termination is signalled if the underlying engineering mechanisms cannot guarantee that the invocation took place - in other words, invocation failure in ANSA has at-most-once semantics. If the invocation does not generate the engineering failure termination, but instead one of the programmer-defined terminations, the invocation is guaranteed to have occurred exactly once.

Figure 4.6 includes an example of an unsuccessful invocation.

When a complex expression, such as the invocation of figure 4.6, generates a termination it is not possible to determine directly which subexpression is

responsible. If this is necessary, the expression can be factored into its constituent subexpressions, and the intermediate results bound to constants, as shown in §4.8.1. Each individual subexpression can then be embedded in a termination handler to determine whether it generates a particular termination.

The handling of terminations will be discussed in more detail in §4.15.1.

4.8.3 Behaviour of the invoked operation

When an operation is invoked the block forming its body is evaluated and the result of this (which may be either an anonymous or a named termination) is passed back as the result of the invocation. The binding environment (that is, the set of fixed and variable bindings) in which the block is evaluated is the one that was in force when the interface instance containing the operation was created, augmented by constant bindings of the operation's formal parameters to its actual parameters. The scoping of variable and fixed bindings and the construction of binding environments for interface and object instances will be discussed in full in §4.10.

Note: The fate of terminations which are not part of the operation signature and the precise nature and structure of the engineering terminations are still under review: see RC.182

4.9 Activity constructors

An activity constructor creates a new concurrent activity (see §3.7) to evaluate the block which forms its body.

activity block

Evaluating an activity constructor yields an anonymous termination with no results. The newly created activity evaluates its body independently of its parent and any other activities. It will have shared access to all bindings that were in scope when it was created.

All the expressions in the block forming the body are evaluated by the new activity and the termination of the body is discarded. Communication with other activities is only by side effects; i.e. updating shared variable bindings.

4.10 A brief introduction to scope and closure

This section serves as a brief introduction to the scope rules of DPL, and is provided to facilitate understanding of the material that follows. A complete description of scope rules is to be found in §4.17.

The rules concerning the scope of interface names in DPL are similar to those for variable names in other block structured languages, and will hold few surprises for those versed in a lexically-scoped Lisp (such as Scheme), Pascal, one of the Algols, or (to a lesser extent) C.

The general rule of thumb is that a binding of a particular interface name may be referenced by using its name anywhere within the block in which it is created, at any time after its creation. However, because of the different ordering operators in DPL, it is necessary to be somewhat careful when attempting to define “after” in this context. Since the expressions in a sequential expression list (i.e. delimited by “;”) are evaluated one at a time

from left to right, a reference to a binding is allowed in any expression lexically succeeding the one that created it. A reference from a previous expression would be a *forward reference* and is disallowed, because the binding concerned does not exist at the point of reference. For the other three kinds of expression list (concurrent, independent and exclusive), the temporal evaluation order of the expressions is not known in advance, and hence it is impossible to decide *a priori* whether a binding created in one expression will have been established when referenced from another expression in the same list. For this reason all such references (termed *peer references*) are disallowed.

A binding may also be referenced from within any constructions lexically contained within the range in which the binding is established (for example, a nested block) except where these nested constructions contain something that establishes another binding of the same name. In this case references from within the nested construction refer to this inner, *shadowing*, binding.

Shadowing, peer references and forward references are illustrated by the short program shown in figure 4.7, which includes both legal and illegal references to bindings.

Figure 4.7: Scope and shadowing

```
object
  [ x = 0 ;      % Creates fixed binding of x
    x.print(s) ;% OK - references binding of x to 0

     [ x = "a" ; % Creates shadowing binding of x
       x.print(s)% OK - references binding of x to “a”
     ] ;

    x.print(s) ;% OK - References binding of x to 0

     [ x = "b" ,% Creates shadowing binding of x to “b”
       a = x     % Not OK - This would be a peer reference
                 % to binding of x to “b” , because of the
                 % exclusive order in this block.
                 % It is therefore illegal.
     ]

    y.prints(s) ;% Not OK - this is a forward reference
    y = 99       % Creates binding of y to 99
  ]
```

The general scope rule applies equally when the nested construction is an interface constructor; references to bindings from within the operations defined in the constructor are references to that particular binding of the name, and in this case the binding is said to have been *closed over*. This mechanism enables an interface to capture a variable or fixed binding at one particular instant. This is demonstrated in the program in figure 4.8. In this case each time `op1` is invoked, a new interface containing operation `op2` is created. When `op2` is invoked it is passed an argument `y`, and will return an integer whose value is `y` plus the value of `x` at the time that the interface was created. In the example, `op1` is invoked with arguments `3` and `5`, producing a pair of interfaces (`a` and `b`), each containing an operation called `op2` which adds the appropriate quantity (`3` or `5`) to its argument.

Figure 4.8: Closures and interfaces

```
z = interface
    ( op1(x:Integer) ->(AddOne)
         % This interface could be surrounded by an
         % object constructor - see note in text
         ( interface
              ( op2(y:Integer) ->(Integer)
                   [x.Add(y)]
              )
         )
     )
.
.
.
; a = z.op1(3)     % a is bound to an interface that adds 3
; b = z.op1(5)     % b is bound to an interface that adds 5
; c := a.op2(4)    % c <- 7
; d := b.op2(4)    % d <- 9
.
.
.
```

An object constructor may also capture a binding by closing over it. The program fragment in figure 4.8 constructs a new interface to the current object every time `op1` is executed, but we could modify the program to create new objects by placing an object constructor around the nested interface expression (indicated in the example). In this case the parameter binding of `x` in operation `op1` would be captured in each new object created.

It is of course possible to close over the same variable binding in several object and interface constructors, and in such cases any change in the value held in the binding would be visible from each object or interface concerned. While this poses no problems in the case of interfaces to the same object, shared access to variable bindings from separate objects would provide a back-door communication mechanism between them, thus violating the encapsulation rules of the computational model. In order to prevent this, object constructors are only permitted to close over fixed bindings. This restriction is enforced by the scope rules; no variable binding established outside an object constructor is in scope within the constructor. Fixed bindings may safely be closed over by objects precisely because the value of a particular binding cannot be changed, allowing the implementation to copy it into the object concerned.

4.11 The bank account example

Figure 4.9 provides an example of operation bodies, and the use of (…) and […] for delimiting expressions. It is a much-simplified version of the bank account example used in the *ANSAware implementation manual* to demonstrate the facilities of ANSAware. The main program object constructs two other objects - one encapsulating the bank account, the other the user. The user object is created holding a reference to the service interface provided by the account object.

Figure 4.9: The bank account example

```
object  % Program object - generates account and user objects
  ( account =
     object% account object presents the service interface
   [ balance:Integer = 0;% the current balance
     interface
       (  credit (i:Integer) ->()
         [ balance := balance.add(i)|()]
          debit (i:Integer) ->()
         [ balance := balance.subtract(i)|()]
          getbalance () ->(Integer )
         [ balance ]
       )    % end of account interface
   ]
    object % the client object constructor
   (
      [  account.credit(1000)% credit account
      ;  b = account.getbalance()% get balance and
      ;  b.print(standardOutput) % check by output
      ]
   ;  [  account.debit(589)% debit account
      ;  b = account.getbalance()% get balance and
      ;  b.print(standardOutput)% check by output
      ]
   )        % end of client object constructor
   )        % end of the program object
```

Within the account object the `balance` variable is used to hold the current balance for the (single) user of the service. This variable is closed over by the operations in the service interface generated by the interface constructor: the `credit` and `debit` operations alter the value of the binding, while the `getbalance` operation simply reads the current value.

Note how nesting of blocks is used in the bodies of the `credit` and `debit` operations to ensure that the result of evaluating the operation body matches the operation type declared in the signature. Take for example the expression used in the body of the `credit` operation to increment the balance:

```
balance := balance.add(i)
```

The value of this expression is the value produced by the expression on the right hand side of the assignment, an interface list containing the integer interface representing the new balance. However, the signature of the `credit` operation requires that the result of the operation be an empty interface list:

```
credit (i:Integer) ->()
```

To achieve this an empty block is explicitly used to generate the empty list. The body of the operation is enclosed in […] brackets so that the value returned by this empty block forms the value of the entire operation. Since there are no dependencies between the evaluation of the expression that adjusts the balance and the empty block, the two are specified to have an unconstrained evaluation order.

The client object gets a reference to the bank service because its constructor closes over the fixed binding account. This reference is used to invoke test operations on the service interface to credit 1000 units to the account and debit 589 units from the account. In each case the account is immediately interrogated to find the current balance, which is then printed.

4.12 Factories

Note: The next two sections have been revised to follow the proposals for the type system described in APM/RC.339

Two figures illustrate this section. Figure 4.10 introduces an example built around a diary object, and figure 4.11 expands on this example to introduce the idea of factories.

Figure 4.10: The diary object

```
  .
  .
; diary =
     object
  [ Appointments = SquenceOf(Appointment)
    ; state:Appointments := Sequence.of(Appointment)
    ; interface
       ( read (d:Date) ->(Appointments) ->noAppointments()
       ( ..... ) % Operation to read state
         add (d:Date a:Appointment) ->() ->clash(Appointments)
       ( ..... ) % Operation to add new appts
        ) % end of interface constructor
  ] % end of object constructor
  .
  .
  .
% Test invocation to add operation
; diary.add(dateFactory.create(1, "May", 1990),
          appointmentFactory.create( timeFactory.create(10, 30,
                                                        11, 30
                                                         ),
                                       "Do accounts with Mike"
                                     )
           )
; .
  .
```

Figure 4.10 shows the skeleton of a section of code describing a diary service and demonstrating its use. This diary service is provided by the sole interface to an object. The example presumes that a number of types and factories have been defined;

- `dateFactory` has a `create` operation that manufactures services which represents dates, and whose type is `Date`.
- `timeFactory` has a `create` operation that manufactures services which represent time intervals, and whose type is `Time`
- `appointmentFactory` has a `create` operation that manufactures services which associate dates and times with descriptive strings, and have type `Appointment`

- The type constructor `SequenceOf` and the factory `Sequence` (see chapter 5) are used to construct the collection of appointments in the diary service.

The code in figure 4.10 shows the creation of an object encapsulating the sequence of appointments, and which presents a single interface supporting the diary service. This service interface is bound to the name `diary` in the main program, and its use is illustrated by an invocation of its `add` operation. This invocation has arguments which are themselves the results of invocations. The first argument expected is of type `Date`, and this is provided by the `dateFactory.create` invocation, which is passed strings and integers representing the required date. The second expected argument is an appointment, and this is provided by the `appointmentFactory.create` invocation, which is given a start and end time, and a string giving details of the appointment.

The operation of the diary is straightforward: when the object constructor is executed a new object instance encapsulating a new appointments sequence is created, presenting a single interface. Each time the `read` or `add` operations are invoked, the state of the appointments held in this sequence is updated. A single diary is created, and when the operations are invoked by different users, their entries are held in this diary. This arrangement works well so long as only one instance of a diary is required. Should multiple diary instances be required then a *factory* for diary services is required.

Figure 4.11: The diary factory

```
Appointments = SequenceOf(Appointment);
DiaryType =
   type( read(d:Date) ->(Appointments) ->noAppointments()
         add(d:Date a:Appointment) ->() ->clash(Appointments)
  );
diaryFactory =
  interface
     ( create() ->(DiaryType)
          [ object
             [ state:Appointments := Sequence.of(Apointment);
               interface
               ( read(d:Date)->(Appointments)->noAppointments()
                     ( ..... )
                   add (d:Date a:Appointment) ->()
                       ->clash(Appointments)
                     ( ..... )
                )
             ]
          ]
     ); % end of diary factory
  .
  .
mydiary = diaryFactory.create();
mydiary.add(dateFactory.create(1 "May" 1990),

appointmentFactory.create(timeFactory.create(10,30,11,30),
                          "Do accounts with Mike"));
  .
  .
```

Figure 4.11 shows how such a thing can be made. This example starts by the defining the type `DiaryType`, which has the same two operations `read` and `add` that could be applied to the diary instance constructed in the previous example. An interface is created (bound to `diaryFactory`) which has a single `create` operation, taking no arguments and returning an interface of type `DiaryType`. Invocation of the `create` operation manufactures a new object instance, identical to that created by the object constructor in the previous example; the difference is that an arbitrary number of distinct copies of this object can be created. Each individual user can therefore create his own diary (such as `mydiary` in this example), and add and read his own appointments, independently of other users.

4.13 Making lists

This section explores the construction of an interface which implements a list of integers.

An interface which implements a list of integers should encompass operations to:

- Access the first element of the list
- Return the list minus the first element
- Add a new integer at the front of the list, returning the extended list as the result

Figure 4.12: Type declaration for a list of Integer

```
ListType = type ( Head() ->(Integer)  ->End()
          Tail() ->(ListType) ->End()
          Add(x:Integer) ->(ListType)
        )
```

A possible type declaration for such a list of integers is shown in figure 4.12. The `Head` and `Tail` operations select the appropriate part of the list. Each has an `End()` termination, used to indicate when an attempt is being made to examine an empty list.

Such a list of integers can be implemented as a sequence of cells, each with two parts: a single integer (the head of the list), and another list (the tail). The tail can in turn be implemented as a similar sequence, and so on down to the last element. Each cell can be modelled by an interface giving access to the head and tail.

Figure 4.13: The empty list (nil) interface

```
nil = object
     [ interface
          ( Head() ->End()  [ ->End() ]
            Tail() ->End()  [ ->End() ]
            Add( x:Integer ) ->( ListType )
                 [ ... ]
          ) % end of interface
      ] % end of object
```

The first step in implementing ListType is to implement the interface that represents the empty list. This is sketched in figure 4.13 and is given the name nil. Notice how the signatures for Head and Tail reflect the fact that they can only ever return the named termination End(), never an anonymous termination. For this reason the types of these operations are not equivalent to the corresponding operation signatures in the ListType declaration, but do conform to them.

Figure 4.14: Implementing the add operation

```
nil = object
     [ interface
          ( Head() ->End()  [ ->End() ]
            Tail() ->End()  [ ->End() ]
            Add(x:Integer) ->(ListType)
                 [ object
                      [ interface
                           ( Head() ->(Integer)   [ x ]
                             Tail() ->(ListType)  [ nil ]
                             Add (y:Integer) ->(ListType)
                                  [ ... ]
                           )]]  % end of object
          ) % end of interface
    ]  % end of object
```

The implementation of the Add operation has not been provided in figure 4.13; it must return an interface reference representing a list with a single element (the argument handed to the Add operation on nil). Figure 4.14 extends the nil interface, showing how a first attempt to write the body of the Add operation might look.

The problem that this uncovers is the endless recursion entailed in writing the complete interface for each cell inside the Add operation of the cell that generates it. This is avoided by constructing a factory interface whose purpose is to generate new list cells as separate objects. This is shown in figure 4.15.

Figure 4.15: The cell factory

```
factory = object
    [ interface
        ( MakeCell(y:Integer l:ListType) ->(ListType)
            [ ThisCell =
                object
                    [ interface
                      ( Head() ->(Integer)  [ y ]
                        Tail() ->(ListType) [ l ]
                        Add (z:Integer) ->(ListType)
                          [ factory.MakeCell( z, ThisCell ) ]
                      )]] % end of object
        ) % end of interface
    ] % end of object
```

Note the way in which the fixed bindings of the arguments y and l are closed over to hold the state of the manufactured cell, and the call to the factory from

within the Add operation of the newly-created cell interface. The generated interface ThisCell has Head and Tail operations that always return an anonymous termination, never the End() termination, so here again these operations conform to those declared in ListType, rather than being equivalent to them.

Figure 4.16: Using the cell factory

```
nil = object
    [ interface
        ( Head() ->End()  [ ->End() ]
          Tail() ->End()  [ ->End() ]
          Add (x:Integer) ->(ListType)
              [ factory.MakeCell( x, nil ) ]
        ) % end of interface
    ] % end of object
```

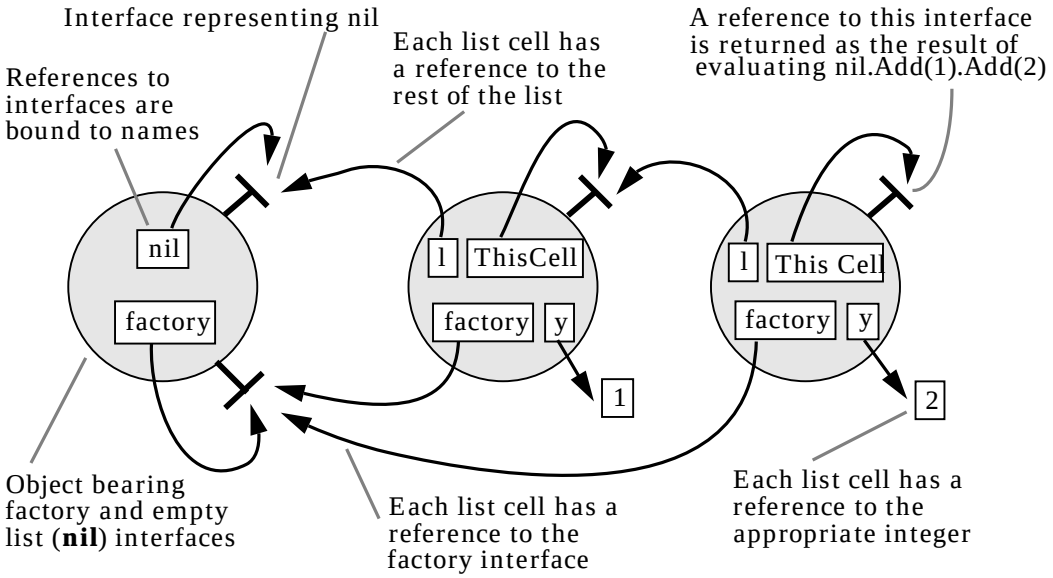
Assuming that the name factory is bound to the factory interface, a complete implementation of nil can now be written. This is shown in figure 4.16.

All the pieces required for a full implementation of the empty list object are now available. Rather than make the cell factory a completely separate object, in this implementation (shown in figure 4.17) it is a second (private) interface to the object which presents the nil interface. Figure 4.18 gives an example of the object structure that might result from using this implementation.

Figure 4.17: Complete implementation of list of integer

```
object
    [ ListType = type ( Head() ->(Integer)  ->End()
                        Tail() ->(ListType) ->End()
                        Add (x:Integer) ->(ListType) )
    ; factory =
        interface
          ( MakeCell(y:Integer l:ListType) ->(ListType)
              [ ThisCell =
                  object
                    [ interface
                        ( Head() ->(Integer)  [ y ]
                          Tail() ->(ListType) [ l ]
                          Add (z:Integer) ->(ListType)
                            [ factory.MakeCell(z,ThisCell) ])]]
          ) % end of factory interface
    ; nil = interface
              ( Head() ->End()  [ ->End() ]
                Tail() ->End()  [ ->End() ]
                Add (x:Integer) ->( ListType )
                  [ factory.MakeCell( x, nil ) ]
              ) % end of nil interface
    ] % end of object
```

Figure 4.18: Object structure in list(2,1)



4.14 Making list factories

The next step in developing a useful list type is to construct a factory to manufacture new, empty lists.

Note: The examples in this section have been altered to use the dependent type mechanism and syntax proposed in APM/RC.339; however, since the translation tools do not yet accept this syntax, the examples are untried.

The specification for such a factory interface includes a `new` operation which returns a newly-generated instance of the `nil` interface on each invocation:

```
ListFactoryType = type( new() ->(ListType) )
```

The required behaviour can be achieved by making the factory's `new` operation create a new object instance and initialise it by executing the code that creates a `nil` interface. The resulting factory for lists of integers is shown in figure 4.19.

The final step to creating a truly useful list type is to parameterise the factory so that it will generate lists of any type, rather than just lists of integers. This can be done by giving the factory's `new` operation a parameter representing the type of the elements of the desired list type.

Note: This example is heavily dependent on the (as yet unimplemented) work on parametric polymorphism described in APM/RC.339. The example in Figure 3-20 uses the currently suggested syntax. Once this has been implemented and is known to work, some suitably descriptive words will be inserted here.

This is shown in figure 4.20.

4.15 Terminations

Terminations were introduced in §3.2 as a way of representing and distinguishing the multiple possible outcomes from the operation. Terminations and termination handlers can be used in a number of ways in DPL:

Figure 4.19: The Integer list factory

```
ListFactory = object
     [ ListType = type ( Head() ->(Integer)  ->End()
                         Tail() ->(ListType) ->End()
                         Add (x:Integer) ->(ListType)
                       );
       interface
        ( new() ->(ListType)
           [ object
              [ factory = interface
                 ( MakeCell(y:Integer l:ListType) ->(ListType)
                    [ ThisCell = object
                       [ interface
                          ( Head() ->(Integer)  [ y ]
                            Tail() ->(ListType) [ l ]
                            Add (z:Integer) ->(ListType)
                             [ factory.MakeCell(z,ThisCell) ])]]
                 ( % end of factory interface
              ; nil = interface
                 ( Head() ->End()  [ ->End() ]
                   Tail() ->End()  [ ->End() ]
                   Add (x:Integer) ->(ListType)
                       [ factory.MakeCell( x, nil ) ]
                 ) % end of nil interface
              ]]  % end of body of new operation
        )  % end of interface
     ]  % end of object
```

- As exception handling mechanisms.
- To distinguish multiple outcomes from an operation.
- To change the flow of control in a program. Terminations and handlers are the only primitive mechanism that DPL provides for conditionalising programs; all conditionals and loops are built using syntactic sugars that expand into expressions involving terminations.

4.15.1 Handling terminations

When the invocation of an interrogation operation is completed a termination of some sort is returned; the signature of the operation reveals the set from which that termination must be chosen. For example, the operation with signature:

```
read() ->(String) ->eof() ->fileError(ErrorType)
```

might be intended to read a string from a file. The normal outcome is an anonymous termination, returning the string which has just been read. If the end of file has been reached, then the named termination `eof` (with no arguments) is returned instead. If some other problem occurred when trying to read the file then the `fileError` termination would be returned, passing a single parameter indicating the cause.

When a named termination is signalled, control is passed to a *termination handler* for a termination of that name; this allows the programmer to define what action to be taken under those circumstances. For example, the `read`

Figure 4.20: The generic list factory

```
ListFactory = object
     [ ListOf(‘X) = type( Head() ->(‘X)  ->End()
                          Tail() ->(ListOf(‘X))  ->End()
                          Add (x:‘X) ->(ListOf(‘X))
                        );
       interface
        ( new(‘T:AbstractType) ->(ListOf(‘T))
            object
              [ factory = interface
                  ( MakeCell(y:‘T l:ListOf(‘T)) ->(ListOf(‘T))
                     [ ThisCell = object
                        [ interface
                           ( Head() ->(‘T)   [ y ]
                             Tail() ->(ListOf(‘T))  [ l ]
                             Add (z:‘T) ->(ListOf(‘T))
                              [ factory.Makecell( z, ThisCell ) ]
                           )]]  % end of MakeCell operation
                  ) % end of factory interface
              ; nil = interface
                  ( Head() ->End()  [ ->End() ]
                    Tail() ->End()  [ ->End() ]
                    Add (x:‘T) ->(ListOf(‘T))
                        [ factory.MakeCell( x, nil ) ])]]
        ) % end of interface
     ] % end of object
```

operation given above might be used in a program fragment like the one shown in figure 4.21.

The evaluation of the `after…handle` expression (called a handled block) begins with the evaluation of the enclosed block, in this case (`s1 := f1.read()`). If this block terminates anonymously then its result (which in this example will be the string read from the file) is delivered as the result of the entire expression. On the other hand, if the block delivers a named termination for which a termination handler is defined, such as `eof()`, then the corresponding block is evaluated, and its result is delivered as the result of the whole expression. In this case the handler for `eof()` closes the file and then returns the null string as its anonymous termination.

Named terminations can pass parameters, just like anonymous terminations.; this is shown by the handler for `fileError(ErrorType)` in the example. The number of parameters in the termination must match that in the handler(s) for that termination.

If there is no appropriate named termination handler, but a default handler block (prefaced by “?”) is present then this is evaluated to yield the overall result. This is illustrated in the simple example in figure 4.22, which catches all named terminations generated by the read operation on the file using a default handler which close the file. Note that a default handler cannot gain access to the parameters associated with a termination (because it isn't known how many there are and of what types), but merely gets control if no handler specific to a termination exists in a particular handled block.

Figure 4.21: Termination example

```
after
     ( s1 := f1.read() )% the block
handle
     ( eof()
         ( f1.close() ; ("") )
       fileError(code:ErrorType)
         [ f1.close();
           ("Warning - read operation generated
error").print(output);
           code.print(output);% Print warning
           ("")
         ]
     )
```

Having closed the file, the default handler in figure 4.22 evaluates the `->reterminate` expression. This causes the named termination currently being handled to be re-signalled, passing control to the next enclosing handled block. While `->reterminate` expressions can be used inside any handler, they are particularly useful within default handlers, since they regenerate the original named termination with its parameters.

Because of DPL's static type system it is always possible to determine in advance the complete set of named terminations that an expression may generate. Hence, unlike the equivalent facility in a dynamically-typed language such as Common Lisp, default handlers are not essential to writing DPL programs, but are merely a convenience feature; a quick way of writing the handlers for all the named terminations that may occur, but which are not explicitly handled.

Figure 4.22: Default handler example

```
after
    ( s1 := f1.read() )
 handle
    ( ? ( f1.close(); ->reterminate ) )
```

A handled block, just like every other expression in DPL, returns either an anonymous or a named termination, and hence has an expression type. This type is automatically determined by the DPL translator to be the type conformed to by every possible expression type that can be generated by the handled block. If the value of the handled block is thrown away (as many are) then this expression type construction can largely be ignored; however, problems arise if terminations of the same name (including anonymous terminations) with differing interface list lengths can be generated in different handlers, since the translator cannot construct a common type for two such terminations. It is therefore quite common to use handled blocks whose body has the following form:

```
after( ... )
handle
    ( term1 () [ ... ; ... ; ()]  % Result is null list
      term2 () [ ... ; ()]        % Ditto
    )
```

The purpose of the empty blocks at the end of the handlers is to ensure that it is possible to construct an expression type to which all the possible outcomes of the handled block conform.

4.15.2 Using terminations to generate recursion

Looping in DPL can be performed using recursion. Figure 4.23 is the classic example of recursion, the factorial function, written in DPL.

The `factorial` operation takes a single argument, the number whose factorial is to be calculated. Its anonymous termination has a single result, the factorial value, and it also has a single named termination, used to indicate when an attempt has been made to find the factorial of a negative number.

Figure 4.23: Constructing a loop using terminations

```
object (    % the complete program object
   f = interface (
       factorial(n:Integer) ->(Integer) ->Failure()
           [i = n.Subtract(1);
           after (i.Sign())% Is i zero?
           handle (
               zero() [n] % Yes - factorial 1 = 1
               positive(x:Integer)[n.Multiply(f.factorial(i))]
               ? [->Failure()]
               )
           ]
       );
   j = f.factorial(5);
   j.print(output)
   )
```

When the factorial operation is invoked its formal parameter `n` is bound to the actual argument; in the example invocation in figure 4.23 this is the integer 5. The expressions forming the body of the operation are surrounded by square brackets […], as the result of the operation is a single interface reference representing an integer, returned by the handled block. The first expression, which does not contribute directly to the result of the operation, assigns the name `i` to value of the intermediate result `n-1`.

The block within the handled block invokes the `Sign` operation on `i`. This is an operation on the Integer type from the type library which returns one of three named terminations: `zero`, `positive`, or `negative`. The latter two each return a single result which is the next integer nearer to zero than the argument (so `5.Sign()` will return the named termination `positive` with the result 4). Named handlers are given for terminations `zero` and `positive`. Any other termination (either the `negative` termination or one resulting from some engineering failure) results in the return of the named termination `Failure`.

Since `i` is `n-1`, the `zero` termination can only occur when `n` is `1`, in other words when the result of the factorial calculation is the same as its argument. For all

other input values, the termination will be `positive`. When this happens, an invocation is made to multiply `n` (the original parameter) by factorial of `i`, which is calculated by a recursive invocation of `factorial`.

Table 4.1 lists the results of the various expression evaluations within each invocation resulting from invoking `f.factorial(5)`.

Table 4.1: Intermediate results in `f.factorial(5)`

n on entry to operation	5	4	3	2	1
i	4	3	2	1	0
f.factorial(i) invocation	24	6	2	1	n/a
positive namedHandler	120	24	6	2	n/a
factorial operation	120	24	6	2	1

4.16 Embedded code

While DPL can be used as a complete programming language, there are many reasons why it must also support embedded code in another language. Primary amongst these is that many useful programs already exist in a variety of languages, and it should not be necessary to rewrite them in DPL. Not only might existing functions and libraries be required, but there is also a need for the ability to package existing programs so that they can be used in a distributed system. In addition, some programming languages are particularly appropriate for certain types of application, and it should be possible to make use of these. All of these requirements are met in DPL by the use of embedded code.

Embedded code takes two forms: executable code and data. Target language data can be inserted in both objects and interfaces, using variants of the object and interface constructors containing a “`data`” clause. This enables target language data declarations to be added to the object (or interface) instance data. The exact form of these declarations depends on the target language and translator.

In a similar way, a “`code`” clause can be used to specify the body of an operation using the target language of the DPL translator. The embedded code must conform to the code generation conventions for the translator when accessing arguments, accessing object and interface instance data, or returning results. These conventions are target language specific.

Figure 4.24 demonstrates a `flag` factory whose `new` operation generates an interface with embedded data and operations with embedded bodies coded in ANSI-C. The example is of an interface which has operations to `clear`, `set` and `test` the new flag value. The `test` operation returns a different named termination according to whether the value is equal to zero or not. Each new flag value is cleared before it is returned. For more information on embedded code see RC.273:*DPL Engineering*.

Note: It is intended at some point to include a section relating DPL to PREPC and IDL in ANSAware.

4.17 Names and scopes

The scope rules in DPL determine what meaning should be attached to interface, termination and operation names in any given context. Although

Figure 4.24: Embedded code example

```
Flag = type ( clear()->() set()->() test()->yes()->no() ) ;
flag = interface
  ( new() ->(Flag)
      [ f = interface data [ int value ; ]
      (
        clear() ->() code
       [
       dpl_Interface *ip = (dpl_Interface *)dpl_pop();
       ip->value = 0;
       return NULL;
       ]
        set() ->() code
       [
       dpl_Interface *ip = (dpl_Interface *)dpl_pop();
       ip->value = 1;
       return NULL;
       ]
        test() ->yes() ->no() code
       [
       dpl_Interface *ip = (dpl_Interface *)dpl_pop();
       if ( ip->value == 0 ) return "no" ;
       else return "yes";
       ]
      )
      ; f.clear()
      ; f
      ]
  )
```

many of these issues have been touched on previously, this section serves to bring together all the relevant information. Because many of the terms used in the description are defined in terms of each other, they will first be briefly introduced, and then individually explained.

4.17.1 Introduction of terms

An *identifier* is a symbol which provides a *name* for an operation, a termination or an interface. Identifiers consist of strings of characters drawn from the set of upper and lower case characters, digits and the underline character (“_”), with the first character being either a letter or the underline character. Letter case is significant within identifiers.

A *binding* is an association between an interface name and an interface instance.

A *range* is a textual region of a program delimited by certain syntactic constructions. Its purpose is to help describe the scoping mechanism for bindings.

The *scope* of a binding is a textual region of a program where all occurrences of a particular interface name refer to that binding.

The *extent* of an instance of an object, interface or binding is the temporal interval in the execution of a program for which it exists.

4.17.2 Name spaces

The names of operations, terminations and interfaces inhabit separate name spaces, and hence the same identifier can be used for an operation, a termination and an interface without any possibility of ambiguity - except perhaps in the mind of the reader!

The operations within an interface (and hence also the operation signatures in a type expression) are distinguished by their distinct and intrinsic *operation names* - no two operations in the same interface may share the same name. On the other hand, it may serve the programmer's purpose to employ the same operation name in many different interfaces. DPL places no restriction on such identically-named operations, which can have completely different numbers and types of arguments and results.

As an example of this last point, in figure 4.2 the `true` and `false` interfaces each have an operation called `if`, but they have different types, since one always signals the `true()` termination, the other always `false()`. However, the operation type of `if` in the `Boolean` type declaration is such that both operations types conform to it.

Operation names are introduced into a program only by writing them in syntactic contexts where an operation name is expected; they cannot be assigned or manipulated in any way.

Termination names are symbolic constants used to distinguish the possible outcomes from an operation. Every operation may also have at most one termination with no name, called the anonymous termination. The outcomes from an operation are distinguished by the termination name alone, so that all the possible terminations from one operation must have distinct names. Termination names are introduced into a program only by writing them in syntactic contexts where a termination name is expected; they cannot be assigned or manipulated in any way.

Interface names denote the interface instances with which they are associated in a binding. The rest of this section is devoted to describing the rules for the construction of bindings of names to interface instances, and for resolving which binding an interface name denotes in any given context.

4.17.3 Bindings

4.17.3.1 Establishing bindings

Initialisation and definition expressions for creating variable and fixed bindings were introduced in §4.7. Although initialisation expressions are the only mechanism for establishing a variable binding, fixed bindings can also be established in two other ways: when an operation is invoked, and when a termination is handled.

Invoking an operation with arguments establishes fixed bindings between the formal parameter names in the operation signature and the corresponding interfaces in the interface list which provides the arguments to the invocation. In the following example the operation and is declared to take a single formal parameter, n. The invocation of the and operation with an argument list consisting of the single interface produced by evaluating the name b causes the body of the and operation to be evaluated with the name n bound as a constant to that interface:

```
    true = object
       ( interface
            ( and(n:Boolean) ->(Boolean)
                 [n]
                 % and operation yields value of n as result
              ...
            )
       )
    ...
  ; c:Boolean := true.and(b)
```

Signalling and handling a named termination also causes a binding to be established between the formal parameter names declared in the appropriate termination handler and the actual parameters of the termination. As an example, figure 4.25 shows the `factorial` operation that you saw earlier recoded to use the Integer type's `sign` operation. In this case the result returned from the `positive` termination, which is the integer next closer to zero from the argument, is used in performing the recursive call to factorial, thus avoiding the explicit invocation of `subtract`.

Figure 4.25: Factorial using termination parameters

```
object
  ( self =
      interface
        ( factorial(x:Integer) >(Integer)
            [ after [ x.sign() ]
              handle
                ( zero() [ 1 ]
                  negative(z:Integer)  [ ->Failure() ]
                  positively(y:Integer)
                 [    % y refers to binding of
                      % termination parameter
                   x.multiply(self.factorial(y))
                 ]
                 )
            ]
        )
  )
```

4.17.3.2 *Referential occurrences*

Both variable and fixed bindings are referenced in the same way, simply by writing the appropriate name in a context where an interface list containing a single value is expected. In each case the interface type of the reference will be that of the binding. In the case of fixed bindings this will be the type of the interface that was originally bound, but this is not necessarily the case for variable bindings.

Consider the following piece of code:

```
object
  ( x:type(print(s:OutputStream)->()) := 5;
                        % x's type only has print op'n
    x.print(output);    % Will print 5
    x := "A string";    % No problem: strings have print
    x.print(output)     % Will print "A string"
  )
```

In this example a variable binding of `x` is created which will hold any interface that conforms to the type:

```
type(print(s:OutputStream)->())
```

The digit `5` is an expression that returns an interface of type `Integer`, which (since it contains the appropriate `print` operation) conforms to the type of the binding of `x`, and hence both the initialisation and the invocation of the `print` operation on `x` succeed. The expression `"A string"` returns an interface of type `String` which (since this also contains the appropriate `print` operation) conforms to the type of the binding, and so the assignment and subsequent printing of `x` succeed here too.

On the other hand, the following piece of code is not type-correct and hence is not legal DPL:

```
object
  ( x:type(print(s:OutputStream)->()) := 5
                        % x's type only has print op'n
    y = x.add(3) )      % NO! x has no add operation
```

Although an interface of type `Integer` has been used to initialise the variable binding of `x`, the binding itself has a type whose signature only has a `print` operation, and hence no operation other than `print` can be performed on the value retrieved from the binding.

4.17.4 Scopes and Ranges

This section discusses the rules that specify the region of the program from which a particular binding may be referenced. A binding is said to be *in scope* in those regions of a program where using the bound name refers to that particular binding (as opposed to another binding of the same name).

The two related concepts used to describe the visibility of bindings from textual regions of the program are *scope* and *range*. The scope of a binding is the textual region of the program from which references to that binding may occur, while ranges are a concept used to describe scopes.

4.17.4.1 Ranges

A range is the region textually contained within certain syntactic constructions, for example the signature and block of an operation body or the block form of a unit (see below for the complete list). When ranges are nested (for example, when an operation body contains a nested block), the region inside the nested construction is part of both the outer and the inner ranges:

```
interface
  ( op( % this argument list is in range 1
     x:Integer)  ->( % This result list is in range 1
               Integer Integer)
     [ % Region inside these []'s is in ranges 1 & 2
       x.print(output);
       ( % Region within these ()'s is in ranges 1, 2 & 3
         x.add(1), x.subtract(1) )
     ]
  )
```

Ranges come in two varieties, namely open ranges and closed ranges. Open ranges are established by:

- each signature in a type expression
- each signature and operation body in an interface expression
- each named termination handler
- all other blocks except those that form the bodies of objects
- each expression in a concurrent, unconstrained or exclusive expression list

Closed ranges are established exclusively by the body of an object expression.

4.17.4.2 Scope of fixed bindings

Fixed bindings can be established in a number of ways. In every case the scope of the binding spans a number of complete expressions. Where one of these expressions establishes a range, the scope includes that range only if the range does not contain a construction that establishes another, shadowing binding of the same name.

The individual cases are:

- Operation invocation establishes fixed bindings of the formal parameter names to the actual arguments. These bindings are in scope for the whole of the block forming the body of the operation.
- Handling of named terminations by termination handlers establishes fixed bindings of the formal parameter names to the actual results in the termination. These bindings are in scope for the whole of the block forming the body of the handler.
- A fixed binding expression forming one of the expressions in a sequential expression list (within a block) establishes binding(s) that are in scope for all subsequent expressions in that list. The expression whose value is being bound is in scope only if it is a type or interface constructor, or a block composed only of these constructors, or a binding whose right hand side is such an expression. This allows the constructing of recursive operations or types (see §4.17.4.5).
- A fixed binding expression forming an expression in a concurrent, exclusive or unconstrained expression list does not establish a scope at all, unless the expression whose value is being bound is a type or interface constructor, or a block composed only of these constructors, or a binding whose right hand side is such an expression, in which case the scope is that expression.

Consider the following rather contrived example, which constructs an interface with a single operation and then immediately invokes the operation

with the argument 7. The body of the operation contains a nested block. This block establishes a range within which a binding of the name x is created. This binding masks the original argument binding of x as 7. However, outside the nested block, that binding of x is not in scope, and references to the name x denote the original (argument) binding. The net result of this program fragment is to print the sequence 7, 8, 16, 7.

```
[ interface
    ( op(x:Integer) ->()
        [ x.print(output);       % x denotes argument binding
          y = x.add(1);          % new binding of y to x+1
          ( x = y.multiply(2);   % new binding of x to 2*y
            y.print(output);     % binding of y still visible
            x.print(output) );   % accesses inner binding of x
          x.print(output)        % accesses arg. binding of x
        ]
    )
].op(7)
```

4.17.4.3 *Scope of variable bindings*

Unlike fixed bindings, variable bindings may be established in only one way: by the evaluation of an initialisation expression. A variable binding is in scope for all open ranges nested within the one in which it is established. Accessing the binding from within a closed range (i.e. the body of an object constructor) is disallowed because the expressions within the body of the constructor execute in a separate object, and to allow them to fetch or update the value of a variable binding would violate the encapsulation rules of the computational model discussed in the last chapter. This problem does not arise for fixed bindings precisely because they are fixed; this permits the value of the binding to be copied into the new object when it is created without violating either the semantics of DPL or the encapsulation constraints of the computational model.

A variable binding expression in a serial expression list (within a block) establishes binding(s) that are in scope for all subsequent expressions in that list, but never for the expression whose value is being bound. Variable bindings in other expression lists establish no scope.

4.17.4.4 *Common idiom for writing objects*

Although DPL is an object-based language, it does not need explicit mechanisms for making object instance variables. Instead the scope rules permit the use of ordinary variable bindings as instance variables, since bindings established within the body of an object can be accessed from within operations in the object's interfaces. This leads to a frequently-encountered DPL programming idiom, of which the following is a trivial example:

```
object
  [ v:Integer := 0;          % v holds object state
     ( interface             % state updated in this interface
        ( put(x:Integer) ->()% operation updates object state
            [ v := x; () ])
     , interface             % state accessed in this interface
        ( get() ->(Integer)
            [ v ]
     ))
  ]
```

Each execution of the object constructor produces a new object instance, with a new variable binding of v to the integer 0. This binding is accessible from the operations of the object's two interfaces, one of which provides a put operation to update the value of the binding, the other a get operation to interrogate the value, with the intention that these two interfaces could be given to different clients.

Since the initialisation expression for v is evaluated each time an object instance is created, each instance could be given a different initial state. Of course, the only bindings from outside the object constructor that can be accessed to calculate this initial value are fixed, but this is by no means as serious a restriction as it at first appears. In particular, since all operation arguments have fixed bindings, variations on the following idiom are often used to allow new objects to be parameterised by the arguments to the operation in which they are created:

```
interface
  ( MakeObject(y:Integer) ->(type ( put(x:Integer) ->()
                      get() ->(Integer)
                      reset() ->() ) )
      [ object
       [ v:Integer := y          % v holds object state
       ; ( interface             % single interface to object
          ( put(x:Integer) ->()  % put operation changes state
              [ v := x; () ]
            get():(Integer)      % get operation fetches state
              [ v ]
            reset() ->()         % reset to initial value
              [ v:=y; () ]
          )
         )
       ]
      ]
  )
```

The important point to note in this example is that the fixed binding of y is “captured” by the references to it within the object constructor. The binding will outlive the activation of the MakeObject operation which created it, and will be accessible at any time that the reset operation is invoked. This process of capturing a binding by constructing a new interface instance that refers to it is known as *closing over* the binding. Each invocation of MakeObject creates a new fixed binding of y, which is in turn closed over by the new interface instance created by the operation. Hence each invocation of MakeObject creates a completely distinct object instance that encapsulates a distinct value of y.

If you refer back to the list example constructed in §4.13 you will find that a similar construction was used there to create and initialise the new list cells.

4.17.4.5 *Recursive definitions*

There will be times when it is desired to construct types or interfaces that refer to themselves, either directly, or indirectly via another type or interface. In order to permit this the DPL scope rules place type and interface constructors on the right hand side of a definition expression within the scope of the names being bound on the left. This allows the construction of recursive

and mutually recursive types and interfaces. In fact examples of this sort of construction have already been seen in the definition of the Boolean type:

```
Boolean = type ( not() ->(Boolean)         % This type refers to
                 and(Boolean) ->(Boolean)  % the name to which
                ...                        % it is bound
               )
```

The interfaces True and False in the same example refer to each other in a mutually recursive fashion:

```
true false =
   object
     ( interface
       ( not() ->(Boolean) [false] % Reference to binding of false
          ...
        )
     , interface
       ( not() ->(Boolean) [true]  % Reference to binding of true
          ...
        )
     );
```

4.17.5 Extents

The general rule for the extent of bindings, objects and interfaces is that they exist for as long as it is possible to access them. Thus an interface instance will exist for as long as it is referenced by at least one binding, and bindings in turn exist for as long as they are accessible via an interface. Objects exist for as long as their longest-lived interface or activity.

4.18 The type model

This section summarises the type model of DPL. As with the previous section on scope rules, many of the issues raised have been covered in earlier sections, but are summarised here for reference and completeness.

4.18.1 Informal statement of type conformance

A useful, informal expression of the type conformance rules is the so-called no-surprises rule, which states that neither the provider nor the user of an interface should receive an operation or termination (respectively) with which it is not equipped to deal. From this it follows that, for an interface type S to conform to an interface type T:

- S must have *at least* the operations of T,
- each operation in T must have *at most* the terminations of the corresponding operation in S.

This reversal of direction in the conformance relationship arises because the user of an interface chooses the operation but the provider of an interface chooses the termination.

This rule is recursively applied to the argument and result types. The direction of conformance for operation arguments is the reverse of that for operation results because the user of an operation supplies the arguments while the provider of an operation supplies the results.

4.18.2 Full type conformance rules

4.18.2.1 Definitions

The type model can be described in terms of five kinds of type;

- expression types
- termination types
- interface list types
- interface types
- operation types

Interface types play a central role in the type model, since all fixed and variable interface bindings have interface types, and type and interface expressions both construct interface types. The interface type is therefore usually the only kind of type which is of interest and in more general contexts is simply abbreviated to “type”. However, other kinds of type are introduced to structure the description of the type model.

Expression types denote the set of terminations that result from making invocations and evaluating expressions in general. Some expressions (such as interface constructors or references to bindings) only ever generate an expression with a single anonymous termination bearing an interface list of length one, but in the general case expression types must also be able to cater for invocations and blocks which have multiple terminations, each with a different interface list.

Termination types associate a termination name (which may be the empty “anonymous” name) with an interface list type.

Interface list types are lists of interface types. For consistency all results are delivered as lists, even if the length of the list is zero or one.

Interface types define the type of an interface or binding as a set of associations between operation names and operation types.

Operation types (also known as signatures) map their argument types to their result type, that is, they map an interface list type to an expression type.

4.18.2.2 Rules

There is a conformance rule for each kind of type. The rules are defined so that, for types S and T, S conforms to T if and only if S provides whatever is required by T. Therefore if S conforms to T, then something of type S may be provided wherever something of type T is required.

An *expression type* E1 conforms to an expression type E2 if and only if:

- for each termination in E1 there is a termination with the same name in E2,
- each termination in E1 conforms to the corresponding termination in E2.

A *termination type* T1 conforms to a termination type T2 if and only if:

- the interface list delivered by T1 conforms to the interface list delivered by T2.

An *interface list type* L1 conforms to an interface list type L2 if and only if:

- the list L1 and L2 are the same length,

- each interface type in L1 conforms to the interface type in the corresponding position in L2.

An *interface type* I1 conforms to an interface type I2 if and only if:

- for each operation in I2 there is an operation with the same name in I1,
- for each such operation in I1 the operation type conforms to the operation type of the corresponding operation in I2.

An *operation type* O1 conforms to an operation type O2 if and only if:

- the interface list type of the arguments of operation O2 conforms to the interface list type of the arguments of operation O1,
- the expression type delivered by O1 conforms to the expression type delivered by O2.

5 DPL library

The core DPL programming language has no built-in types, except for `AbstractType`, the type of a run-time type representation. It is theoretically possible for all the datatypes regarded as primitive in other languages (including integers, strings and so on) to be built upon the foundation of DPL's named operations and terminations. However, in order to be able to write practical programs in DPL, and in order that it can be used as an interface definition language, a library of commonly-used datatypes is provided. These types are divided into two categories; basic types generated via the use of literal expressions, and library types accessible through bindings in the environment in which the program is run.

Note: This chapter describes the first, provisional release of the DPL library. Subsequent work has exposed some shortcomings and omissions, which are identified in editorial comments the text and will in due course be rectified.

5.1 Basic types

The basic types are:

`Integer`	A sub-range of the integers
`String`	Vectors of ASCII Characters
`OutputStream`	An output stream

5.1.1 Integer

5.1.1.1 Type name

DPL programs run by the DPL system are compiled and executed in an environment in which the name `Integer` is bound to the type:

```
type
  (add           (n:Integer) ->(Integer)
   subtract      (n:Integer) ->(Integer)
   multiply      (x:Integer) ->(Integer)
   divide        (n:Integer) ->(Integer Integer) ->divideByZero()
   sign          () ->zero()->positive(Integer)->negative(Integer)
   equal         (n:Integer) ->true() ->false()
   notEqual      (n:Integer) ->true() ->false()
   greaterThan   (n:Integer) ->true() ->false()
   lessThan      (n:Integer) ->true() ->false()
   greaterOrEqual(n:Integer) ->true() ->false()
   lessOrEqual   (n:Integer) ->true() ->false()
   toString      () ->(String)
   print         (s:OutputStream) ->()
  )
```

5.1.1.2 Literals

The DPL translator recognises a sequence of digits (with an optional leading plus or minus sign) as an expression which yields a reference to an interface that represents the integer in question.

5.1.1.3 Values

The present DPL system represents an Integer as a C `integer`, giving a 32 bit two's complement representation on most machines. No checks are currently provided to determine when this range is exceeded.

Note that although it is possible for the user to define a type whose signature conforms to that of `Integer`, an attempt to present such an interface as an argument to an operation (such as `equal` or `add`) on the built-in integer type will not give the desired result, because of the way that integers are currently implemented.

5.1.1.4 Operations

```
add           (n:Integer) ->(Integer)
subtract      (n:Integer) ->(Integer)
multiply      (n:Integer) ->(Integer)
```

These operations perform addition, subtraction or multiplication of the integers represented by the interface on which they are invoked and the argument, returning a reference to an interface representing the result.

```
divide        (n:Integer) ->(Integer Integer) ->divideByZero()
```

This operation divides the integer represented by the argument into that represented by the interface on which it is invoked, returning the interfaces representing the quotient and remainder. Alternatively, `divideByZero()` is signalled if the argument is a reference to zero. The results satisfy:

```
q r = x.divide(y)    % where q*y+r=x and 0<r< y
```

```
sign          () ->zero() ->positive(Integer) ->negative(Integer)
```

This operation is present in order to allow different implementations of integer to communicate their values to each other. The results of the positive and negative terminations are in each case the integer next nearest to zero.

```
equal         (n:Integer) ->true() ->false()
notEqual      (n:Integer) ->true() ->false()
greaterThan   (n:Integer) ->true() ->false()
lessThan      (n:Integer) ->true() ->false()
greaterOrEqual(n:Integer) ->true() ->false()
lessOrEqual   (n:Integer) ->true() ->false()
```

These operations compare the integer represented by the interface on which they are invoked with that represented by the argument, signalling `true()` or `false()` as appropriate.

```
toString      () ->(String)
```

This operation returns a decimal representation of the integer represented by its interface as an String.

```
print          (s:OutputStream) ->()
```

This operation prints a decimal representation of the integer represented by its interface on the output stream s.

5.1.2 String

5.1.2.1 Type name

The DPL translator runs programs in an environment in which the name `String` is bound to the type:

```
type
  (substring     (from to:Integer) ->(String) ->boundsError()
   append        (str:String) ->(String)
   length        () ->(Integer)
   equal         (str:String) ->true() ->false()
   notEqual      (str:String) ->true() ->false()
   greaterThan   (str:String) ->true() ->false()
   lessThan      (str:String) ->true() ->false()
   greaterOrEqual(str:String) ->true() ->false()
   lessOrEqual   (str:String) ->true() ->false()
   print         (s:OutputStream) ->()
  )
```

5.1.3 Literals

DPLTools' parser recognises any sequence of characters between a pair of double quote marks (symbol ", ASCII code 34) as an expression which yields a reference to an interface that conforms to `String`.

Every character between matching pairs of double quote characters forms part of the string except for the escape character (backslash \) and characters immediately following it. The meanings of these escape sequences are:

`\"`	A double quote character
`\\`	A backslash character
`\n`	A newline character
`\r`	A return character
`\t`	A tab character
`\b`	A backspace character
`\f`	A formfeed characte
`\v`	A vertical tab characte
`\a`	A character that causes an audible alarm (if possible)
`\nnn`	A single ASCII character whose octal code is nnn

In addition, adjacent string literals are automatically concatonated into a single string literal - this permits a long string to be split over several lines of DPL source code.

5.1.3.1 Values

The range of values for string characters is at least the 96 printable characters of the ASCII set.

Note that although it is possible for the user to define a type whose signature conforms to that of `String`, an attempt to present such an interface as an argument to an operation (such as `equal` or `append`) on the built-in string type will not give the desired result, because of the way that strings are currently implemented.

5.1.3.2 *Operations*

All operations on strings number the character positions from zero (the leftmost character position).

```
substring     (from:Integer to:Integer) ->(String) ->boundsError()
```

Takes two integer arguments representing string positions, and return a new string of length (`from-to`) consisting of successive characters from the source string starting at position `from`.`boundsError`() is signalled instead if the following inequality does not hold:

$$0 <= from <= to <= length(self)$$

```
append        (str:String) ->(String)
```

This operation takes a string argument and returns a string consisting of the concatenation of the target string and the argument (in that order).

```
length        () ->(Integer)
```

Returns the number of characters in the string.

```
equal         (str:String) ->true() ->false()
notEqual      (str:String) ->true() ->false()
greaterThan   (str:String) ->true() ->false()
lessThan      (str:String) ->true() ->false()
greaterOrEqual(str:String) ->true() ->false()
lessOrEqual   (str:String) ->true() ->false()
```

These operations signal `true`() if the target string bears the given relationship to the argument. The ordering is by comparing the ASCII codings of the characters, with a string considered to be greater than any of its left-substrings. Case is significant.

```
print         (str:OutputStream)->()
```

This operation prints a representation of the string onto the output stream given as argument.

5.1.4 OutputStream

5.1.4.1 *Type name*

DPL programs processed by the DPL translator are executed in an environment in which the name `OutputStream` is bound to the type:

```
type
  ( integer      (i:Integer)->()
    string       (s:String)->()
  )
```

5.1.4.2 *Literals*

There are no literals of this type.

5.1.4.3 *Values*

The interface bound to the identifier `output` is of type `OutputStream`; currently this is the only accessible value of this type.

5.1.4.4 *Operations*

`integer (i:Integer)->()`

This operation causes the integer supplied as an argument to be written to the output. The output format is decimal with minimum number of characters.

`string (s:String)->()`

This operation causes the string supplied as an argument to be written to the output.

5.2 Library types

It is planned to make available a large number of useful data and control abstractions for DPL under the umbrella title of a “DPL Library”. An initial version of this library exists, and contains the following types:

`List`	List of elements of one type
`Stack`	Push-down stack of elements of one type
`Set`	Set of elements from a domain with equal
`Dictionary`	Map from domain with `equal` operation to codomain
`Sequence`	Map from domain with `equal` and `greaterThan` to codomain

5.2.1 Common Operations

It is intended that instances of most types should implement a common set of operations akin to the SmallTalk-80 object protocol. These common operations are supported by the `Integer` and `String` basic types and the `dictionary`, `sequence` and `set` library types.

The common operations are:

`print(x:OutputStream)->()`

This operation prints a representation of the abstraction represented by the interface onto the output stream given as argument.

`equal(x:`*`TypeOf(self)`*`) ->true() ->false()`

The equal operation on an interface, given as argument another interface that conforms to the interface type of the first interface, signals `true`() or `false`() depending on whether the two instances are equal in a sense appropriate to the type. *TypeOf(self)* is a convenient shorthand for the type of the interface upon which the `equal` operation is invoked.

5.2.2 The type constructor protocol

Note: This text describes the protocol supported by the old, DPLTools version of the language, which permitted one interface to represent both a type and a factory. Once the language changes described in APM/RC.339 have been implemented, and the library brought into line with this, then this section will be revised.

All the types currently included in the library are composite types which permit construction of collections of instances of some other type. For each one the library provides an instance of an interface to a type constructor which in turn provides a single operation of. This operation takes as arguments the types(s) used to parameterise the type constructor, and returns an interface that represents the new type. So, for example, the DPL expression:

```
Set.of(String)
```

returns an interface that represents both the type of the set of strings, and a factory that makes new instances of a set of strings. For the latter purpose all such type interfaces also possess an operation new which returns a new, empty instance.

5.2.3 The list, set and stack types

5.2.3.1 List

Note: This definition of a list type differs from the other collection types; these lists are immutable, so that adding a new element to the front of a list generates a new, longer list while at the same time leaving the original list unchanged. From this point of view the list type could be termed “immutable”. We're experimenting with the idea of including both mutable and immutable variants of all collection types in the next release of the library.

`List` is a type constructor for immutable lists of interfaces that conform to a given base type. The base type must include the common `print` operation.

`carcdr () ->(ElementType ListType) ->noMore()`

The `carcdr` operation on an instance of a specific list type returns the element at the head of the list and the list representing the list tail (that is, all the elements except the first). If the list has no elements, then the termination `noMore()` is signalled instead.

`cons (a:ElementType) ->(ListType)`

The `cons` operation on an instance of a specific list type, given as argument an instance of the appropriate type, returns a list whose first element is the argument, and whose tail is the original list. The original list is not modified.

`print (x:OutputStream) ->()`

The elements of the list are printed (by invoking the print operation of each element in turn). The string “`List(`” is printed before the first element, the string “,” is printed between each pair of elements and the `string` “)” is printed after the last element.

5.2.3.2 Stack

`Stack` is a type constructor for mutable stacks of interfaces that conform to a given base type. The base type must include the common `print` operation.

`push (x:ElementType) ->()`

The `push` operation on an instance of a specific stack type, given as argument an instance of the appropriate type, pushes the element onto the top of the stack.

`pop () ->(ElementType) ->noMore()`

The `pop` operation on an instance of a specific stack type removes the top element from the stack and returns it as the result. If the stack has no elements, then the termination `noMore()` is signalled instead.

`print (x:OutputStream)->()`

The elements of the stack are printed (by invoking the `print` operation of

each element) with the top element first. The string "Stack(" is printed before the first element, the string " " (i.e. a space) is printed between each pair of elements and the string ")" is printed after the last element.

5.2.3.3 Set

Set is a type constructor for mutable sets of interfaces that conform to some base type. This base type must possess print and equal operations that match the signatures given in the common protocol. The print operation will be used to print individual members of the set, while the equal operation will be used when testing for set membership. If the base type's equal does not have the commonly-accepted semantics of an equality operation (i.e. if it is not an equivalence relation) then attempting to construct a set from that type will give very unpredictable results.

`add (x:DomainType) ->()`

The add operation on an instance of a specific set type, given as argument x, an instance of the appropriate type, inserts x into the set if and only if the set already contains no element equal to x.

`member (x:DomainType) ->true() ->false()`

The member operation on an instance of a specific set type, given as argument x, an instance of the appropriate type, signals either true() or false() depending on whether the set contains an element that is equal to x.

`delete (x:DomainType)->()`

The delete operation on an instance of a specific set type, given as argument x, an instance of the appropriate type, removes from the set the instance which is equal to x, if present.

`subset (x:ThisSetType) ->true() ->false()`

The subset operation on an instance of a specific set type, given as argument x, another instance of the same set type, signals true() or false() depending on whether x is a subset of the instance.

`iterator () ->(type (next() ->(DomainType) ->noMore()))`

The iterator operation on an instance of a specific set type returns an interface with a single operation, next. Successive invocations of this interface will return all the elements of the set, in no particular order, and thereafter signal noMore() whenever invoked. Performing destructive operations on the set (i.e. add and delete) between invocations of the next operation yields undefined results.

5.2.4 The Mapping Abstract Supertype

The dictionary and sequence type constructors conform to an (unwritten) abstract supertype called mapping, while possibly also augmenting the operations provided by the abstract supertype. Mapping provides a way of building types that map the values in some domain onto the values of a codomain. Both domain and codomain types must possess a print operation, while the domain type must possess an equal operation with appropriate semantics. (see section 4.2.3.3). The mapping type supports the following operations:

```
get            (x:DomainType)->(CodomainType) ->notPresent()
```

The get operation on an instance of a specific mapping type, given as argument x, an instance of the domain type, retrieves the corresponding codomain element from the mapping, or signals notPresent() if no mapping exists. The operation does not change the instance of the mapping type.

```
put            (x:DomainType y:CodomainType)->()
```

The put operation on an instance of a specific mapping type, given as arguments x and y, instances of the domain and codomain types, adds the mapping from x to y into the instance of the mapping type, possibly overwriting any existing mapping from x.

```
filterDomain   (x:type(Predicate(y:DomainType) ->true() ->false()))
               ->(TypeOf(self))
```

The filterDomain operation on an instance of a specific mapping type, given as argument x, an interface containing a predicate operation that signals true() or false() depending on its single argument drawn from the DomainType, returns a new instance of an equivalent mapping type that contains only the mappings where the domain elements satisfy the predicate. The original mapping is not changed.

```
filterCoDomain(x:type(Predicate(y:CoDomainType) ->true()
                                                ->false()))
               ->(TypeOf(self))
```

The filterCoDomain operation on an instance of a specific mapping type, given as argument x, an interface containing a predicate operation that signals true() or false() depending on its single argument drawn from the CoDomainType, returns a new instance of an equivalent mapping type that contains only the mappings where the codomain elements satisfy the predicate.

```
iterator       () ->(type(next()->(DomainType CodomainType)
                  ->noMore()))
```

The iterator operation on an instance of a specific mapping type returns an interface with a single operation, next. Successive invocations of this interface will return all the (domain, codomain) pairs in the mapping, and thereafter signal noMore() whenever invoked. The order in which the elements are yielded depends upon the kind of mapping. Performing destructive operations on the mapping (i.e. add and delete) between invocations of the next operation has undefined effects.

The mapping protocol will in due course be extended to include other common operations such as size of mapping.

5.2.4.1 *The Dictionary type*

The Dictionary type is equivalent to the mapping abstract supertype. The only reason for keeping the two notions separate is to guard against some future necessity of adding operations to Dictionary that are not appropriate for the abstract supertype.

The interface returned from the iterator operation of a Dictionary delivers its results in an undefined order.

5.2.4.2 The Sequence type

The Sequence type also conforms to the mapping type, and at present adds no operations to it. However, the domain type of a sequence must have a `greaterThan` operation defined upon it - this permits Sequence to implement the mapping operations more efficiently.

The interface returned from the `iterator` operation of a Sequence delivers its results in increasing order on the domain type for the Sequence, with elements with equal keys being produced in arbitrary order.

A Formal DPL Syntax

A.1 Notation

The syntax of DPL is defined using extended Backus Naur Form (BNF) as the meta-syntax.

A.1.1 Meta-syntax

The meta-syntax is used to define how the language tokens may be composed into programs. It is defined as BNF with the simple extension of enclosing iterative parts of a production rule in curly brackets.

xyz	denotes the terminal symbol xyz
xyz	denotes the non-terminal symbol xyz
→	delimits a non-terminal symbol from its production rule
|	delimits alternative parts of a production rule
[ ]	enclose an optional (0 or 1 occurrences) part of a production rule
{ }	enclose an iterative (>=0 occurrences) part of a production rule

A.1.2 Lexical Meta-Syntax

The lexical meta-syntax is used to define how the language symbols may be composed into language tokens. It is defined as the meta-syntax with the following extensions, which may contain any character in the character set:

***	skips input characters until the closing bracket symbol which matches the immediately preceding opening bracket symbol; nested on every opening bracket symbol and un-nested on every closing bracket symbol while nested
...	skips input characters until the following terminal symbol
0--9	denotes any input character in the range **0** to **9** inclusive
tab	denotes the appropriate control character

These extensions are sufficient to define both types of DPL comments but need additional refinement to define the syntax of embedded blocks (see §A.2.3) and string literals (see §A.2.4).

A.2 Lexical syntax

Before the text of a program is parsed it must first be scanned to remove the separators and to compose the remaining symbols into a stream of tokens which form the terminal symbols of the DPL language. The set of valid tokens is composed of the lexical tokens defined in §A.2.2 plus the characters and keywords explicitly defined in §A.3.1.2 and §A.3.1.3.

A.2.1 Separators

Separators are recognised and processed before the token stream is constructed. A separator separates adjacent tokens but is otherwise ignored.

separator → *comment* | space | newline | return | tab | form feed | vertical tab

comment → **{** *** **}** | **%** ... newline

A.2.2 Lexical tokens

The following productions are used in the definitions of the lexical tokens:

digit → **0--9**

letter → **a--z** | **A--Z** | _

number → [**-**] *digit* { *digit* }

string → **"** ... **"** % see §A.2.4 for escape sequences

The following productions define the syntax of the lexical tokens:

embedded → **[** *** **]** % Square brackets in comments and
% constants are ignored (see §A.2.3)

identifier → *letter* { *letter* | *digit* }

literal → *number* | *string*

A.2.3 Embedded blocks

Within an embedded block any square brackets enclosed in string constants, character constants or comments must be ignored; this is dependent on the syntax of the embedded language. Macro definitions and calls may also contain square brackets but as these are impossible to process in incomplete program fragments, the programmer is responsible for ensuring that they are properly balanced in an embedded block.

A.2.4 String literals

A string literal is a (possibly empty) sequence of characters enclosed in double quotes. This sequence may contain any characters in the source character set except the double quote **"**, backslash ****, or newline characters.

These characters, plus others that would be impossible or inconvenient to enter directly in the source program, may be included in string literals by using an escape sequence. Escape sequences are introduced by a backslash character. The effect of an invalid escape sequence is undefined.

A.2.4.1 Character escape sequences

Character escape sequences are used to represent some common special characters independently of the source character set. They consist of a backslash followed by an escape code. The escape codes and their meanings are:

a alert (bell) **n** newline **v** vertical tab

b	backspace	**r**	return	****	backslash
f	form feed	**t**	tab	**"**	double quote

Both the backslash and the following escape code are replaced by the appropriate character for the escape code in the target character set.

A.2.4.2 Numeric escape sequences

Numeric escape sequences allow any character in the target character set to be expressed by writing the numeric value of the character in octal. They consist of a backslash followed by up to three octal digits. A numeric escape sequence terminates when three octal digits have been used or when the first character that is not an octal digit is encountered. The backslash and valid octal digits are replaced by the character in the target character set with that numeric value.

A.2.4.3 Concatenation

Adjacent string literals are automatically concatenated into a single string literal. This is intended for string literals that will not fit on one source line.

A.3 Language syntax

The sentence symbol in the DPL language syntax is *program* (i.e. every syntactically correct DPL program can be generated by repeated application of the productions in §A.3.4 to the non-terminal *program*).

A.3.1 Terminal symbols

The set of terminal symbols in the DPL language syntax consists of the lexical tokens, characters and keywords defined below.

A.3.1.1 Lexical tokens

The following lexical tokens (defined in §A.2.2) are terminal symbols:

embedded identifier literal

A.3.1.2 Characters

The following characters and character pairs are terminal symbols:

( ) [ ] < > ? . , ; | : = || := ->

No separators are allowed between the two characters in a pair.

A.3.1.3 Keywords

The following keywords are terminal symbols:

activity	**code**	**handle**	**object**	**type**
after	**data**	**interface**	**reterminate**	

A.3.2 Reserved words

There are no reserved words; all of the keywords can be used as identifiers. This will not confuse a parser but it may confuse a human reader.

A.3.3 Semantic Indicators

The grammar defines the positions in a DPL program where *identifiers* may be written, but not which *identifiers* may be used in any particular place; this is dictated by the scope rules (for *names*); the type system (for *operationNames* and *terminationNames*); the attribute libraries (for *attributeNames*) and the compiler (for *languages*).

The following productions are strictly redundant; they are used to convey semantic information such as to which name space an *identifier* belongs.

attributeBlock → *block*

attributeName → *identifier*

language → *identifier*

name → *identifier* % of an interface binding

operationName → *identifier*

terminationName → *identifier*

A.3.4 Syntax rules

activity → **activity** *block*

argumentList → **(** { *declaration* } **)**

assignment → *name* { *name* } **:=** *expression*

attributeList → **<** { *attributeName* [*attributeBlock*] } **>**

block → **(** [*expressionList*] **)** | **[** [*expressionList*] **]**

declaration → *name* { *name* } **:** *typeExpression*

defaultHandler → **?** *block*

definition → *name* { *name* } **=** *expression*

expression → *activity* | *assignment* | *definition* | *handledBlock* | *initialisation* | *interface* | *literal* | *terminate* | *type* | *unit*

expressionList → *expression* [{ **;** *expression* } | { **,** *expression* } | { **||** *expression* } | { **|** *expression* }]

handledBlock → **after** *block* **handle (** { *namedHandler* } [*defaultHandler*] **)**

initialisation → *declaration* { *declaration* } **:=** *expression*

interface → **interface** [*attributeList*] [**data** [*language*] *embedded*] **(** { *signature operationBody* } **)**

invocation → *unit* **.** *operationName block*

namedHandler → *terminationName argumentList block*

object → **object** [*attributeList*] [**data** [*language*] *embedded*] *block*

operationBody → *block* | **code** [*language*] *embedded*

program → *object*

resultList → **(** { *typeExpression* } **)**

signature → *operationName* [*attributeList*] *argumentList*
-> [*terminationName*] *resultList*
{ **->** *terminationName* *resultList* }

terminate → **->** *terminationName* *block* | **-> reterminate**

type → **type** [*attributeList*] **(** { *signature* } **)**

typeExpression → *type* | *unit*

unit → *name* | *invocation* | *block* | *object*