Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (0223) 323010**
INTERNATIONAL: **+44 223 323010**
FAX: **+44 223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

The ANSA Computational Model

Owen Rees

Abstract

The ANSA computational model is a framework for describing the structure, specification and execution of programs in the context of ANSA. It emphasises the principles of encapsulation and abstraction that are essential for the design of programs for use in an evolving distributed environment which is not necessarily subject to any central authority.

The model is in two major parts: an interaction model and a construction model. The interaction model defines the relationships between service providers and service users; this includes the invocation and parameter passing scheme and a corresponding type system based upon conformance. The construction model defines the relationship between an object an idealised supporting environment.

A representation of the model in its own terms is also introduced. This provides a basis both for an explanation of the type analysis and for the introduction of the principle of transformation to provide attributes introduced declaratively.

APM.1001.00.02 **Draft** 23 June 1993

Architecture Report

Distribution:
Supersedes:
Superseded by:

The ANSA Computational Model

Architecture Report

The ANSA Computational Model

Owen Rees

APM.1001.00.02

23 June 1993

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	(0223) 323010
INTERNATIONAL	+44 223 323010
FAX	+44 223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Introduction

This document describes the ANSA computational model. This chapter explains the design principles and background to the model, and gives a short overview of the major features of the model. The rest of the document gives detailed definitions and explanations of the model.

1.1 Computational models

1.1.1 What is a computational model?

A computational model is a framework for describing the structure, specification and execution of programs. There are many possible computational models and every programming language has some underlying model that defines the nature of the entities that can be manipulated and the manipulations that can be performed upon those entities.

A computational model differs from a programming language in that a model does not prescribe a particular syntax: for example, the internal structures used by compilers are as much a representation of the model as is a textual form written by a programmer. It is also usual for a programming language to provide simple syntactic forms for expressing useful compositions of the underlying elements; in a computational model, such things are inappropriate.

The purpose of the ANSA Computational Model is to define the facilities required of a programming system that permits the implementation of distributed applications for an ANSA infrastructure (i.e. an infrastructure conforming to the ANSA engineering model). The design principle for the computational model is to minimize the amount of engineering detail that the application programmer is required to know, yet at the same time not preventing the programmer from exploiting the benefits of distributed computing.

The audience for the computational model is expected to be language designers and the authors of software libraries who want to provide a programming system for application programmers to use. Those who want to read a more direct account of how the model is manifested in a particular language should read *DPL Programmers' Manual* [APM1014].

1.1.2 ANSA computational model design principles

Distribution imposes various constraints on programs but also provides the means for enhancing certain qualities such as performance and reliability. The ANSA computational model was developed because no existing model or language was able to cope with all of the constraints that arise directly or indirectly from distribution.

Some existing models have come close to satisfying the needs of distribution and these, especially Emerald [BLACK 86][BLACK 87] and Argus [LISKOV 88], have influenced the design of the ANSA computational model.

The problem space for distributed systems is described briefly in §1.2. A discussion of the issues of distributed computing, with particular reference to the use objects can be found in *Distributing Objects* [APM1009].

The design of the ANSA computational model has been guided by the following principles:

Separation of "service" and "implementation": Software is viewed at two levels: (1) as an entity providing a service accessible by other parts of the system; (2) as a statement of an algorithm meeting the same service requirement. Service is the stronger notion and is the focus of the model. There can be many implementations, based on the ANSA computational model or other computational models capable of meeting a service requirement. This is the foundation for masking heterogeneity and preserving openness in ANSA.

Minimum orthogonal set of concepts: The model should not include unnecessary concepts and should clearly separate independent concerns. This is achieved by having few concepts and general combination rules rather than a proliferation of special cases. A particularly important feature of the ANSA computational model is that concurrency and communication are separated.

Symmetry: Parts of the model that are similar should be made the same. Even where distinct concepts are needed, the model should minimise the number of differences between concepts.

Maximum opportunity for early checking: The model should promote the early checking of consistency of programs. The model is strongly typed in accordance with a principle of avoiding surprises. The model should include declarative rather than imperative constructs since these are easier to analyze for consistency and can be translated into an efficient implementations for a wider range of supporting environments.

Selective distribution transparency: The model should allow for the insertion of a transparency policy rather than prescribing a particular policy.

Multiple implementations of types: The model should permit entities of the same type to be constructed in different ways. This is essential to accommodate heterogeneity.

Federation as well as hierarchy of types: The model should allow independently developed collections of types to be brought together without requiring them to be put into a hierarchical structure and related by a common superior.

Remote case general - local case special: The model should be based upon the idea that entities exist in different places. It should not prevent the optimizations that can be made when entities are and will always remain in the same place but this should be considered as a special case rather than being the normal assumption. This gives the maximum flexibility for program configuration in different environments.

Must be an abstraction of ANSA engineering: The model must recognise and accommodate the constraints imposed by engineering concerns, in particular, the ANSA engineering model must be the basis of a valid implementation of the computational model.

Open to a wide range of languages: It must be possible to overlay the model on a wide range of current languages as a basis for adding distribution capabilities to those languages.

1.2 The problem space

In a distributed system, a number of assumptions that may be made for a centralised system are not valid. Being able both to cope with the problems and to exploit the opportunities of a distributed system requires a different set of assumptions. The key issues on which these new assumptions must be based are:

- separation - pieces of program exist in separate places
  - failures - more failure modes may be visible
  - latency - interaction delays may be significant
- concurrency - multiple processors gives true concurrency
  - a resource to be exploited
  - potential for conflict
- heterogeneity - interworking between different kinds
  - of machines
  - of operating systems
  - of programming languages
- federation - autonomy of administration, no common superior
  - ownership of data, limited sharing
  - independently developed programs
- evolution - continuous gradual changes
  - dynamic binding
  - migration of services
  - new types with arbitrary relationships to existing ones
- scaling - solutions must range from small to very large
  - number of machines
  - number of users
  - volume of data
  - physical distances

These distribution issues invalidate the assumptions of globality that are common in centralised systems. Both the application programs and their supporting infrastructures have typically assumed global memory, global knowledge, global names and global trust.

In a distributed system, it is not valid to assume that parts of a program can communicate through memory that appears fast and failure free from everywhere.

Algorithms that depend upon global knowledge, for example for resource management or scheduling, are inappropriate because the information will be out of date before it has all been collected.

The allocation of new unambiguous global names depends upon global knowledge of which names are already in use, it also depends upon the willingness of administrations to give up some of their autonomy.

The assumption that the parts of a program are all equally well protected and equally trustworthy is also inappropriate for a distributed system.

The ANSA computational model has been designed to accommodate these issues. A particularly important point is to distinguish the computational effects from the mechanisms that may be used to achieve those effects. The computational model defines the computational entities and the effects of various manipulations; the mechanisms that can be used to achieve those effects efficiently are defined by the engineering model.

The separation between the computational view and the engineering view is important because different mechanisms may be appropriate to achieve the same effect under different circumstances. Abstraction, and particularly the hiding of mechanisms behind well defined interfaces, is not just a matter of good practice in a distributed system, it is the means by which real differences in implementations are hidden so that distributed applications can be successfully deployed in heterogeneous and dynamically changing environments.

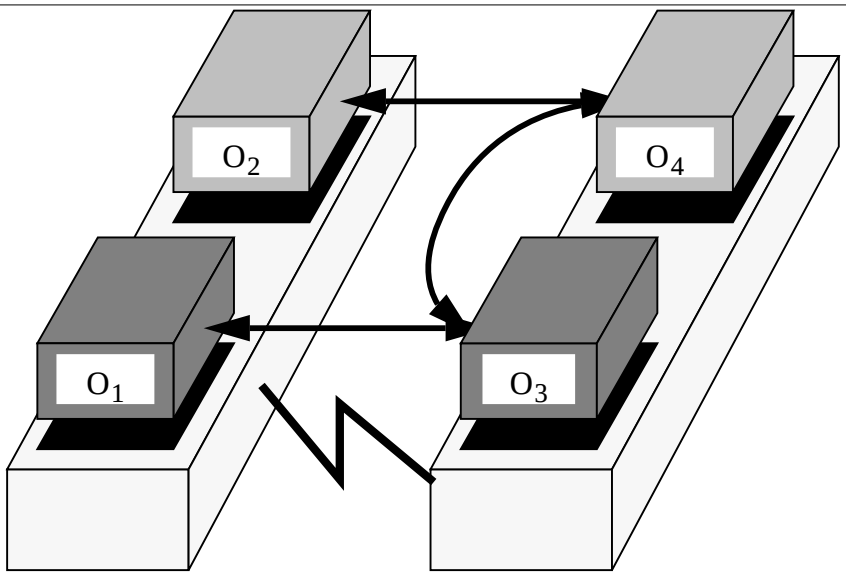
1.3 Concepts

The computational model provides a view of an ANSA system as a distributed, multitasking abstract machine supporting objects executing sequential activities. An activity in one object can access other objects without needing to know how they are implemented or where they are located. Interaction between objects may become impossible because of communication problems and/or partial failures of the distributed virtual machine.

1.3.1 Illustrative example

The concepts of the ANSA computational model will be illustrated by examples based upon the simple system shown in Figure 1.1.

Figure 1.1: A simple system



In this system there are four computational objects supported by two physically separate and differently implemented infrastructures, one of which supports O_1 and O_2 and the other O_3 and O_4. It is possible for enterprise issues such as the ownership of the objects to give a quite different grouping from that corresponding to the choice of supporting environment. For example, O_2 and O_4 might belong to one department in an organisation whereas O_1 and O_3 belong to a different department.

A programmer responsible for O_3 should not be forced to deal with the fact that O_1 and O_2 run on a physically separate and different kind of machine nor with the fact that O_2 and O_4 will be owned by a different department unless these facts have a direct bearing on the problem in hand. The focus of attention should be on the services that O_3 is to provide for use by the other objects and the services provided by those objects that O_3 may use.

If O_3 uses a service provided by O_2, it must know what requests it can make, what information must be supplied with each request and the range of possible responses. O_3 may use a number of different services and it must be able to specify the particular service that it wishes to use.

O_3 may also use a service provided by O_4 and may wish to pass the ability to use that service to O_2. For example, suppose that O_4 is an output device that provides a stream upon which O_3 may request that data be written, O_3 may wish to pass that stream as a parameter when using the service provided by O_2 so that O_2 can request that data be written to the stream. The ANSA computational model provides the concepts necessary to describe such a situation independently of the infrastructures that support the various objects.

1.3.2 Summary of concepts

The concepts of the ANSA computational model are summarised below. Full descriptions of the concepts are given in the chapters that follow.

(Computational) object: a unit of program modularity having state and *operations* for initializing, accessing and updating that state. Object state may contain references to the interfaces of both the object itself and other objects.

Interface: a view of an object as an abstract service. An interface is specified as a set of *operations* together with synchronization and ordering constraints on the use of those operations.

Operation: part of an interface. An operation has a *signature* and a body which defines the effect and outcome from an *invocation* of the operation.

Signature: a specification of the name of an operation, the number and *interface types* of the argument parameters and, optionally, a set of *terminations* which specify the possible outcomes from the operation.

Activity: the agency by which computations make progress. An activity may pass from one object to another by the first *invoking* an operation on an interface of the second. Activities may split into parallel sub-activities and later recombine. New activities can be initiated to proceed in parallel, these may be able to communicate with other activities but are not dependent upon their initiating activity.

Termination: the specification of a set of possible outcomes from *invocations* of an operation. A termination has a name and specifies the *interface types* of the result parameters for an outcome with that name.

Interface type: a schema for an interface, the signatures of the operations in interfaces of the type.

(Operation) Invocation: the execution of the body of an operation defined by a reference to an interface and an operation name in a context established by the referenced interface and a set of arguments.

Server: in the context of an invocation, the object which provides the interface containing the operation being invoked.

Client: in the context of an invocation, the object from which the invocation was initiated.

1.4 Structure of the model

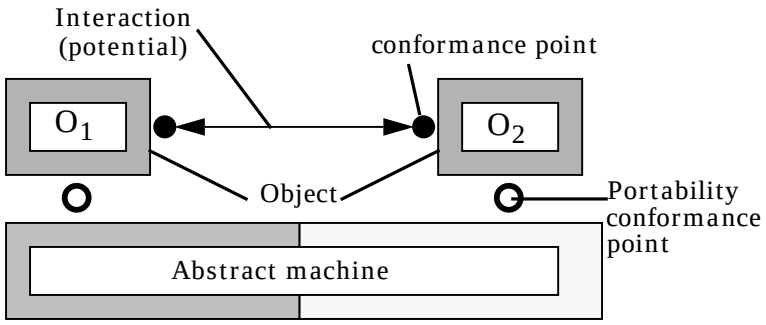
The ANSA computational model is in two parts:

- the interaction model defines permitted forms of interaction and a type scheme within which potential interactions are to be classified.
- the construction model defines elements from which the interacting objects may be constructed.

The structure of the model and the organisation of the description of the model are derived from the relationships that exist between computational objects and the relationship between a computational object and its supporting environment. The model establishes conformance requirements that must be satisfied if the pieces of a distributed system are to fit together.

There are two computational conformance points; the interworking conformance point and the portability conformance point. Figure 1.2 shows two objects and the positions of the conformance points with respect to the objects and the environment that animates them.

Figure 1.2: Computational conformance points



The **interaction model** which is described in Chapter 2 addresses the relationship between objects and the issues of interworking conformance.

The **construction model** which is described in Chapters 3 and 4 addresses the relationship between an object and its supporting environment and the issues of portability conformance.

It is possible to conform to the interaction model without conforming to the construction model.

Conforming to the construction model guarantees conformance to the interaction model since there are no interaction facilities other than those corresponding to the interaction model.

1.4.1 Interworking conformance

At the interworking conformance point there are two kinds of conformance. The first is conformance to the interaction model. The second is interface type conformance for potential interactions.

An **interaction conformance** statement for an object asserts that all interactions at the conformance point follow the rules of the interaction part of the ANSA computational model.

Interface type conformance applies to the potential interactions between objects rather to the objects themselves. An interface type conformance statement can be made only about a potential interaction in which the participant objects are interaction conformant. An interface type conformance statement for a potential interaction asserts that neither party to the interaction will attempt to interact in a way that the other does not expect.

Objects cannot interact if their models of interaction are different. Interaction conformance is mandatory for an object that is to participate in an ANSA system.

1.4.2 Portability conformance

The portability conformance point is between an object and the abstract machine which animates it.

A statement of portability conformance for an object asserts that the object is defined in terms of the elements of the ANSA construction model.

A statement of portability conformance for an abstract machine asserts that it can animate objects that conform to the ANSA construction model.

Each object must match the animation environment that supports it. The animation environments in a system need not conform to the ANSA construction model. If a system has animation environments based upon more than one model then there will be restrictions upon where each object may be placed which will limit the way in which the system resources can be exploited.

The ANSA construction model has been designed to be well matched to the interaction model and also to permit the development of mechanisms and techniques that allow the resources of a distributed system to be exploited effectively.

2 Interaction Model

2.1 Overview

This chapter describes the interaction part of the ANSA computational model. The interaction model is concerned with the interfaces provided and used by the objects in a system. The interaction model consists of an invocation scheme and a type scheme.

The invocation scheme defines how clients may use interfaces provided by servers.[1] It defines the permitted forms of interaction and also the parameter passing model.

The type scheme provides a set of types into which interfaces are classified and defines a relation over interface types that allows the detection of the possibility of interaction errors (as defined in §2.3.2) before the interaction commences.

2.2 Invocation scheme

A client interacts with a server by invoking operations in interfaces provided by the server. There are two kinds of operation, interrogation and announcement.

Invocation of an interrogation is a synchronous request/response style, the activity which invoked the interrogation continues within the object that provides the invoked operation, and subsequently returns to the object from which the invocation was made. There is no change in the degree of concurrency in an interrogation.

Invocation of an announcement is an asynchronous request only style, a new activity is created in the object that provides the invoked operation, and the invoking activity continues in the object from which it made the invocation. Invoking an announcement increases the concurrency in the system, the completion of the evaluation of the body of an announcement decreases the concurrency in the system.

Restricting the model to these forms of invocation ensures that the activity in the system remains a set of activity trees, the structure required by the atomic activity model [APM1004]. In other models, further forms of invocation are sometimes proposed as a cheap form of concurrency where the invoked interface happens to be remote. The ANSA computational model defines the provision of concurrency in such a way that it can be provided cheaply without adding forms of invocation that create problems for the atomicity model.

1. In general a service may be provided by a set of servers acting as a group and used by a number of sets of clients acting as groups. The definitions in this document will be expressed in terms of individual clients and servers. Groups are described in *A Model for Interface Groups* [APM1002].

2.2.1 Requests and responses

A **request** consists of an operation name and zero or more parameters distinguished by position in the request. A request conveys to the server the information that the client has invoked the operation named in the request with those parameters as arguments.

A **response** consists of a termination name and zero or more parameters distinguished by position in the response. A response conveys to the client the information that the evaluation of an operation has completed delivering as its outcome, that termination name and those parameters.

2.2.1.1 Parameter passing

The parameter passing model is defined so as to give the ability to pass the use of a service as was described in Chapter 1. Parameters, both argument and result, are references to interfaces.

The recipient of a parameter gains the ability to refer to an interface and may then invoke the operations in that interface in order to use the service or it may pass on a reference to the interface as a parameter.

Passing the ability to refer to an interface does not imply an ability to determine whether or not the recipient object is already able to refer to that interface nor any kind of intrinsic equality test for interfaces.

2.2.1.2 Alternative outcomes

The termination name in a response distinguishes between alternative outcomes. This is used to report failures in the underlying mechanisms but is also a means by which to deliver different numbers of results or results of different types.

Both the termination name and the ability to deliver more than one result parameter avoid the need for the explicit creation of structures simply in order to deliver several results or to pick one of a number of alternative types of result. A reference to a created structure could be retained by both the sender and the recipient. That reference could then be made visible to other, potentially concurrent activities. This raises the questions of where the structure should be created, whether or not a copy is equivalent to the original, and for how long the structure should be retained. None of these questions is applicable to the response mechanism defined in the ANSA computational model.

Responses have the same structure as requests, reflecting the symmetry of network communication mechanisms.

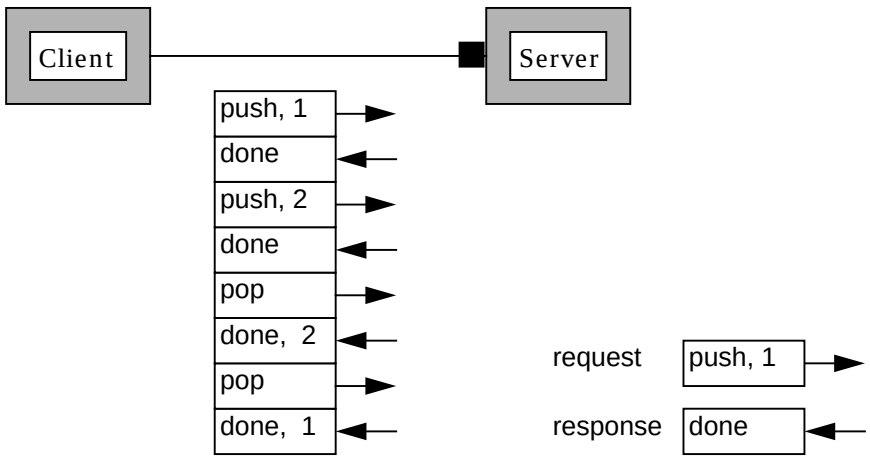
2.2.1.3 Object state

Both the effect of the evaluation of an operation and its outcome may be influenced not only by the request that initiated the evaluation but also by the state of the object that provides the interface containing the operation. The state of the object may be affected by any interaction in which the object participates either as the server or as the client. The state may change while an evaluation is in progress both as a consequence of that evaluation and also as a consequence of other, concurrent evaluations.

The issues arising from concurrent evaluations are discussed in *ANSA Atomic Activity Model and Infrastructure* [APM1004] and *Using path expressions as concurrency guards* [APM1010].

Figure 2.1 illustrates a simple interaction between a client and a server. The server is acting as a stack and the operation names have been chosen to reflect this. The responses that follow the 'pop' requests depend upon the state of the server. In this example,the state changes in the way expected of a stack.

Figure 2.1: Interaction example



2.2.2 Operations in interfaces

2.2.2.1 Interrogations

An **interrogation** is an operation for which the client that invoked the operation receives a response. The response informs the client of the outcome delivered on completion of the evaluation of the operation.

For each invocation of an interrogation, there is one request and one response in the interaction by which the client uses the interface which contains the interrogation. Both the 'push' and 'pop' operations in the example in Figure 2.1 are interrogations.

The implicit pairing of request and response in interrogations models the concept of procedure call and return found in programming languages.

2.2.2.2 Announcements

An **announcement** is an operation for which the client that invoked the operation receives no response. The client is informed neither of the completion of evaluation nor of the outcome delivered.

For each invocation of an announcement, there is one request in the interaction by which the client uses the interface which contains the announcement.

2.2.2.3 Other kinds of interface

The interfaces described in this report contain only interrogations and announcements. An additional kind of interface is necessary to support multimedia; these interfaces, called stream interfaces, require additional work and are not described in this document.

2.2.3 Activity

Operations are invoked by activities.

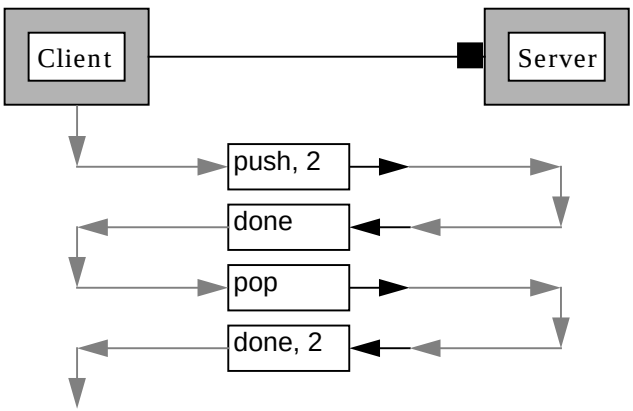
2.2.3.1 Activity passing in interrogations

When an activity invokes an interrogation, the activity passes to the server and evaluates the invoked operation. When the evaluation of an interrogation completes, the activity passes back to the client from which the invocation was made.

Figure 2.2 shows how an activity is passed in an interrogation request, continues in the server and then returns in the response.

The example shows a single activity, multiple activities are described in §2.2.6

Figure 2.2: Activity passing in interrogations



2.2.3.2 Activity creation by announcements

When an activity invokes an announcement, a new activity is started and the original activity continues. The new activity evaluates the announcement.

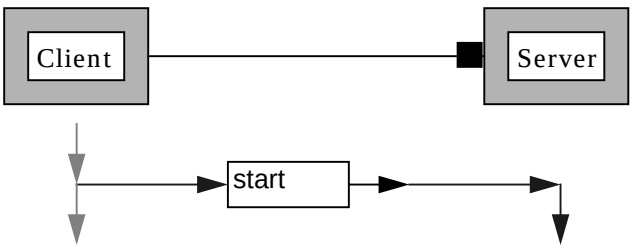
When the evaluation completes, the new activity stops.

Any effect that the evaluation of the announcement may have had on the state of the server will affect the future behaviour of the server. In this respect an announcement is like an interrogation.

An announcement differs from an interrogation in that an announcement has no outcome either to be returned or to be retained for later collection.

Figure 2.3 shows the creation of a new activity by an announcement.

Figure 2.3: Activity creation by an announcement



2.2.4 Sharing an interface

Where more than one object has a reference to an interface, the interface is **shared** by the objects.

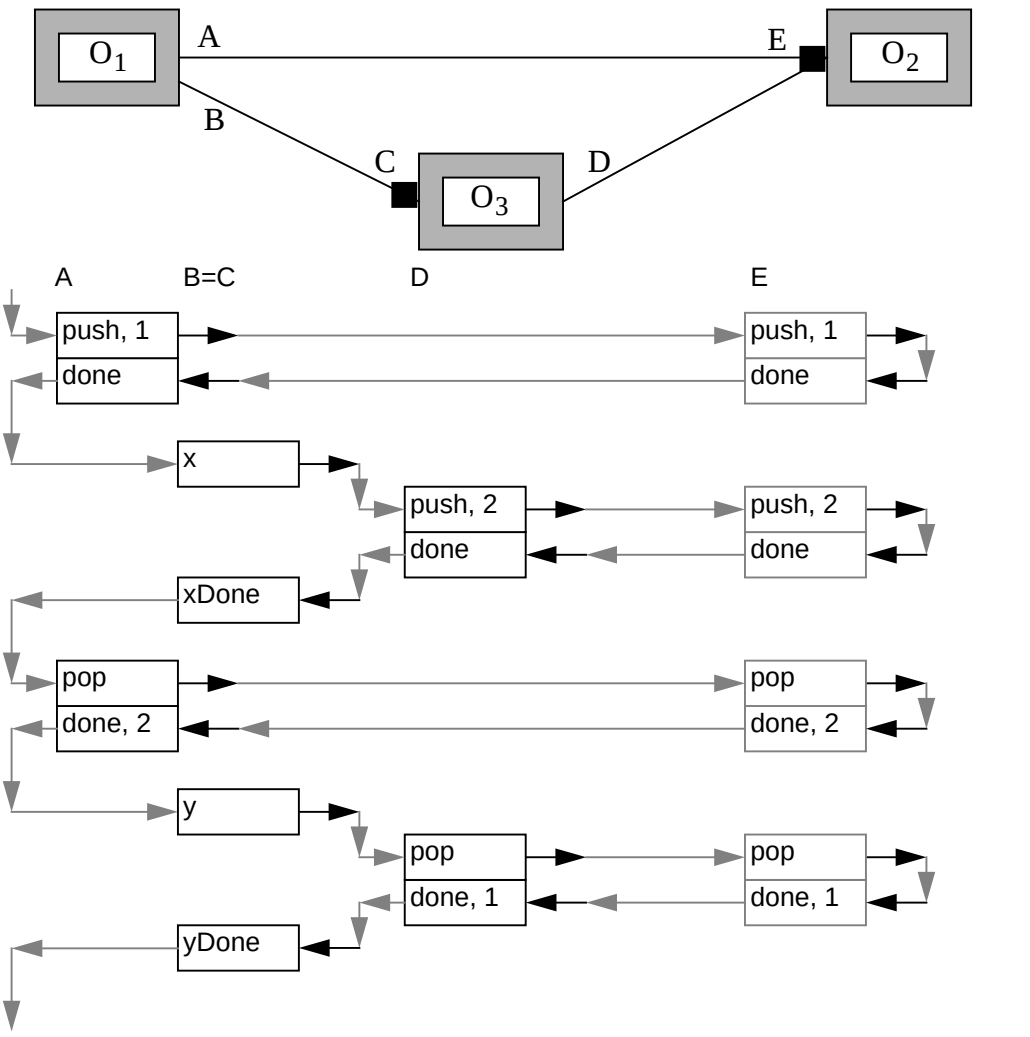
Sharing an interface means that when an operation in that interface is invoked, neither the effect upon the state of the server nor, in the case of an interrogation, the response chosen by the server are affected by which client object was the source of the invocation.

It also means that state changes caused by one client are visible to other clients that use the interface.

2.2.4.1 Sharing of an interface by objects

The only way that an activity can use an interface from more than one object is for the activity to have been passed in an invocation. Figure 2.4 illustrates how different objects observe different parts of the sequence of requests and responses at the points of provision and use labelled A, B, C, D and E in the diagram.

Figure 2.4: Sharing between objects



Each of the three objects participates in two interactions but since the single interface provided by O_2 is used from both O_1 and O_3, O_2 observes a single sequence of requests and responses at E. The responses that it chooses are

required to be those that it would have chosen if all the requests had been made by a single object. Although the interface provided by O_2 behaves as a stack and is seen to behave as a stack by the single activity, neither of the clients O_1 and O_3 observe stack-like behaviour.

2.2.4.2 Client identity problem

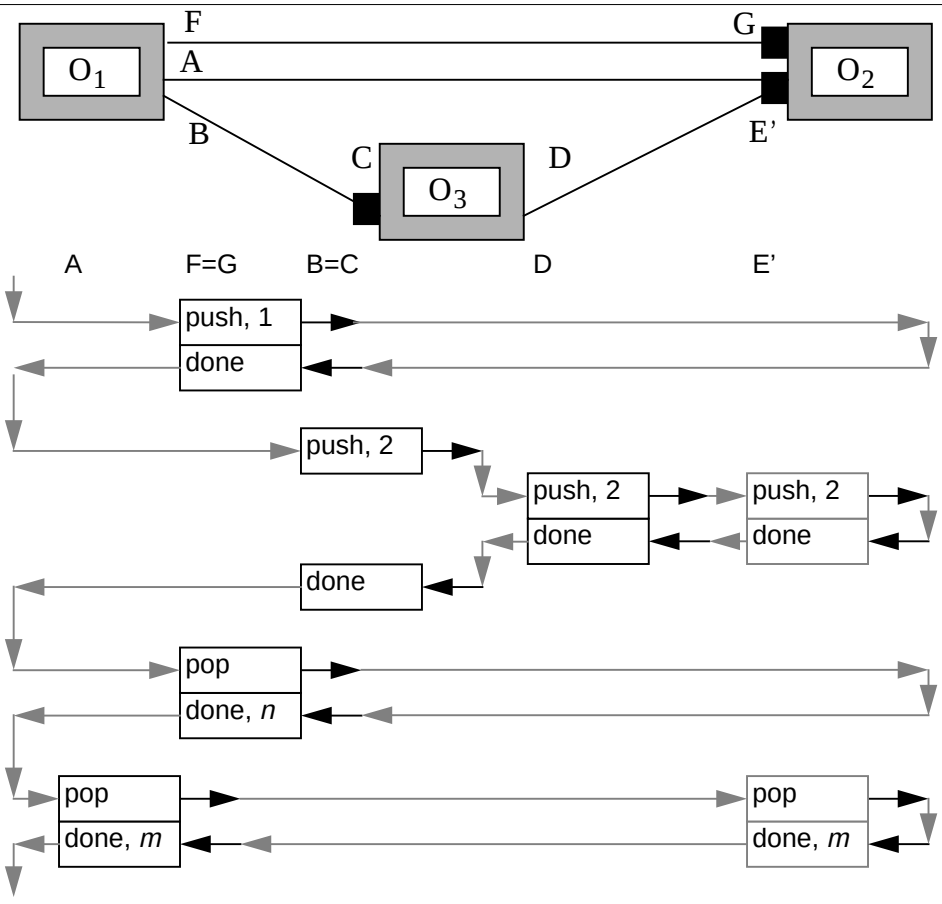
In a distributed system it is often necessary to insert transparency agents. These agents hide details of the mechanisms that are being used to achieve the effect expected by the client and server. The identity of the caller can be misleading under these circumstances.

If some knowledge about the originator of the request is needed then this can be passed as a parameter. The identity of the caller can thus be defined as appropriate to the needs of the application, rather than as an accident of implementation.

2.2.4.3 Multiple interfaces

If O_2 provided two interfaces of which one was used by O_1 and the other by O_3 then it would be necessary to know how the behaviours at those interfaces are related. O_2 might arrange that the two interfaces act as if they were one thus giving the same pattern of responses as in Figure 2.4. O_2 might equally well provide two independent stacks at the two interfaces. Figure 2.5 shows O_2 with two interfaces.

Figure 2.5: Server providing more than one interface



If the two interfaces act as a single stack then n will be 2 and m will be 1. If they act as independent stacks, then n will be 1 and m will be 2. Knowing that

clients are using interfaces provided by the same server object does not in itself give any information about the interactions.

Note that the observations at G are the same as at F for this example and so are not shown separately.

O_3 in Figure 2.5 has been made a simple forwarding agent. Whenever one of its operations is invoked it performs the same invocation on the interface to which it has a reference. In this case, the interactions at A and B are related (whichever version of O_2 is in use) even though O_1 is not only using two different interfaces but also using interfaces provided by different objects.

Knowledge about which object provides an interface is not sufficient to know how that interface is related to any other. Where the relationship between interfaces is significant, the interfaces must designed in such a way that related collections of interfaces can be acquired or the relationship between interfaces can be tested. This is typically done by assigning names to the collections of interfaces and providing operations through which the names can be retrieved.

An interface may have its operations invoked by different activities either from a single client object or from different client objects. As was the case for use by a single activity, the evaluations are not affected by which client objects were the sources of the invocations. The use of an interface by more than one activity is one of the ways in which concurrent evaluations within an object may arise. Multiple activities are the subject of §2.2.6.

2.2.5 Argument and result passing

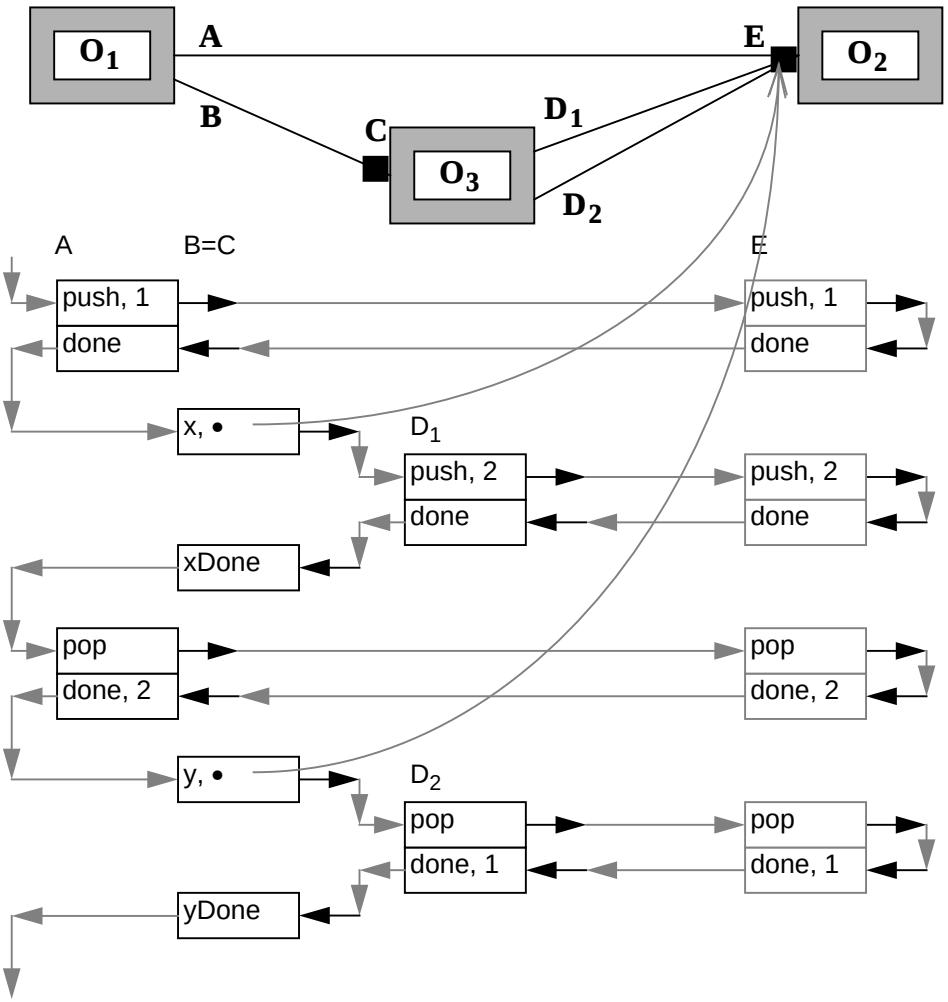
Parameters in requests and responses are references to interfaces. The interfaces passed as parameters are shared by the client and server. The recipient of the request or response gains the ability to refer to the parameter interfaces both as targets for invocations and as parameters in its own requests and responses.

In the example in Figure 2.4, both O_1 and O_3 already had references to the interface provided by O_2. Figure 2.6 illustrates the case where O_3 is told which stack to use.

Both of the requests in the interaction labelled B now include the interface provided by O_2 as a parameter. It is important that the parameter is a reference to the particular interface and is not just an indication of the type of the interface or a reference to the object which provides the interface. The type is not sufficient since there may be many interfaces of that type. A reference to the object is not sufficient since the object may provide more than one interface. A reference to the object together with an indication of the type is still not sufficient since, as was the case for O_2 in Figure 2.5, an object may provide more than one interface of the specified type.

The interaction labelled D has been split into two parts in this example because O_3 need not be aware that the same parameter was passed in both requests. The invocation scheme has been designed so that there is no need to attempt to discover a relationship between these invocations beyond that which is implicit in their having been initiated by the same activity.

Figure 2.6: Interface reference passing



2.2.6 Multiple activities

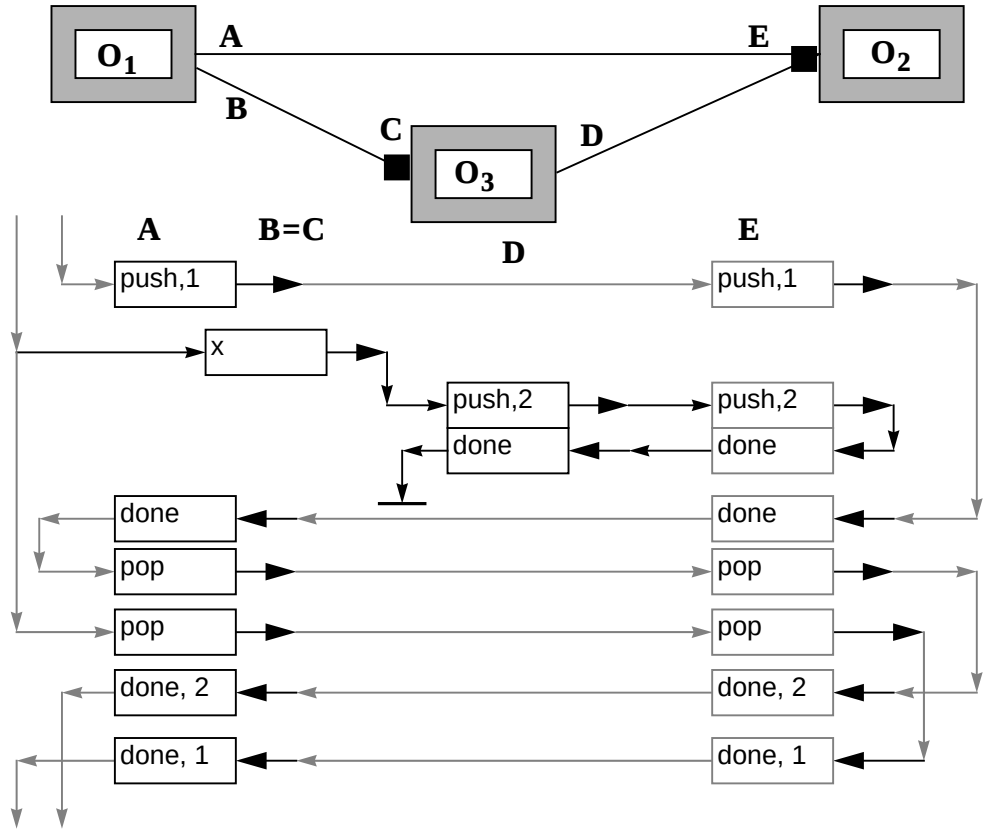
There are a number of additional issues to consider when there is more than one activity in the system. Figure 2.7 illustrates some of the issues that can arise.

There are initially two activities, a third starts when the announcement named 'x' is invoked.

In this example, the sequence observed at E is no longer a sequence of [request / response] pairs. O_2 must be defined in some way that explains how its state will be affected and how it will respond for all the possible sequences. In this example it has been shown as dealing with more than one request at a time but now some care must be taken when describing what it does. If the interaction at E is considered in request order then the behaviour is stack-like but if it is considered in response order then it is queue-like. Neither of the two initial activities observes either stack behaviour or queue behaviour.

The example in Figure 2.7 also shows requests and responses being observed in the same order by both client and server. The two 'pop' requests are shown as in the same order at A and E but there is no fundamental reason why this need be the case.

Figure 2.7: Sharing with multiple activities



An extensive discussion of the issues that arise from concurrency, including the preservation of consistency and the control of visibility between activities is the subject of *ANSA Atomic Activity Model and Infrastructure* [APM1004] and *Using path expressions as concurrency guards* [APM1010].

2.3 Type scheme

The interaction model type scheme defines interface types and response types. It also defines conformance relations for these types.

The type scheme and the conformance relation have been chosen so that knowledge of the type of an interface provides information that is useful to any potential user of the interface in a distributed system, and also so that there is an algorithm for testing conformance between types.

The type scheme does not specify local representations. A remote user of an interface must depend upon the contents of messages which need not reflect the representation preferred by either sender or recipient. Issues of representation, and translation between different representations of the same computational entity, form a significant part of the information and engineering models.

The type scheme does not specify behaviour. Each individual interface has a particular behaviour, but a type scheme which captures useful similarities in behaviour for the required range of cases has not been established. A particular problem occurs where interfaces are shared as was illustrated in Figure 2.7. The behaviour of the server differs from the behaviour observed by

any particular client object, and from the behaviour observed by any particular activity. Matching a client's behaviour requirement against a server's offer is difficult when the client might see only some arbitrary part of the server's behaviour.

2.3.1 Interface types and response types

An **interface type** defines the requests and responses permitted in an interaction where a client uses an interface of that type.

A client using an interface of a given type may make only those requests specified by the interface type. A client that invokes an interrogation must expect any one of the responses specified for that interrogation by the interface type. A client that invokes an announcement must not expect any response.

A server providing an interface of a given type must expect any of the requests specified by the interface type. If an interrogation is invoked, the server must make one of the responses specified for that interrogation by the interface type. If an announcement is invoked, the server must make no response.

An interface type is defined by a set of signatures.

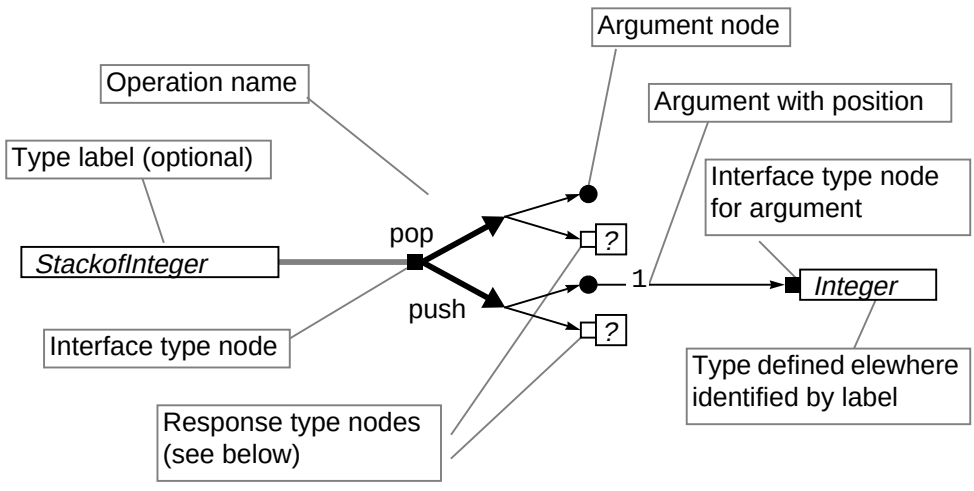
A signature specifies:

- the name of an operation
- the number and interface types of argument parameters
- whether the operation is an interrogation or an announcement
- in the case of an interrogation, the response type for that operation.

No operation in an interface may have the same name as any other operation in that interface. Operations in different interfaces may have the same name.

Types will be defined graphically as illustrated in Figure 2.8 which is part of the definition of the interface type of the stack used in earlier examples.

Figure 2.8: Interface type - graphical representation



Interfaces of the type *StackOfInteger* as defined by the graph must have an operation named pop and an operation named push. The operation named pop has no arguments and is an interrogation but the response type has not been defined in this graph (the label "?" is used to indicate this). The operation named push takes one argument of the type which has been given the label

Integer; the operation is also an interrogation for which the response type has not been defined.

A label such as *StackOfInteger* is not part of the definition of the type; it is there to allow explanatory text such as this to refer to a type defined by a graph. A label of this kind may also be used to show how a number of graphs are to be connected. The label *Integer* is used in this way.

An announcement will be indicated by the absence of an arc leading to a response type node from the head of the signature arc.

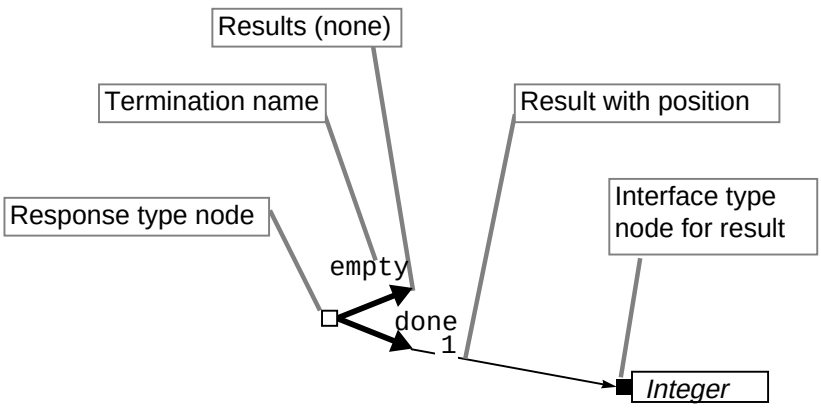
A **response type** defines the set of permitted responses for an interrogation. A response type is defined by a set of terminations.

A termination specifies a termination name and the number and interface types of the parameters that may be delivered in an outcome (and therefore a response) with that termination name.

No termination in a response type may have the same name as any other termination in that response type. Terminations in different response types may have the same name.

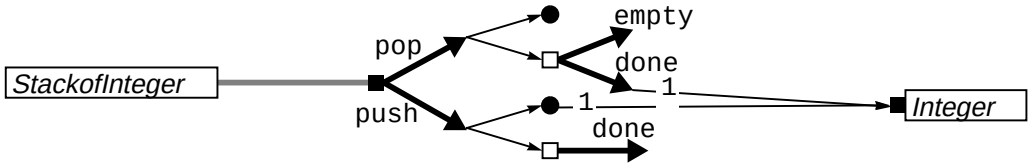
Response types will be defined graphically as illustrated in Figure 2.9.

Figure 2.9: Response type - graphical representation



The response type in Figure 2.9 defines the permitted responses as a termination named empty and no result parameters and the termination named done with a single result parameter of the type which has been labelled *Integer*. This is the appropriate response type for the pop operation of the StackOfInteger type and the graphs can be combined as shown in Figure 2.10.

Figure 2.10: Stack of integer interface type



The graphical notation provides a simple way to deal with recursively defined types, types which appear as arguments or results of their own operations. Some examples of such types can be found in Chapter 5.

2.3.2 Type conformance

2.3.2.1 Definition

Interface type X **conforms to** interface type Y if no interaction errors can arise from the use of an interface of type X as if it were of type Y.

An **interaction error** occurs:

- when a server receives a request which is not in the set of requests defined by the type of the interface provided by the server
- when a client, having invoked an interrogation, receives a response which is not in the set of responses defined by the response type for that interrogation
- when a client, having invoked an interrogation, receives no response
- when a client, having invoked an announcement, receives a response

2.3.2.2 Interpretation

For interface types that are not defined recursively, the conformance relation can be described in terms of signatures and response types as follows:

An interface type X conforms to an interface type Y if:

1. for every signature in Y there is a signature in X which defines an operation of the same name
2. for each signature in Y the signature in X with the same operation name defines an operation with the same number of arguments
3. each interface type in each Y signature conforms to the interface type in the same position in the corresponding X signature
4. for every signature in Y which defines an announcement the signature in X with the same operation name defines an announcement
5. for every signature in Y which defines an interrogation the signature in X with the same operation name defines an interrogation with a response type which conforms to the response type in the Y signature.

A response type X conforms to a response type Y if, for every termination in X:

6. there is a termination with that name in Y
7. for each termination in X the termination in Y with the same name has the same number of parameters
8. each interface type in each X termination conforms to the interface type in the same position in the corresponding Y termination

These rules do not cover the cases where a type refers to itself as the type of an argument or result. In such cases, following these rules leads to situations where a type X conforms to a type Y if X conforms to Y. This case can be resolved by referring back to the original definition of conformance.

Chapter 5 uses examples to show how the rules capture the definition of conformance and gives an alternative formulation of the rules which state when types do not conform rather than when they do conform. It also uses the

graphical notation introduced above to show how conformance is determined when types refer to themselves.

2.4 Summary

2.4.1 Invocation scheme

Interaction between objects is by invoking named operations in interfaces.

Operations are either interrogations or announcements.

An interrogation is invoked in a synchronous request/response style, no change to activity structure.

An announcement is invoked in an asynchronous request only style, new activity is initiated.

Parameters in requests and responses are references to interfaces.

Parameters are passed by sharing the referenced interface.

2.4.2 Type scheme

An interface type defines the requests and responses permitted in an interaction where a client uses an interface of that type.

An interface type is defined by a set of signatures.

A signature specifies:

- the name of an operation
- the number and interface types of argument parameters
- whether the operation is an interrogation or an announcement
- in the case of an interrogation, the response type for that operation.

A response type defines the set of permitted responses for an interrogation. A response type is defined by a set of terminations.

A termination specifies:

- the termination name
- the number and interface types of result parameters

Interface type **X conforms** to interface type Y if no interaction errors can arise from the use of an interface of type X as if it were of type Y.

An **interaction error** occurs

- when a server receives a request which is not in the set of requests defined by the type of the interface provided by the server
- when a client, having invoked an interrogation, receives a response which is not in the set of responses defined by the response type for that interrogation
- when a client, having invoked an interrogation, receives no response
- when a client, having invoked an announcement, receives a response

3 Construction Model

3.1 Overview

The ANSA construction model provides the elements necessary to construct objects that conform to the ANSA interaction model.

In addition to the interaction elements, the ANSA construction model includes some additional elements to provide a computationally complete structure. The model defines evaluation effects and type issues, it does not define concrete representations.

Although the model does not define a representation, in order to write examples, a representation of the model is needed. The definition of each construct includes the definition of a textual notation for use in examples.

A notation for a computational model is inevitably like a programming language, although not necessarily a language that is convenient for programming. In a practical programming language, other issues need to be taken into account. In particular, it must be possible to express ideas that belong in other projections. A description of a programming language based on the ANSA computational model, but which also takes other issues into account, can be found in *DPL Programmers' Manual* [APM1014]

3.2 Elements of the model

The ANSA construction model consists of

- form templates for instantiation as forms to be evaluated by activities
- state that may be observed and modified by the evaluation of forms and
- communication in accordance with the ANSA interaction model.

A form is a semantic element of the ANSA Computational Model, and is equivalent to the notion of a statement or expression in a program. A form template expresses the structure of the form and is the semantic equivalent of a production rule in a grammar for a programming language.

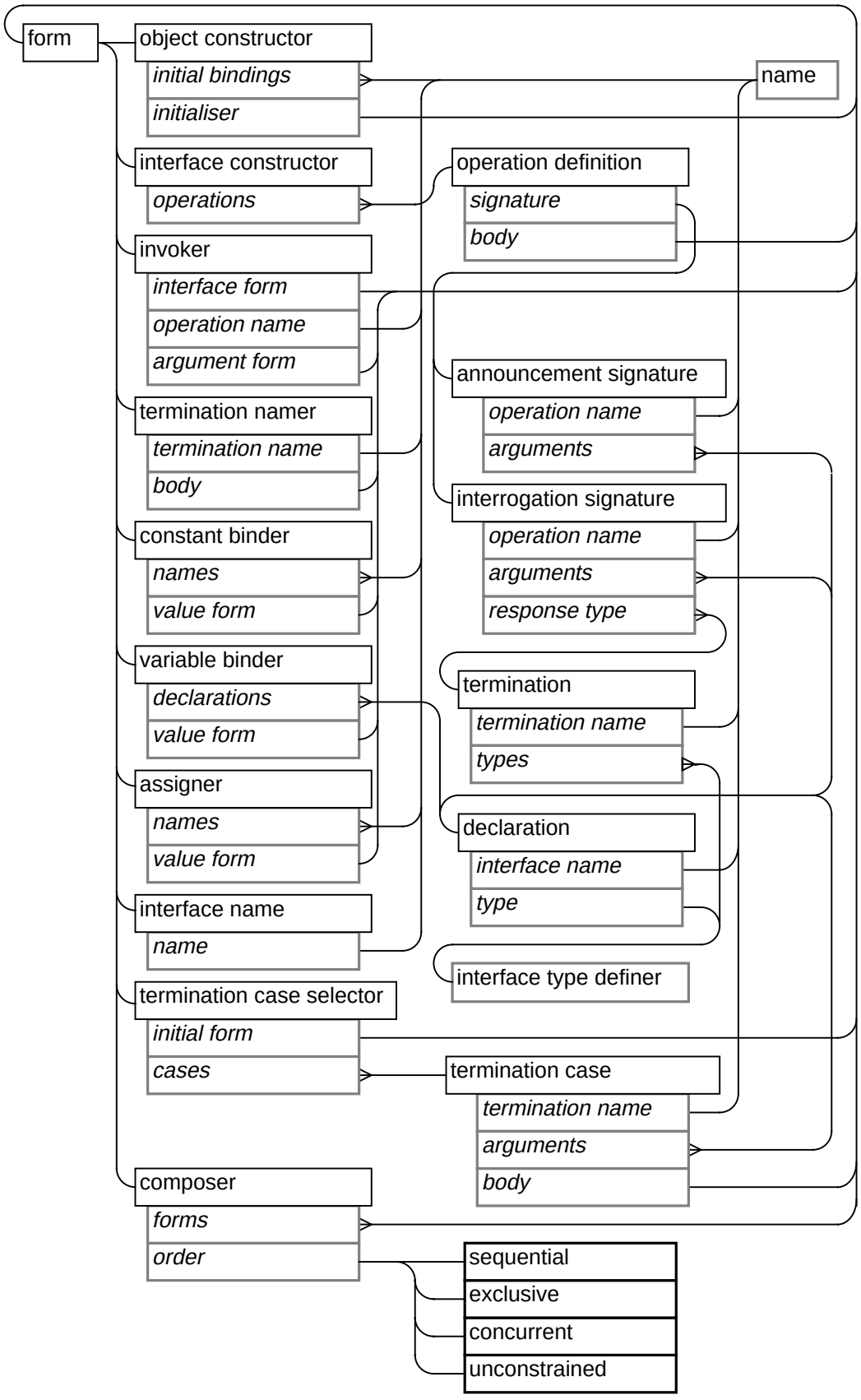
3.2.1 Processing

Each form template defines the components that a form of that kind must have**.** The form templates are defined in §3.3, a summary of the relationships between form templates is given in Figure 3.1 .

Associated with each form template are the rules for evaluating a form of that kind. The evaluation of a form delivers an **outcome**. The evaluation rules define the outcome in terms of the context in which the evaluation takes place, and the outcomes of the evaluations of sub-forms.

The evaluation rules impose some constraints which are expressed as type rules. The type rules form a type scheme. It is important to remember that **the type scheme is derived from the evaluation rules**.

Figure 3.1: Summary of form templates



3.2.2 State

In the ANSA construction model, state is encoded as bindings, each of which is an association of a name with an interface. The names in these bindings may occur in forms; an interface name is a kind of form.

During evaluation, interface names are resolved in naming contexts - sets of bindings - determined by the interface through which the activity performing the evaluation entered the object. For an object to conform to the ANSA construction model, it must be possible to determine from its object constructor form that interface name resolution will not fail during any evaluation that takes place in that object.

Each binding is either constant or variable. In a constant binding, the name always refers to the same interface. In a variable binding, the name may refer to different interfaces at different times.

Objects provide disjoint naming contexts and every binding must be part of an object. The set of objects in a system defines a partitioning of the set of bindings in the system.

The use of a naming model to define state shared between activities, and the ability to change contexts upon invocation by using closures, is an adaptation of the techniques described by Saltzer [SALTZER 78]. The need for separation that arises in a distributed system is satisfied by the use of objects to partition the set of bindings.

The priciples of naming and the terminology are described in *The ANSA Naming Model* [APM1003].

3.2.3 Communication

Communication between objects is by invocations of the kinds permitted by the interaction model.

An object may invoke an operation in an interface provided by another object, passing parameters which are references to interfaces. If the invoked operation is an interrogation then the invoker receives a response consisting of a termination name and parameters.

3.2.4 Types

An interface type is associated with each binding. The interface named in a binding must be of a type which conforms to the associated type.

The type of a form is a response type and every valid form has such a type. The outcome of the evaluation of a form must correspond to one of the terminations in its type. Each form template description includes the rules for deriving the type of a form from the types of its components. For some forms, the type is defined to be the 'most specific' type to which each type in some set of types conforms. This means the type which conforms to every type to which every type in the set conforms.

Some forms contain other forms and their evaluation is defined to use the outcome of the evaluation of the component form. Since both the number and types of results depends upon which termination is chosen, it is necessary to identify which of the possible terminations of the component form is to be used in subsequent evaluation of the containing form. The construction model resolves this issue by requiring that there be a particular termination name used for this purpose. Terminations with this name are known as

anonymous terminations, an outcome with this name is known as an **anonymous outcome**. Terminations that are not anonymous are collectively known as **named terminations**, an outcome that is not an anonymous outcome is known as a **named outcome**.

3.3 Definitions of forms

This section defines each form template under the headings:

Purpose	A short statement of the purpose of the forms of this kind.
Components	A list of components defining the kind of each component and a name for the component in the description. A component that is a form may be any kind of form.
Notation	The example notation syntax for the form template.
Evaluation	The definition of the effect of evaluating the form and the outcome of the evaluation.
Type	The type of the form and any constraints on the types of its components.

In order to provide an unambiguous syntax, the notation introduces the restriction that in some places, only a `composer` form is permitted. This is not a serious restriction since a composer containing only a single form is equivalent to the contained form in terms of evaluation and type.

In some cases the ambiguity can be avoided with a weaker restriction. In order to reduce the number of cases where a form must be wrapped in a composer, forms are divided into two kinds as shown in the prodution rules:

```
form                = assigner | variable_binder |
                      constant_binder | termination_namer |
                      unit | termination_case_selector

unit                = composer | interface_name | invoker |
                      object_constructor|
                      interface_constructor | literal
```

In some cases, a `unit` can be permitted without ambiguity, but other forms could not.

The non-terminal `literal` does not correspond directly to any model form template. It is a means to introduce into the examples, integers and strings that are represented as interfaces in the model.

The representation of integers and strings as interfaces, and forms that construct appropriate interfaces are described in §3.4.

A `literal` can be considered either as a kind of interface constructor, or as a name to which an appropriate interface has been bound, without affecting the interpretation of the rest of the model.

3.3.1 Object constructor

Purpose: To create objects

Components

initial bindings	a list of names
initialiser	a form

Notation

```
object_constructor    =  “object” composer
```

The *initial bindings* need not be written explicitly. All of the names defined outside the object constructor in constant bindings and used in the *initialiser* are included in the list.

For example, in

```
[ n = 1; m = 2; object( interface( one()->(Integer) (n) ) ) ]
```

The *initial bindings* list is <n> and the *initialiser* is

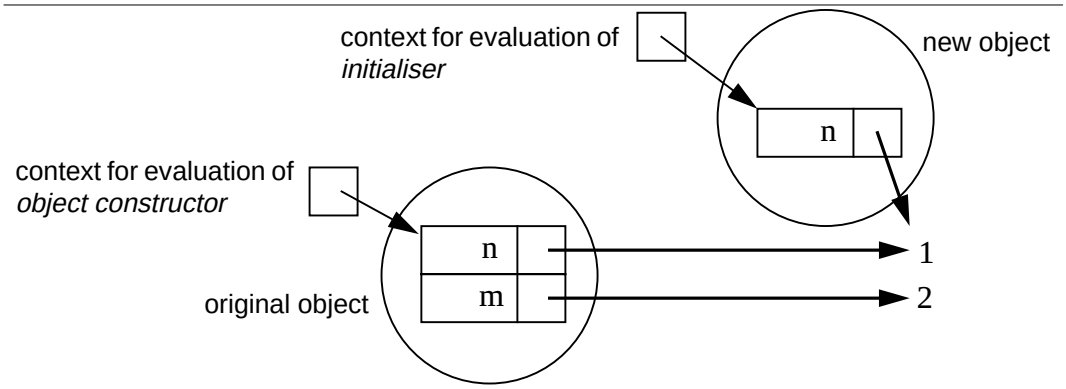
```
( interface( one()->(Integer) (n) ) )
```

which is a composer containing an interface constructor.

Evaluation

Evaluating an object constructor creates a new object which contains constant bindings defined by *initial bindings*. For each name in *initial bindings*, a constant binding is established in the new object, from the name to the interface to which that name resolves in the naming context in which the object constructor is evaluated. Figure 3.2 shows the context in which the object constructor is evaluated and the context in which the initialiser is evaluated.

Figure 3.2: Object constructor contexts



The implicit definition of the *initial bindings* is restricted to constant bindings since the names in the *initialiser* will be resolved through the new constant bindings. Where the original bindings are also constant, the name will always resolve to the same interface in either context and so forgetting that a new binding has been created will not have serious consequences.

When the new object has been created and the initial bindings established, the *initialiser* is evaluated in the context defined by the new object. The outcome

of the evaluation of an object constructor is the outcome of the evaluation of its *initialiser*.

In the example above, evaluating the object constructor creates a new object in which the name n is bound to the interface that represents 1 because n will be bound to 1 in the context in which the object constructor is evaluated. The *initialiser* form is then evaluated in the newly created context.

Evaluating the initialiser creates a new interface which refers to the n in the new context. In this case, the outcome of the evaluation of the interface constructor, the composer that contains it and the object constructor itself is anonymous with only a reference to the new interface.

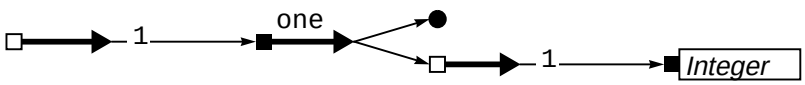
The establishment of the *initial bindings*, the evaluation of the *initialiser* and the delivery of the outcome are like the passing of arguments, evaluation of body and delivery of outcome for an interrogation. The evaluation of an object constructor is equivalent to the invocation of a 'create' (or 'new') operation in an interface to a factory for that kind of object.

Type

The type of an object constructor is the type of its *initialiser*.

In the example above, the type of the interface constructor, the composer used as the *initialiser* and the object constructor itself would be defined graphically as shown in Figure 3.3.

Figure 3.3: Type of example object constructor



Where this type appears (as part of a signature) it would be written as:

```
->( type( one()->(Integer) ) )
```

3.3.2 Interface constructor

Purpose: To create interfaces

Components

operations	a set of operation definitions
concurrency guard	a rule defining when operations may be activated

The components of each operation definition are:

signature	a signature
body	a form

A signature is either an interrogation signature or an announcement signature.

The components of an interrogation signature are:

operation name	a name
arguments	a list of declarations
response type	a set of terminations

The components of an announcement signature are:

operation name	a name
arguments	a list of declarations

The components of a declaration are:

interface name	a name
type	an interface type definer

The components of a termination are:

termination name	a name
types	a list of interface type definers

The effect of the *concurrency guard* is defined below in the description of the invoker form template. The principles of concurrency control and the elements necessary to define a concurrency guard are discussed in *Using path expressions as concurrency guards* [APM1010]

Interface type definers are not evaluated or operated upon by any of the forms defined in this chapter. Chapter 4 describes the use of interfaces as interface type definers, and extends the basic model to include an interface type defining form, type parameters and polymorphism.

Notation

```
interface_constructor =  “interface” “(” [ “path” “=” path-expr ]
                                        {signature body}
                                    “)”
signature             =  operationName arguments { termination }
body                  =  composer
arguments             =  “(” {declaration} “)”
termination           =  “->” [TerminationName] “(”
                         {type_definer} “)”
declaration           =  name {name} “:” type_definer
```

The notation includes an abbreviated syntax for a list of declarations with the same interface type definer. The syntax for concurrency guards is described in [APM1010].

Evaluation

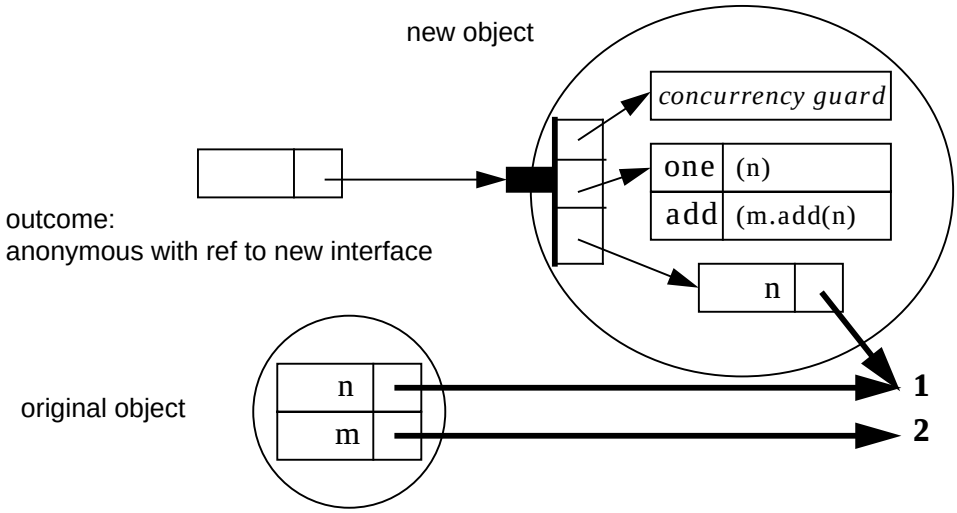
Evaluating an interface constructor creates a new interface to the object in which the evaluation takes place. The evaluation delivers an anonymous outcome with a reference to a single interface, the newly created interface.

The interface consists of the concurrency guard for the interface, a mapping from the operation names to the forms provided as bodies, and the naming context in which the interface constructor is evaluated.

Figure 3.4 shows the interface created by evaluating the interface constructor in

```
[ n = 1 ; m = 2
; object( interface( one()         ->(Integer) (n)
                     add(m:Integer)->(Integer) ( m.add(n) ) )
]
```

Figure 3.4: Interface construction



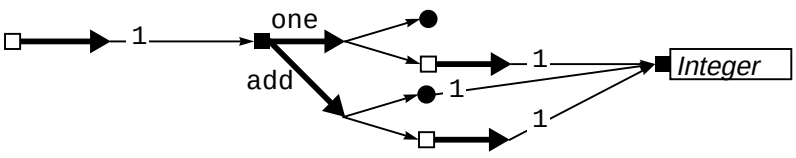
Evaluating an interface constructor creates a set of closures and a mapping from operation names to the closures. The common environment for all of these closures can then be factored out into the interface instance as illustrated in Figure 3.4. The evaluation of operation bodies is included in the description of the invoker form template.

Type

The type of an interface constructor has only an anonymous termination with a single interface type. That interface type is the type defined by the signatures in the operation definitions in the interface constructor.

In the example above, the type of the interface constructor would be defined graphically as shown inFigure 3.5

Figure 3.5: Type of example interface constructor



and written as:

```
->( type( one()->(Integer) add(m:Integer)->(Integer) ) )
```

For each operation definition, if *signature* is an interrogation signature then the operation is an interrogation and the type of the body must conform to the type specified by *response type* in the *signature*. If *signature* is an announcement signature then the operation is an announcement and the type of the body must conform to the type that includes only an anonymous termination with no results.

3.3.3 Invoker

Purpose: To invoke operations

Components

interface form	a form
operation name	a name
argument form	a form

Notation

```
invocation          =  unit "." operationName composer
```

Evaluation

The evaluation of an invoker begins with the evaluation of one of its component forms.

If the evaluation of that component form delivers a named outcome, then the evaluation of the invoker is complete and delivers that named outcome.

Otherwise the other component form is evaluated[1]. If that evaluation delivers a named outcome then the evaluation of the invoker is complete and delivers that named outcome.

If the evaluations of both component forms deliver anonymous outcomes then an invocation takes place.

The anonymous outcome delivered by the evaluation of the *interface form* refers to an interface. This is the interface to which the invocation request is sent.

The invocation request consists of *operation name* and argument parameters which are the references to interfaces delivered as the anonymous outcome of the evaluation of the *argument form*. The order of the interfaces in the outcome is the order in which they appear as arguments.

The arrival of the invocation request at an interface is the request event[2] for invocation of the operation named in the request. The activate event occurs after the request event at a time determined by the *concurrency guard* for the interface. After the activate event, the *body* of the operation is evaluated in the naming context defined by the interface, augmented by constant bindings of the argument names in the *signature* of the operation to the parameters in the request. The terminate event of the invocation occurs when the evaluation of the body is complete and delivers its outcome.

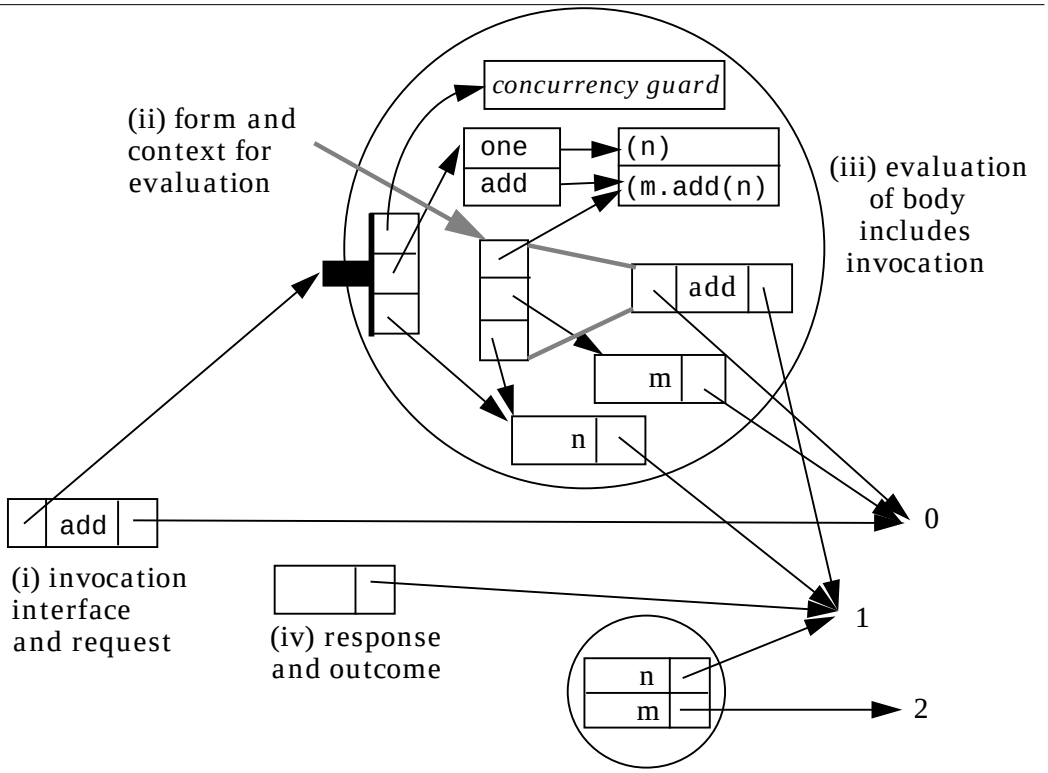
Figure 3.6 extends the previous example to show invocation of an operation in the newly created interface. The whole of the previous example is now used as the *interface form* of the invoker.

```
[ n = 1 ; m = 2
; object( interface( one()          ->(Integer) (n)
                     add(m:Integer)->(Integer) ( m.add(n) ) )
].add(0)
```

1. Note that this evaluation order constraint is the one named ‘exclusive’ in the definition of ‘composer’ in §3.3.10

2. Request, activate and terminate events are described in more detail in *Using path expressions as concurrency guards* [APM1010].

Figure 3.6: Invoking an interrogation



1. The interface form has been evaluated giving a reference to the target interface. The argument form has been evaluated giving a reference to the interface that represents 0. The request sent to the target interface consists of the name `add` and the reference to 0.
2. The form named in the request is selected for evaluation. The naming context consists of the naming context in the interface - a binding of `n` to 1 - and a new constant binding of `m`, the argument name from the signature, to 0 which was passed as a parameter.
3. During the evaluation of `(m.add(n))`, `m` resolves to 0 and `n` to 1 in the evaluation of the invoker in the body of the operation.
4. The outcome of the evaluation of the inner invoker - effectively `(0).add(1)` - is anonymous with a reference to 1. This outcome defines the response to the invocation and the outcome of the outer invoker.

The bindings of the arguments of an invocation do not become visible to other concurrent invocations either of this operation or of any other operation. This is achieved by organising the naming context as a list of sets of bindings and adding the argument bindings as a new set at the front of the list so that a binding of a name as an argument will be found in preference to any binding of the name in the naming context defined by the interface.

This definition allows the common naming context for an interface to be factored out even where operations use names for their arguments that are bound in the common context. The definition also covers the case of recursive invocations of an operation.

If an interface constructor is evaluated during the evaluation of the body of an operation then the naming context captured in the constructed interface is the whole of the list of sets of bindings that is the context for the evaluation.

If the invoked operation is an interrogation then the evaluation is performed by the activity which is evaluating the invoker and the outcome of the evaluation of the *body* defines the response and, in the absence of errors, is the outcome of the evaluation of the invoker.

If the invoked operation is an announcement then the evaluation is performed by a new activity which, when the evaluation of the *body* is complete, discards the outcome and stops. The outcome of the invoker may be delivered without waiting for any synchronisation with the evaluation of the body, and, in the absence of errors, is anonymous with no results.

Engineering failures can become visible at invocations and are reported as named outcomes. Engineering failures include exhaustion of resources and failure of equipment.

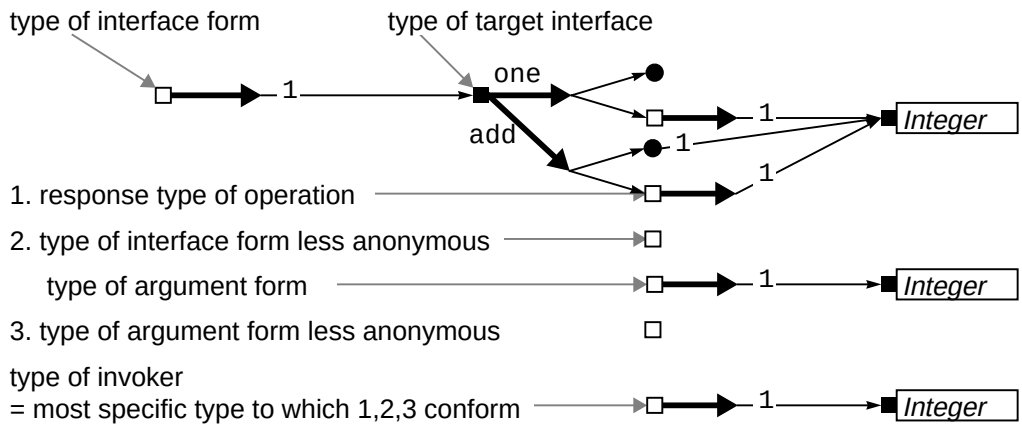
Type

The type of an invoker is the most specific type to which the following types conform:

- the response type of the operation
- the type of the *interface form* less its anonymous termination
- the type of the *argument form* less its anonymous termination

Figure 3.7 illustrates the type of the invoker for the example above.

Figure 3.7: Type of example invoker



The anonymous termination in the type of the *interface form* must have exactly one interface type. That interface type must have a signature containing *operation name*. The anonymous termination in the type of the *argument form* must have the same number of interface types as the signature, and each argument interface type must conform to the corresponding signature interface type.

3.3.4 Termination namer

Purpose: To name an outcome.

Components

termination name	a name
body	a form

Notation

```
termination_namer      =  "->" name composer
```

Evaluation

The *body* is evaluated. If the outcome of the evaluation is named, then that is the outcome of the evaluation of the termination namer.

If the outcome is anonymous then the outcome of the evaluation of the termination namer has the name *termination name* and the interface references in the anonymous outcome of the evaluation of the *body*, in the same order.

For example, the outcome of `->x(5,6)` has the name `x` and references to 5 and 6 because the outcome of `(5,6)` is anonymous with references to 5 and 6. The outcome of `->ignored(->x(5,6))` has the name `x` and references to 5 and 6 because the outcome of the *body* of the outer termination namer is named.

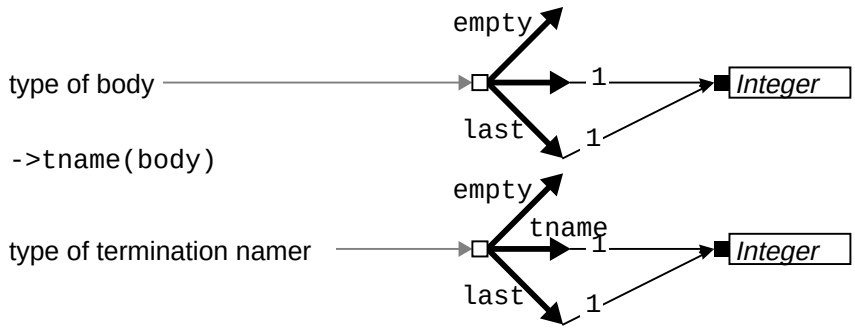
Type

The type of a termination namer is the most specific type to which the following types conform:

- the type of *body* less its anonymous termination
- the type that contains only the termination with name *termination name* and the same list of interface types as the anonymous termination in the type of *body*.

Figure 3.8 illustrates how the type of a termination namer is derived from the type of its body .

Figure 3.8: Type of termination namer



3.3.5 Constant binder

Purpose: To create constant bindings

Components

names	a list of names
value form	a form

Notation

```
constant_binder        =  name {name} "=" form
```

Evaluation

The *value form* is evaluated. If the outcome is anonymous then constant bindings of the names in *names* to the interfaces referenced in the outcome are added to the set of bindings at the front of the list that is the naming context.

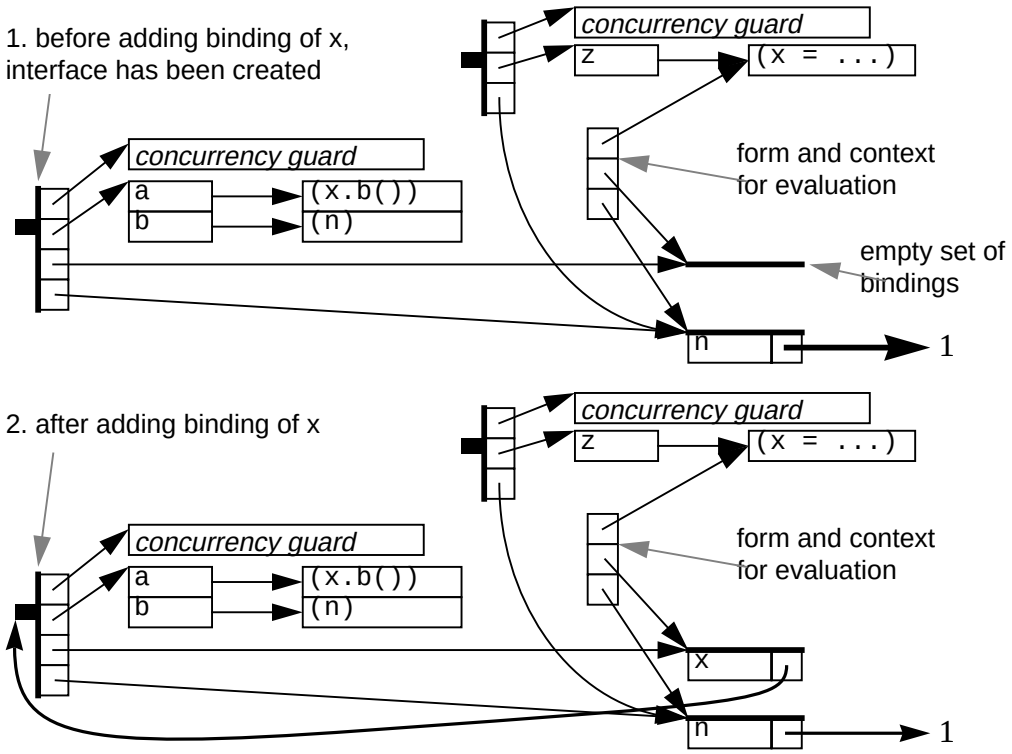
The outcome of the evaluation of a constant binder is the outcome of the evaluation of its *value form*.

Adding bindings to an existing set allows the body of an operation to refer to the interface that contains the operation.

Figure 3.9 illustrates the construction of an interface that refers to itself during the evaluation of:

```
[ n = 1
; interface( z()->(type(a()->(Integer) b()->(Integer)))
            ( x = interface( a()->(Integer) (x.b())
                             b()->(Integer) (n)    )))
].z()
```

Figure 3.9: Constant binder example



Type

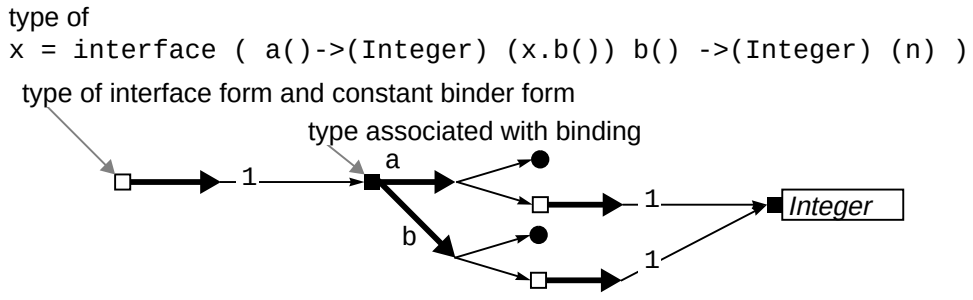
The type of a constant binder is the type of its *value form*.

The anonymous termination in the type of the *value form* must have the same number of interface types as there are names in *names*.

The interface type associated with each binding created by a constant binder is the interface type in the position in the anonymous termination in the type of the *value form* that is the same as the position of the name in *names*.

Figure 3.10 illustrates the type of a constant binder and the type associated with the binding for the same example as in Figure 3.9.

Figure 3.10: Constant binder types



3.3.6 Variable binder

Purpose: To create variable bindings

Components

declarations	a list of declarations
value form	a form

The structure of a declaration is defined in section 3.3.2.

Notation

```
variable_ binder      =  declaration {declaration} ":=" form
```

Evaluation

The *value form* is evaluated. If the outcome is anonymous then variable bindings of the names in *names* to the interfaces referenced in the outcome are added to the set of bindings at the front of the list that is the naming context. The outcome of the evaluation of a constant binder is the outcome of the evaluation of its *value form*.

This is the same as for a constant binder except that variable bindings are created.

Type

The type of a variable binder is the type of its *value form*.

The interface type associated with each binding created by a variable binder is the type specified by the *type definer* in the declaration corresponding to that binding.

The anonymous termination in the type of the *value form* must have the same number of interface types as there are declarations in *declarations*. Each interface type in the anonymous termination in the type of the *value form* must conform to the type specified by the *type definer* in the declaration in the corresponding position in *declarations*.

3.3.7 Assigner

Purpose: To change which interfaces are bound in variable bindings.

Components

names	a list of names

value form a form

Notation

```
assigner          =  name {name} ":=" form
```

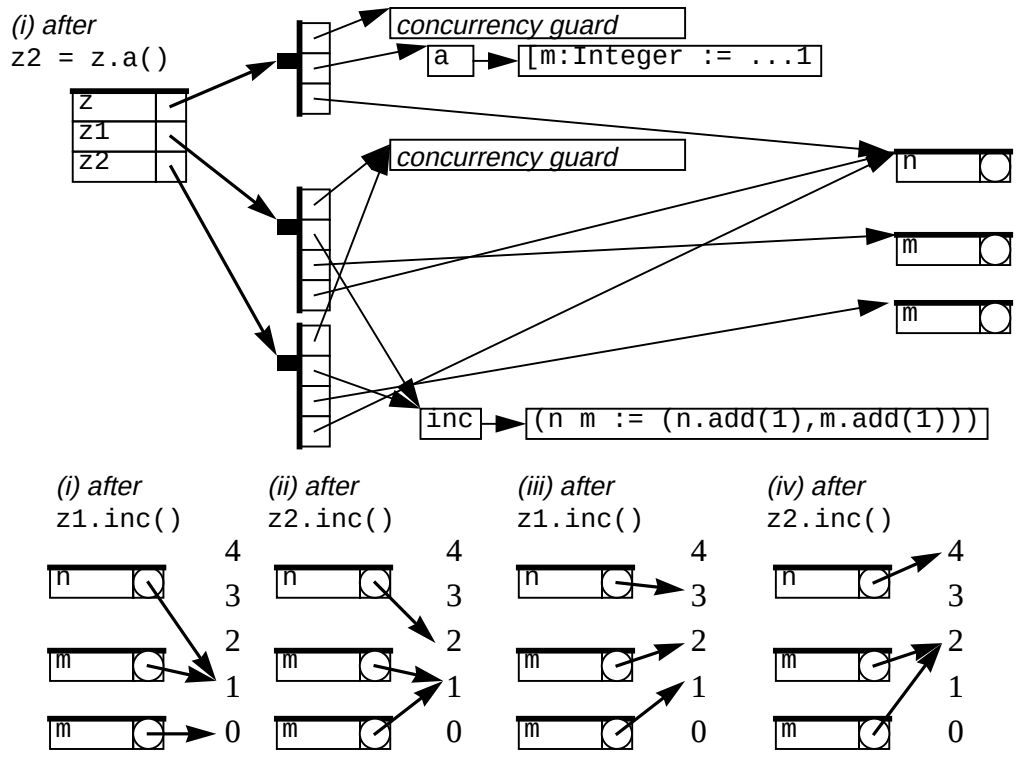
Evaluation

The *value form* is evaluated. If the outcome is anonymous then the variable bindings of the names in *names* are changed to refer to the interfaces referenced in the outcome. The outcome of the evaluation of an assigner is the outcome of the evaluation of its *value form*.

Figure 3.11 illustrates how the changing of existing bindings allows state to be shared between interfaces as well as there being state private to a particular interface.The example is of a generator for counters that also maintain a total for all counters created by the generator:

```
[ z = [ n:Integer := 0; % the total - shared by all
        interface ( a()->( type(inc()->(Integer Integer)) )
                  [ m:Integer := 0; % count - distinct for each
                      interface
                        ( inc()->(Integer Integer)
                          (n m := (n.add(1),m.add(1))))])];
  z1 = z.a();
  z2 = z.a();
  z1.inc(); % returns 1 1
  z2.inc(); % returns 2 1
  z1.inc(); % returns 3 2
  z2.inc()  % returns 4 2
]
```

Figure 3.11: Assigner example



This example also illustrates that different interfaces created by the same interface constructor may share the mappings from operation names to body forms and therefore the body forms as well. The differences in effect between the different interfaces requires only that the naming contexts include different sets of bindings.

Type

The type of an assigner is the type of its *value form*.

The anonymous termination in the type of the *value form* must have the same number of interface types as there are names in *names*. Each interface type in the anonymous termination in the type of the *value form* must conform to the type associated with the binding having the name which occurs in the corresponding position in *names*. The bindings of the names in *names* must be variable.

3.3.8 Interface name

Purpose: To refer to the interface bound to a name.

Components

name	a name

Notation

```
interface_name        =  name
```

Evaluation

Evaluating an interface name delivers an anonymous outcome which refers to one interface. The interface is the one bound to *name* in the naming context in which the interface name is evaluated, when the evaluation takes place.

Interface name forms have appeared in all of the examples above, a study of those examples will show that this evaluation rule gives the effects already described.

Type

The type of an interface name has only an anonymous termination with a single interface type which is the type associated with the binding of *name* in the naming context in which the interface name can be evaluated.

3.3.9 Termination case selector

Purpose: To choose a form to evaluate by termination name

Components

initial form	a form
cases	a set of termination cases

The components of a termination case are:

termination name	a name
arguments	a list of declarations
body	a form

Notation

```
termination_case_sele =  “after” composer “handle”
ctor                     “(“ { termination_case } ”)”

termination_case      =  terminationName arguments composer
```

Evaluation

The *initial form* is evaluated.

If the outcome of the evaluation of the *initial form* is anonymous then that is the outcome of the termination case selector.

If the outcome of the evaluation of the *initial form* is named and the termination name in the outcome is not a *termination name* in a termination case in *cases* then that outcome is the outcome of the termination case selector.

If the outcome of the evaluation of the *initial form* is named and the termination name in the outcome is a *termination name* in a termination case in *cases* then the body of that termination case is evaluated.

The evaluation is performed in the naming context in which the termination case selector is being evaluated augmented with constant bindings of the interfaces referenced in the outcome from the evaluation of the *initial form* to the names in the declarations in the *arguments* of the termination case.

The evaluation of the termination case selector is complete when the evaluation of the *body* is complete and the outcome from the evaluation of the *body* is the outcome of the evaluation of the termination case selector.

The following examples show termination case selector forms and equivalent, simpler forms for the three cases described above.

```
after (1) handle (x(n:Integer) (n.add(1)))      ≡ (1)
after (->y(1)) handle (x(n:Integer) (n.add(1))) ≡ (->y(1))
after (->x(1)) handle (x(n:Integer) (n.add(1))) ≡ (1.add(1))
```

Type

The type of a termination case selector is the most specific type to which the following types conform:

1. the type of each *body* in a termination case in the *cases*
2. the type of the initial form less those terminations with names that appear as a *termination name* in a termination case in the *cases*

Figure 3.12 illustrates the types of the forms in the following example which contains two termination case selectors:

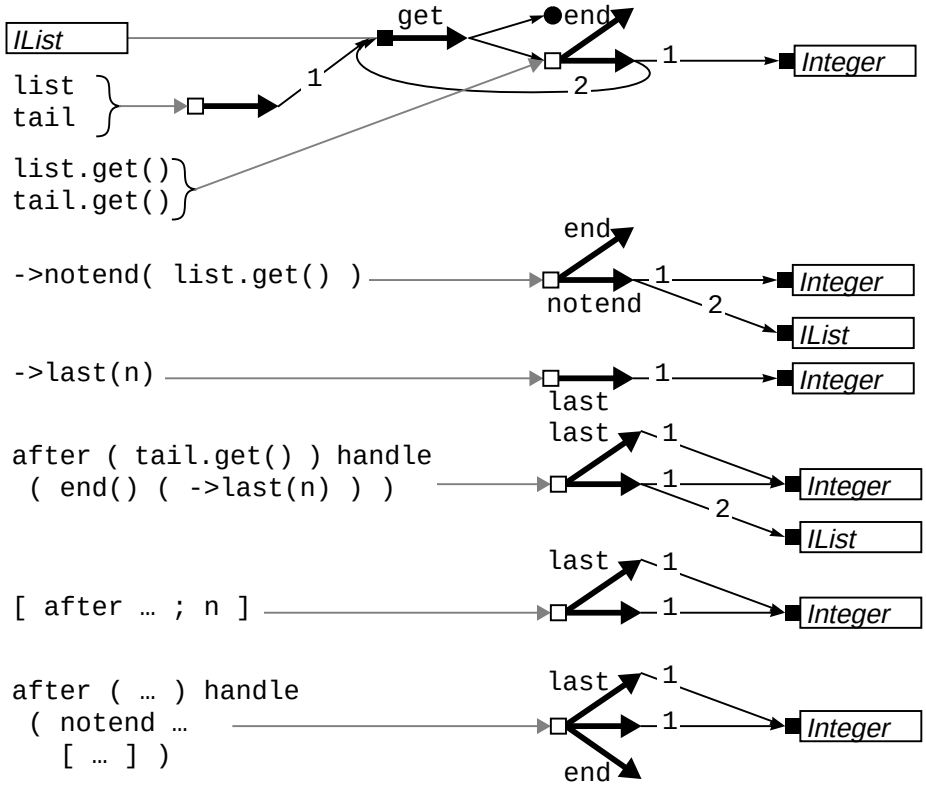
```
after ( ->notend( list.get() ) ) handle
( notend(n:Integer tail:IList)
    [ after (  tail.get() ) handle ( end() ( ->last(n) ) )
    ; n
    ]
)
```

This example would be a type correct body for an operation with signature

```
get(list:Ilist) ->(Integer) ->last(Integer) ->end()
```

Each termination in the type of the *initial form* with a name that appears as a *termination name* in a termination case in the *cases* must have the same

Figure 3.12: Type of example termination case selector



number of interface types as there are declarations in the *arguments* in the termination case. Each interface type in the termination must conform to the type defined in the declaration in the corresponding position in the *arguments*.

3.3.10 Composer

Purpose: To evaluate more than one form and combine the outcomes

Components

forms	a list of forms
order	evaluation order definer

The evaluation order definer specifies one of the following cases:

sequential	one at a time in list order
exclusive	one at a time in any order
concurrent	as separate sub-activities
unconstrained	

Notation

```
composer    =  "(" [formList] ")"
formList    =  form [ {";" form} | {","form}
                    | {"||" form} | {"|" form} ]
```

The evaluation order is determined by the separator:

sequential	;
exclusive	,
concurrent	||
unconstrained	|

The alternative kind of composer with square brackets which has been used in the examples is described in the next section.

Evaluation

If there are no forms in the list then the evaluation delivers an anonymous outcome with no results. All the evaluation orders are equivalent in this case.

If *order* is *sequential* then the forms in *forms* are evaluated in the order in which they appear in *forms* until the evaluation of a form in *forms* delivers a named outcome, or all forms in *forms* have been evaluated and delivered anonymous outcomes.

If *order* is *exclusive* then the forms in *forms* are evaluated one at a time until the evaluation of a form in *forms* delivers a named outcome or all forms in *forms* have been evaluated and delivered anonymous outcomes. The order in which the forms are evaluated is not defined and may vary between evaluations of the composer but is not required to do so.

If *order* is *concurrent* then the forms in *forms* are evaluated by sub-activities of the activity that is evaluating the composer. The evaluation of the composer is complete as soon as any of the evaluations have delivered a named outcome, or when all evaluations have delivered anonymous outcomes. A sub-activity may wait at a concurrency guard, the activity evaluating a concurrent composer is considered to be waiting if all of its sub-activities are waiting. Any activity (or sub-activity) which is not waiting is permitted to progress; if there is at least one activity that is not waiting then at least one activity must progress.

If *order is unconstrained* then the forms in *forms* are evaluated with no constraints upon the order in which they are evaluated. In particular, any of the other evaluation orders or any combination of those orders may be used.

If the outcome of the evaluation of a form in *forms* is named then that outcome is the outcome of the evaluation of the composer. If this occurs where *order* is *concurrent*, the other evaluations are orphaned and may or may not continue. The Atomic Activity Model [APM1004] describes orphans in the context of loss of contact. It also describes mechanisms which may be deployed to ensure that orphans do not interfere with the continuing activity or with other activities. Since *unconstrained* includes the possibility of *concurrent*, these considerations apply in that case as well.

If all of the forms in *forms* delivered anonymous outcomes then the outcome of the evaluation of the composer is anonymous. This anonymous outcome contains all of the interfaces from the anonymous outcomes of the evaluations of the forms with the order within outcomes preserved and with every interface from the evaluation of each form before those from evaluations of forms later in the list.

The definition of concurrent evaluation has been chosen so as to permit the use of multiple processors for the activities that are not waiting but also to permit the use of a single processor which is dedicated to a single activity until it has to wait at a concurrency guard.

The following examples illustrate the different evaluation orders.

```
( z:Integer := 1 ; ( z := z.add(1) ; z := z.subtract(1) ) ; z )
```

Sequential evaluation - outcome must be as for (1,2,1,1).

```
( z:Integer := 1 ; ( z := z.add(1) , z := z.subtract(1) ) ; z )
```

Exclusive evaluation - the assigner forms are evaluated one at a time but in either order. The outcome must be either (1,2,1,1) or (1,1,0,1). Note that the result of the addition must be the second element of the outcome, and the result of the subtraction must be third element of the outcome, whichever order is used for evaluation.

```
( z:Integer := 1 ; ( z := z.add(1) | z := z.subtract(1) ) ; z )
```

Unconstrained evaluation - the assigner forms may be evaluated one at a time as for exclusive evaluation or they may be evaluated concurrently.

Use of unconstrained or concurrent evaluation is not recommended in this case since both assigner forms both resolve the name z and update its binding. Most implementations will do no worse than delivering an outcome of either (1,2,0,2) or (1,2,0,0) but the simultaneous updating of the binding by two activities could have unpredictable effects. This issue of evaluation granularity is discussed in *Using path expressions as concurrency guards* [APM1010]

Unconstrained evaluation is appropriate in cases where the forms do not share state or where such sharing is incidental and protected by a concurrency guard. Unconstrained evaluation allows the implementation to use several processors if they are available but also permits a simple fixed order of evaluation.

```
( (x.a(); x.a()) || (x.b(); x.b()) )
```

Concurrent evaluation - if one of the forms waits at a concurrency guard the other must be allowed to proceed. Suppose the interface bound to `x` has a concurrency guard that requires that the operations `a` and `b` be evaluated alternately. Specifying concurrent evaluation order means that the first `x.a()` will be evaluated followed by the first `x.b()`, then the second `x.a()` and finally the second `x.b()` (assuming that no other activities are invoking the operations). In the absence of other activities, sequential or exclusive evaluation would not complete and unconstrained evaluation might not complete.

Type

The type of a composer is the most specific type to which the following types conform:

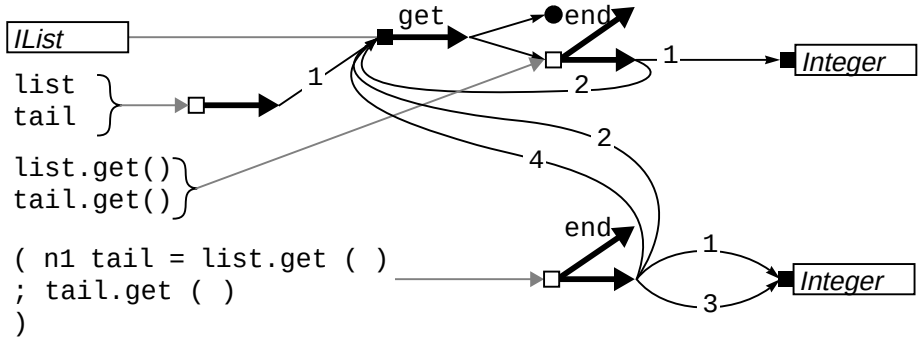
1. the type of each form in *forms* less its anonymous termination
2. the type with only an anonymous termination which has as its list of interface types the concatenation of the lists of interface types of the

anonymous terminations of the types of the forms in *forms* in the order defined by *forms*.

Figure 3.13 illustrates the derivation of the type of:

```
( n1 tail = list.get() ; tail.get() )
```

Figure 3.13: Type of example composer



3.3.11 Expansion of [...]

The examples above have included an alternative kind of composer enclosed in square brackets. That kind of composer delivers the anonymous outcome of the last form in the form list rather than concatenating the outcomes. The full production rule for `composer` is:

```
composer              =  "(" [formList ")" |  "[" [formList] "]"
```

Such composers do not need an explicit counterpart in the model since it is always possible to capture the anonymous outcome of the composer and deliver an anonymous outcome equivalent to that of the last form in the list.

For example,

```
[ x=1; y="Hello"; (x.add(1), y) ]
```

Can be expressed as

```
( after ( ->void( x=1; y="Hello"; (x.add(1),y) ) ) handle
  ( void(f1r1:Integer f2r1:String f3r1:Integer f3r2:String)
    (f3r1, f3r2) ) )
```

The termination name must not be the name of a termination of the form being expanded. The names in the list of declarations are arbitrary, here they have been chosen to suggest which form and which result they will be bound to. The types in the list of declarations are those in the anonymous termination of the form being expanded.

3.3.12 Stream interface manipulation

Stream interfaces were mentioned in §2.2.2, but none of the form templates described above provides any way to manipulate them.

It has been proposed that there should be an "explicit binding" template which allows stream interfaces to be connected.

The issues involved in adding such a form template are for futher study.

3.3.13 Activity creation

A new activity can be created by invoking an announcement.

An alternative way to create activities would be through a form provided specially for the purpose. Such a form might be defined as follows:

Components

body	a form to be evaluated by a new activity

Notation

```
activity_constructor  =  “activity” composer
```

Evaluation

The body is evaluated by a new activity, and the outcome of the evaluation is discarded. The outcome of the activity constructor is anonymous with no results, and may be delivered without waiting for any synchronisation with the evaluation of the body form.

This means that the activity constructor form

```
activity( expr )
```

is, in the absence of errors, equivalent to

```
(interface(x() [ expr ] ).x() )
```

An activity constructor form could replace announcements. If operation a is an announcement in

```
x.a( expr )
```

then, in the absence of errors, the same effect would be achieved if a were an interrogation with the same body in

```
[ v* = ( expr ); activity(x.a( v** ) ) ]
```

where v* is a suitable list of names and v** a form that reconstructs the anonymous outcome of expr. It is not sufficient to substitute the body of a into the activity constructor. The interface containing a might have a concurrency guard, and may have been created in a different context.

Since errors can become visible at an invoker, the activity constructor would have to be able to report any such errors for a more complete equivalence.

The relative merits of these approaches to activity construction are for further study.

3.4 Basic types

Most computational models include 'built in types' as primitive elements. The ability to conditionally evaluate some form is typically dependent upon a 'boolean' type which must therefore be part of the model. The ANSA computational model allows such types, and values of those types, to be defined using only the elements of the model that have already been defined.

In the ANSA computational model, the only elements that are presumed to be comparable are operation names and termination names. In other words,

these are the only elements for which the model requires that a test for sameness must exist. This decision reflects the fact that types such as integer and boolean are represented differently on different kinds of machine or even between different languages on the same machine. In practice, local representations are used for common types and translations are applied when transmitting values between objects that use different representations. The definition of types in terms of the behaviour of the values of those types is simply a statement of the requirement that the correct translations between local representations be used.

It is not immediately obvious that the ANSA computational model can be used to define values of common types such as boolean and integer. This section gives examples of how these two types could be implemented; other types can be constructed using similar principles.

3.4.1 Boolean

A boolean type must have two distinct values. The only way to distinguish between values is to use the values to select between alternative forms to evaluate. This is achieved using a termination case selector and interfaces that contain operations with the same name but which deliver outcomes with different names when invoked.

The conditional expression

```
IF condition THEN then-body ELSE else-body FI
```

can be written:

```
after ( condition.if() ) handle
( true()  then-body
  false() else-body
)
```

where

```
true_value  = interface( if()->true()  (->true() ) )
false_value = interface( if()->false() (->false()) )
Boolean = type( if()->true()->false() )
```

3.4.2 Integer

All of the integers can be defined in terms of 0 and 1 using addition and subtraction, all the comparisons can then be constructed in terms of the ability to tell whether an integer is positive, negative or zero.

The basic integers defined here have only three operations. The operations `add` and `subtract` have their conventional meanings. The operation `sign` not only indicates the sign of the integer but also delivers the next integer nearer to zero for non-zero values. This is the fundamental mechanism upon which all of the rest of the definition is constructed.

```
Int =
  type ( sign() ->zero() ->positive(Int) ->negative(Int)
         add(n:Int)->(Int)
         subtract(n:Int)->(Int)
       )
```

Since zero and one refer to each other they must be constructed together. The definitions use simple arithmetical equalities except for:

1. the subtract operation of zero which constructs a negative number when applied to a positive number and
2. the add operation of one which constructs positive numbers.

```
zero one =
( interface                     % zero
  ( sign() ->zero() (->zero()) % identifies this as zero
    add(m:Int)->(Int) (m)       % 0+m = m
    subtract(m:Int)->(Int)      % 0-m
    [ after (m.sign())
      handle                    % three cases
      ( zero() (zero)           % 0-0 = 0
        positive(mMinus1:Int)   % 0-positive: build an interface
        [ minusm =              % which needs to refer to itself
            interface
            ( % new interface is negative: (0-m)+1 = 0-(m-1)
              sign() ->negative(Int)
              [->negative(zero.subtract(mMinus1))]
              add(n:Int)->(Int)
              [ after (n.sign())
                handle
                ( zero() (minusm)       % (0-m)+0 = (0-m)
                  positive(nMinus1:Int) % (0-m)+n = (n-1)-(m-1)
                    (nMinus1.subtract(mMinus1))
                 negative(nPlus1:Int)  % (0-m)+n = 0-(m+(1-(n+1)))
                    [ zero.subtract(m.add(one.subtract(nPlus1)))]
                )
              ]
              subtract(n:Int)->(Int)
              [ after (n.sign())
                handle
                ( zero() (minusm)       % (0-m)-0 = (0-m)
                  positive(nMinus1:Int) % (0-m)-n = 0-(m+n)
                    [zero.subtract(m.add(n))]
                 negative(nPlus1:Int)  % (0-m)-n = 0-((m-1)+(n+1))
                     [ zero.subtract(mMinus1.add(nPlus1))] )]
            )
        ]
        negative(mPlus1:Int)   % 0-negative: 0-m = 1-(m+1)
        [ one.subtract(mPlus1) ]
      )
    ] { subtract }
  ),
  interface                     % one
  ( sign() ->positive(Int)      % positive: 1-1=0
    [ -> positive(zero) ]
    add(m:Int)->(Int)           % 1+m
    [ after (m.sign())
      handle                    % three cases
      ( zero()(one)             % 1+0 = 1
        positive(mMinus1:Int)   % 1+positive: build an interface
        ( mPlus1 =              % which needs to refer to itself
            interface
            ( % new interface is positive: (1+m)-1 = m
              sign() ->positive(Int)
```

```
                    [ -> positive(m) ]
                    add(n:Int)->(Int)
                    [ after (n.sign())
                      handle
                      ( zero()(mPlus1)          % (1+m)+0 = (1+m)
                       positive(nMinus1:Int)  % (1+m)+n = 1+(1+m)+(n-1)
                           ( one.add(mPlus1).add(nMinus1))
                         negative(nPlus1:Int)   % (1+m)+n = m+(n+1)
                           (m.add(nPlus1))
                      )
                    ]
                    subtract(n:Int)->(Int)     % (1+m)-n = (m+1)+(0-n)
                    [ mPlus1.add(zero.subtract(n))]
                  )
              )
              negative(mPlus1:Int)    % 1+negative: 1+m = m+1
              (mPlus1) )
        ]
        subtract (m:Int) ->(Int)    % 1-m = 1+(0-m)
        [ one.add(zero.subtract(m)) ]
    )
)
```

4 Abstract Representation

4.1 Overview

This chapter describes how the forms defined in Chapter 3 can be represented within the ANSA computational model.

An important reason for defining a construction model is the desire to attach **attributes** to various forms. These attributes imply that the form is to be transformed before it is evaluated in some way designated by the attribute. In order to define transformations, it is also necessary to define a representation of the construction model. Such a representation also allows type analysis to be explained using the ANSA computational model. The representation of the model is abstract in that it defines the operations on a form that might be required by a type analyzer or a translator, rather than how a form represents its own state.

4.2 Type definitions

4.2.1 Interface type definer

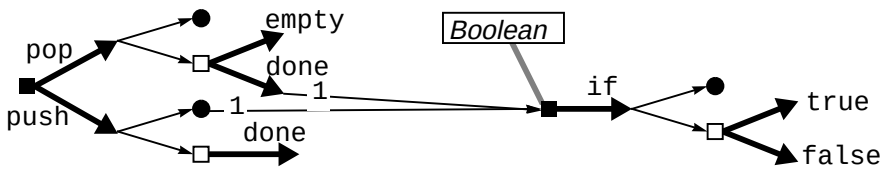
Purpose: To define interface types

Some of the basic forms need to include definitions of interface types. For example, in an interface constructor the types of arguments and results of operations need to be defined. In a textual notation, it must be possible to associate names with interface types since it is often necessary to refer to the same type many times. Consider the interface constructor to create a stack of boolean values. This might be written as:

```
interface ( push(b:Boolean)->done() body-of-push
            pop()->done(Boolean) ->empty() body-of-pop)
```

In the graphical notation, the ability to point to the same entity more than once renders this use of names unnecessary as shown in Figure 4.1

Figure 4.1: Stack of Boolean type



Components

signatures	a set of signatures

Notation

```
type                    =  “type” [attributes] “(“ {signature} “)”
```

where signature is as defined in §3.3.2.

Evaluation

Evaluating an interface type definer creates a new interface which represents the type to be defined. The evaluation delivers an anonymous outcome which refers to a single interface, the newly created interface.

An interface type definer can therefore be used as the value form of a constant binder. This allows a name to be bound to the interface that represents a type. For example, the boolean type above might be introduced and named as follows:

```
Boolean = type ( if() ->true() ->false() )
```

Type

The type of an interface type definer has only an anonymous termination with a single interface type which is the type of interfaces used to represent types.

Interface types are represented in the model by interfaces and can therefore be passed as parameters within the model as already defined. This means that type analysis and checking can be provided through operations in interfaces.

Having decided to use interfaces to represent types, the existing forms that manipulate interfaces can also be used for types. A form that, when evaluated, delivers an interface that represents a type can be used wherever an interface type definer is required in the basic forms. There are some restrictions upon the forms that can be used in this way, these relate to the ability perform static type analysis.

Using interfaces to represent types also allows existing forms to be used to create generic definitions which are then instantiated with a particular type. A generic stack and its instantiation as a stack of booleans can be defined as:

```
stack = interface
        ( of(X:AbstractType)
            ->(type ( push(v:X)->done()
                      pop()->done(X) ->empty() ) )
          [ state-variable-initialization;
            interface ( push(v:X)->done() body-of-push
                        pop()->done(X) ->empty() body-of-pop
                      )
          ]
        );
bs = stack.of(Boolean)
```

Since interfaces are used to represent types there is an interface type to which the types of all such interfaces conform. There is an interface that represents that type and the examples assume a constant binding of the name `AbstractType` to that interface.

If the interface delivered in the anonymous termination of `stack.of(Boolean)` is to be passed as a parameter or bound in a variable binding, then an interface that represents the type of that interface must be constructed. Since the type to be represented is dependent upon the

parameter, it would be convenient for the operation to deliver an interface that represents the type as well as the interface to be used as a stack.

This can be achieved by returning two results in the outcome:

```
stack = interface
        ( of(X:AbstractType)
            ->(type ( push(v:X)->done()
                      pop()->done(X) ->empty() ) )
          [ state-variable-initialization;
            ( interface ( push(v:X)->done() body-of-push
                          pop()->done(X) ->empty() body-of-pop
                        ),
              type ( push(v:X)->done()
                     pop()->done(X) ->empty() )
            )
          ]
        );
bs TypeOfbs = stack.of(Boolean);
v:TypeOfbs := bs;
```

The use of the two results is illustrated by the variable binder in the final line of the example. The result that represents the type is used as the type definer in the declaration, the result that is to be used as a stack is used as the value form.

4.2.2 Definition of the type 'AbstractType'

Type definers are included in the model in order to allow conformance checks to be carried out. The type indicated by the name AbstractType above must therefore have operations appropriate for this purpose. Interfaces of this type must be able to represent the types which have been represented graphically in Chapter 2. For the purpose of this discussion, AbstractType will be assumed to be defined as:

```
AbstractType = type ( getSpecification()->(Specification) )
```

where interfaces of the type Specification have the operations necessary to allow type analysis and conformance checking.[1]

The definitions that follow will need to be extended in order to allow construction of types while carrying out inferencing. The definitions given here were developed in order to explore the operations needed for conformance testing.

The definitions also need to be extended to accommodate analysis in the presence of type parameters.

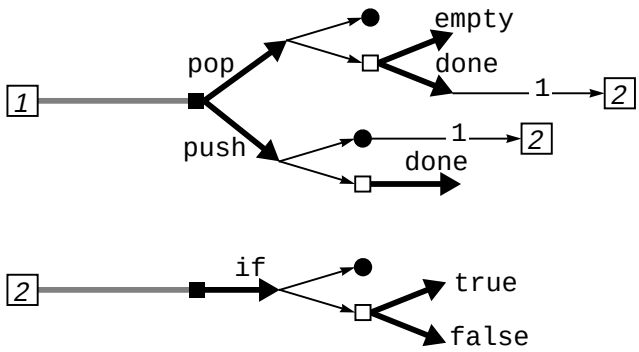
```
Specification = type( index()->(Integer)
                      context()->(TypeContext)
                     conformsTo(y:Specification)->()->no(Reason)
                    );
```

1. This indirection is now considered unnecessary, perhape even harmful. It was originally introduced to allow both an interface and the type of that interface to be returned but the example above shows that multiple results can do this. The indirection has been preserved for the time being because the prototype type checker uses it.

The primary goal of these definitions is to be able to support the conformsTo operation. The anonymous termination of the conformsTo operation indicates success, the termination named no indicates failure. It is not strictly necessary for there to be any explanation of why one type does not conform to another but in practice it is helpful to provide a reason.

In order to be able to represent type graphs with cycles and, in particular, to be able to detect cycles in independently constructed graphs, each type is represented as an index (a name) and a context which maps indices to interfaces that represent fragments of the graph. In the definitions that follow, integers are used as indices but any constant type with an equality operation would do just as well. Figure 4.2 illustrates how this representation would be applied to the stack of boolean example above.

Figure 4.2: Context and index representation



An index is associated with each interface type node and the arcs which previously led to interface type nodes now lead to the index for that node in the context in which the graph is to be interpreted. This representation depends upon allocating indices so that they are unambiguous names for graph fragments in any one context but does not depend upon the indices being unique names - i.e. many indices may refer to identical graph fragments.

The representation as interfaces corresponds closely to the graphical representation. The types of interfaces that follow correspond to types of node in the graph.

The type of a type context is:

```
TypeContext = type( find(i:Integer)->(TINode)->notFound() )
```

The only operation required on a type context is to be able to retrieve the interface type node corresponding to a given index.

The type of an interface type node is:

```
TINode = type( find(n:String)->(TSNode)->notFound()
               isConformedToBy(iw1:TINode
                               cw cx:TypeContext
                               wxp xwp:Pairs)->()->no(Reason)
               conformsTo(i:TINode
                          cx cy:TypeContext
                          xyp yxp:Pairs)->()->no(Reason) )
```

For an interface type node, it must be possible to retrieve a signature node given an operation name, represented here as a string. It must also be possible to test for conformance between this node and some other interface type node. In general, the nodes may be defined in different contexts so both contexts must be accessible. It is also necessary to know which interface type node pairs have already been visited; this is done using two sets of index pairs.

The type of a signature node is:

```
TSNode = type( args()->(TLNode)
               response()->(TRNode)->none()
               conformsTo(s:TSNode
                          cx cy:TypeContext
                          xyp yxp:Pairs)->()->no(Reason) )
```

A signature node must have an operation that returns the list of indices that represent the types of the arguments. It must also have an operation that distinguishes interrogations from announcements and, for an interrogation, returns a description of the response type. It will also be necessary to be able to test for conformance between signatures so an operation has been provided for this purpose.

The type of a list of indices node is:

```
TLNode = type( look()->(Integer TLNode) ->end()
               conformsTo(l:TLNode
                          cx cy:TypeContext
                          xyp yxp:Pairs)->()->no(Reason) )
```

The list of indices required by the definition of a signature node requires an operation that delivers either the head and tail of the list or an end indication. Once again, there is a conformance testing operation.

A signature node may also refer to a response type node, for which the type is:

```
TRNode = type( find(n:String)->(TLNode)->notFound()
               conformsTo(r:TRNode
               cx cy:TypeContext
               xyp yxp:Pairs)->()->no(Reason) )
```

A response type node requires an operation that, given a termination name, delivers the list of indices for the corresponding result types. As for the other node types, there is a conformance testing operation.

The types presented above have referred to sets of index pairs. The type for such a set is:

```
Pairs = type( look()->(Integer Integer Pairs) ->end()
              find(x y:Integer)->()->notFound() );
```

A set of index pairs may be represented as a list of index pairs.

Chapter 5 includes definitions of interface constructors that create interfaces of the types defined above. The bodies of the operations given there combine to form an interface type conformance testing algorithm.

4.3 Transformation

An important goal of the representation of the construction model is to allow a form to be transformed. The idea is that the resulting form is like the original form but with some additional characteristic associated with the particular transformation that has been applied.

If the transformation is to be expressed in terms of the model then it must be able to manipulate the representation of a form and be able to construct the representation of some replacement form. This means that the representations of all the forms must be defined so that they can be generated and manipulated by the forms of the model. As for the type definitions, this means using interfaces.

This chapter outlines the principles of the attribute scheme by which transformations are named. A full description will be made available when significant examples of the use of transformations have been produced.

4.3.1 Attribute processing

The first requirement is to identify the transformation and the form to which it must be applied. The notation introduced in Chapter 3 can be extended to include attibutes which satisfy this requirement.

```
object_constructor    =  “object” [ attributes ] composer

interface_constructor =  “interface” [ attributes ]
                         “(” [ “path” “=” path-expr ]
                             {signature body}
                          “)”

type                  =  “type” [attributes] “(” {signature} “)”

signature             =  operationName [ attributes ] arguments
                          { termination }

attributes            =  “<” { attributeName [ attributeBlock ] }
                         “>”
```

The attribute mechanism operates by applying an operation to the definition of the form to which the attribute applies. This could be likened to the ability to change the text of the indicated part of program but operates upon the abstract syntax tree. This allows the type system to be used to prevent the transformation generating syntactically incorrect forms. When attributes are included in a form then a transformation is applied to the form for each attribute. The particular transformation is determined by the name used for the attribute and the type of form being transformed.

Suppose a specification contained the form:

```
object < example > composer
```

Then the name example must resolve to an interface known to be of a type that conforms to:

```
type( transformObject(ObjectForm, Context)->(Form) )
```

The program would be considered to include the result of

```
example.transformObject(thisObject, transformer_context)
```

where `thisObject` is the abstract representation of the specification given above. The parameter `transformer_context` povides access to information such as the names and types of bindings that are in scope at the place where `thisObject` appears.

This mechanism is revealing what would normally be considered a compiler's internal representation of a program and allowing that representation to be manipulated. The type inferencing and checking can be applied to the transformed program so this is not a way to bypass the type checker.

Experiments with transformations for atomicity [APM1004] indicate that, in general, the transformer will require access to scope and type information. Either the transformer must provide these for itself, or it must be able to interact with mechanisms that perform such an analysis. The experiments suggest that transformation and analysis have mutual dependencies and they will have to be done concurrently[1].

There is an interface type corresponding to each of the contexts in which attributes may appear. These are:

```
type ( transformObject(ObjectForm)->(Form) )
type ( transformType(TypeForm)->(Form) )
type ( transformInterface(InterfaceForm)->(Form) )
type ( transformSignature(Signature)->(Signature) )
```

Note that the outcome of the transformation is not necessarily the same kind of form as the input. This is another issue uncovered by the work on atomicity transformations.

The attribute may resolve to an interface of a type which conforms to more than one of these types. This allows the same name to be used as a attribute in more than one context.

In order to write any of these transform operations, the interface types of interfaces used to represent basic forms must be defined. In general, a transformation may need to replace arbitrarily deeply buried parts of the form being transformed and so it needs to be able to distinguish the different kinds of form, extract subforms and create new forms for all of the kinds of form that might be encountered.

4.3.2 Unresolved issues

The attribute names must be resolved in some naming context which must exist when the transformations are to be applied. The transformation interfaces must have been created in some context which must still be accessible when the transformation operations are evaluated. There is an unresolved issue of the relationships between these contexts and the naming context which will exist when the transformed form is evaluated.

The transformation mechanism is similar to a macro mechanism and many of the same issues arise. In the description above it was assumed that the attributes were part of the representation of the object constructor form supplied to the transformer. If attributes are included in the input form then both the argument and result could include attributes. If the result includes a

1. Concurrently in same sense as for composers. Both must have started before either can finish, and there will be communication between them.

attribute, when and how should the transformation for that attribute be applied?

Having expressions in attributes can be handled in one of two ways. If the argument to the transformation includes the attribute that provoked the transformation, then the expression will be part of that attribute and therefore accessible to the transformation. If the attribute is removed from the argument then the expression must be provided through the arguments of the operation. This could be done by having a second argument (of some type to be defined) through which the representation of the expression was passed. Passing the attribute as part of the input form seems to be the simpler solution.

Since more than one attribute may be specified, in which order should the transformations be applied? In some cases the effect may be the same whichever order is chosen, but in general this is not the case.

Although the atomicity transformation work has provided some guidance, it will not be possible to resolve these issues until other transformations have been attempted.

5 Type Conformance Revisited

5.1 Overview

This chapter uses examples, presented in the graphical notation introduced in Chapter 2, to illustrate some of the implications of the type conformance relation and also to show how the rules capture conformance as defined in terms of the absence of interaction errors.

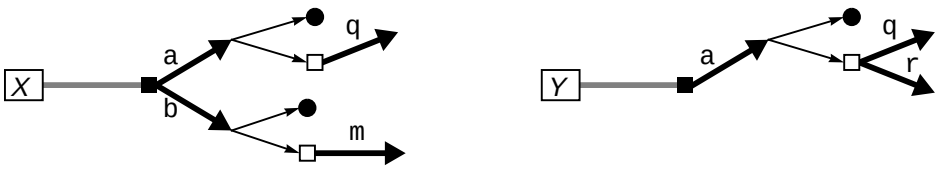
It also gives definitions of interface type constructors that create interfaces of the types defined in Chapter 4 with operations that combine to form a type conformance checker.

5.2 Type conformance examples

5.2.1 Types without parameters

Figure 5.1 defines two interface types *X* and *Y* which will be used to illustrate how conformance is defined for types without parameters.

Figure 5.1: Conformance without parameters



There is an arc named b from *X* but not from *Y*, this corresponds to an operation named b which is in interfaces of type *X* but not in interfaces of type *Y*. A server providing an interface of type *Y* is not expecting requests with the operation name b. If a client attempts to use that interface as if it were of type *X* then it might invoke the operation named b thus causing an interaction error. The absence of an arc named b from *Y* means that *Y* does not conform to *X*.

1. if there is some operation name op for which *T*op exists but *S*op does not exist then *S* does not conform to *T*.

Since *X* has an arc named a, conformance of *X* to *Y* depends upon the graphs beyond those arcs. The nodes reached from *X* and *Y* by arcs named a will be called *X*a and *Y*a respectively.

There must be an arc to an argument node from both *X*a and *Y*a, the nodes reached by these arcs will be called *X*a● and *Y*a● respectively. The number of arcs from *X*a● and *Y*a● must be the same since this defines the number of arguments that a client will send and a server will expect.

2. for each node Top•, if there is an n for which one but not both of Sop•n and Top•n exist then *S* does not conform to *T*.

In this example there are no arcs from either *X*a• or *Y*a• and so the end of a path that can be followed from both *X* and *Y* has been reached without a potential interaction error being detected.

The existence of an arc to a response type node from *X*a determines whether the operation is an interrogation or an announcement. The response type nodes reached from *X*a and *Y*a will be called *X*a❒ and *Y*a❒ respectively. These arcs must either both be present, as in the example, or both be absent. Using an interrogation as if it were an announcement causes an unexpected response interaction error, using an announcement as if it were an interrogation causes a no response interaction error.

3. for each node *T*op, if *S*op❒ exists but *T*op❒ does not exist then *S* does not conform to *T*.

The test for conformance of *X* to *Y* is completed by considering the graphs beyond *X*a❒ and *Y*a❒. It is convenient to define a conformance relation over response types since this will be the basis for type checking in the construction model. For *X* to conform to *Y*, *X*a❒ must conform to *Y*a❒.

4. for each node Top❒, if Sop❒ does not exist or if Sop❒ does not conform to Top❒ then *S* does not conform to *T*.

In order to establish the nature of the response type conformance relation, the conformance of *Y*a❒ to *X*a❒ will be considered first.

The response type *Y*a❒ has terminations named q and r. The response type *X*a❒ has only a termination named q. A server issuing a response of type *Y*a❒ might choose the termination name r and if the client were expecting a response of type *X*a❒ then this would cause an interaction error. Therefore, *Y*a❒ does not conform to *X*a❒.

5. if there is some termination name term for which *S*term exists but *T*term does not exist then *S* does not conform to *T*.

Since *Y*a❒ has an arc named q, conformance of Xa❒ to *Y*a❒ depends upon the graphs beyond *X*a❒q and *Y*a❒q. The number of arcs from these nodes must be the same since this defines the number of parameters that the server will send and the client will expect. In this example there are no arcs from either *X*a❒q or *Y*a❒q and so the end of a path which can be followed from both *X*a❒ and *Y*a❒ has been reached without a potential interaction error being detected.

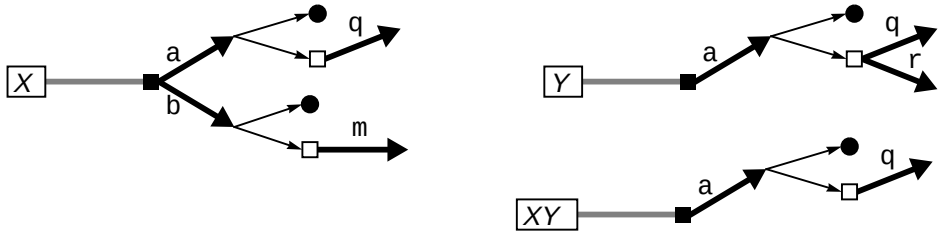
6. for each node *T*term, if there is an n for which one but not both of *S*term.n and *T*term.n exist then *S* does not conform to *T*.

There are no more paths to explore for this example and no potential interaction errors have been discovered therefore *X*a❒ conforms to *Y*a❒ and *X* conforms to *Y*.

Figure 5.2 shows the graphs for *X* and *Y* together with the graph, labelled *XY* which contains only those paths which must exist for *X* to conform to *Y*.

This type has the property that *X* conforms to *XY* and *XY* conforms to *Y* but in general there is more than one type with this property for a pair of types where one conforms to the other.

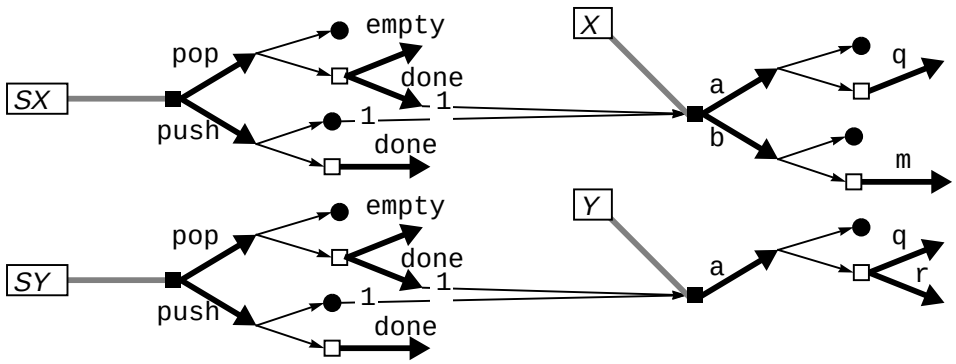
Figure 5.2: Conformance without parameters - common subgraph



5.2.2 Types with parameters

The example in Figure 5.1 had neither argument nor result parameters. Figure 5.3 defines types *SX* and *SY* which combine the stack example with the types *X* and *Y*. This example will be used to show how parameters affect conformance.

Figure 5.3: Conformance with parameters



There are the same number of arcs from *SX*push• as from *SY*push• but in this case both have one arc. *SX*push•1 is X and *SY*push•1 is *Y*, both as in the previous example, so all that needs to be done is to determine what conformance is required for arguments.

A server which provides an interface of type *SX* is expecting to be able to use the argument to the operation named push as if of type *X*. If the argument is known to be of type *Z* then it would be necessary for *Z* to conform to *X* in order to be sure that there will be no interaction errors caused by use of the argument. A client using an interface of type *SY* may supply an argument of type *Y*. Since it has been shown that *Y* does not conform to *X*, using an interface of type *SX* as if of type *SY* may lead to an interaction error and therefore *SX* does not conform to *SY*.

7. if *S*op•n and *T*op•n exist for some op and n but *T*op•n does not conform to *S*op•n then *S* does not conform to *T*.

The other case to consider is the conformance of *SY*pop❒ to *SX*pop❒. There are the same number of arcs from *SX*pop❒done as from *SY*pop❒done but in this case both have one arc. *SX*pop❒done.1 is *X* and *SY*pop❒done.1 is *Y*, both as in the previous example, so all that needs to be done is to determine what conformance is required for result parameters.

A client expecting a response of type *SX*pop❒ is expecting the interface referenced in a response with termination name done to be usable as if of type

X. If the interface is known to be of type *Z*, then it would be necessary for Z to conform to *X* in order to be sure that there will be no interaction errors caused by use of the interface. A server which is supplying a response of type *SY*pop❐ may refer to an interface of type *Y* in a response with termination name done. Since it has been shown that *Y* does not conform to *X*, *SY*pop❐ does not conform to *SX*pop❐ and therefore *SY* does not conform to *SX*.

8. if `S`term.n and *T*`term.n` exist for some `term` and `n` but *S*`term.n` does not conform to *T*`term.n` then *S* does not conform to *T*.

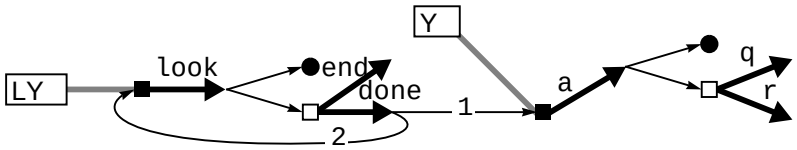
It has been shown above that neither of *SX* and *SY* conform to the other despite their apparent close relationship. The graphs are identical up to the node where one has *X* and the other *Y* and *X* conforms to *Y*. The failure to conform in either direction will arise in every case where a type node can be reached in each graph both by a path with an even number of argument nodes, and a path with an odd number of argument nodes, and the type nodes do not define types that are equivalent (i.e. each conforms to the other).

5.2.3 Recursive types

The graphical notation gives a simple way to define recursive types. The conformance relations have been described, in this chapter, in terms of showing that types do not conform because this description can apply to recursive types in a way that would not be possible for a definition based upon showing that types do conform.

Figure 5.4 defines the type *LY* which is appropriate for a list of interfaces of type *Y*.

Figure 5.4: Recursive type - list of Y



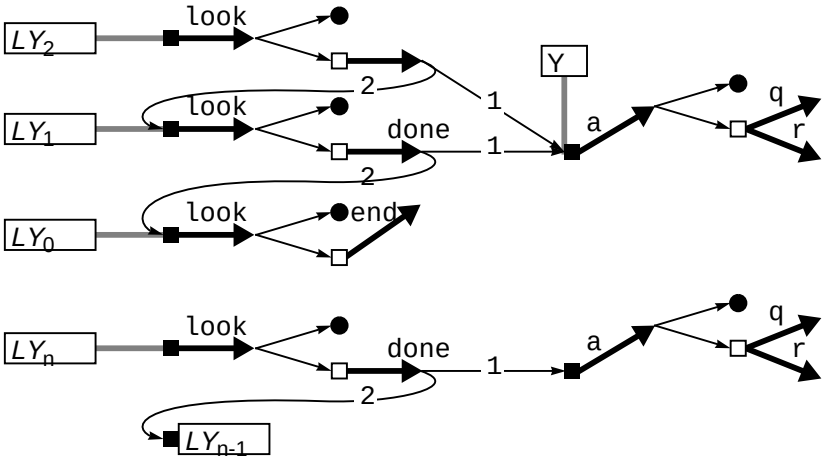
An interface of type *LY* has an operation named `look` which takes no arguments and which is an interrogation. The response type has two terminations; one named `end` with no interface types, the other named `done` with interface types *Y* and *LY*. The idea is that the response to a `look` request will either deliver the head and tail of the list represented by the interface or will signal the end of the list.

Figure 5.5 defines three interface types which conform to *LY*. LY_0 represents an empty list, LY_1 represents a list with one element and LY_2 represents a list with two elements. The extension of this graph to include lists up to any chosen length is simply a case of adding as many copies as necessary of the graph which defines LY_n in terms of LY_{n-1}.

LY_0 involves neither argument nor result parameters and so can be shown to conform to *LY* in the way described for *X* and *Y* in Figure 5.1. Showing that LY_1 and LY_2 both conform to *LY* could be done directly but showing that if LY_{n-1} conforms to *LY* then so does LY_n covers not only those cases but also all of the extensions of the graph suggested above.

Although all of the types LY_i conform to *LY*, and there is no more specific type to which all of the types LY_i conform, there can be interfaces of type *LY* that

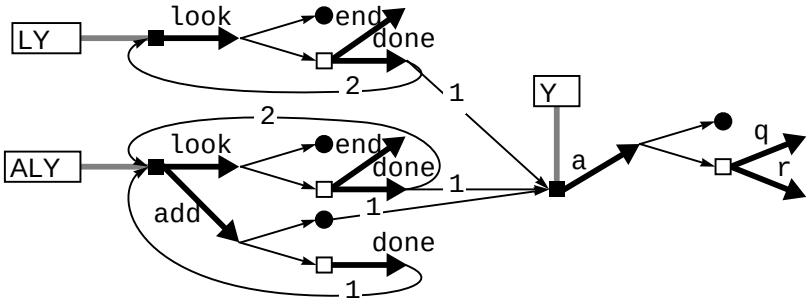
Figure 5.5: List of Y - specific lengths



are not of any type LY_i. In particular, a look operation that sometimes delivers an end response and sometimes delivers a done response has a response type with two terminations. An interface that has such an operation can be of type *LY* but not of any type LY_i.

The type *LY* did not offer any operations that might be used to construct lists. Figure 5.6 defines a type *ALY* which will be used to illustrate conformance between recursively defined types. An interface of type *ALY* has an operation named add which takes an interface of type *Y* as argument and which delivers a response named done containing an interface of type *ALY*.

Figure 5.6: Recursive types - conformance



LY does not conform to *ALY* since an interface of type *LY* does not have an operation named add.

ALY conforms to LY. The demonstration is straightforward up to the point where it is necessary find out whether or not *ALY*look❒done.2 (i.e. *ALY*) conforms to *LY*look❒done.2 (i.e. *LY*). At this point it has been established that *ALY* conforms to *LY* unless *ALY* does not conform to *LY*. Returning to the definition of conformance makes it clear how to handle this case. If there are no paths that lead to potential interaction errors then conformance has been established. The path look❒done.1 leads to a pair of nodes that have already been visited and so cannot lead to an interaction error unless some other path leads to an interaction error.

It is not sufficient that the nodes have been visited at some point in each of the paths, they must have been visited at the same point in both paths and also with the same conformance requirement. Arguments introduce a reversal of

the conformance requirement and this must be taken into account. Figure 5.7 introduces a number of types that will be used to illustrate these issues.

Figure 5.7: Type recursion through arguments

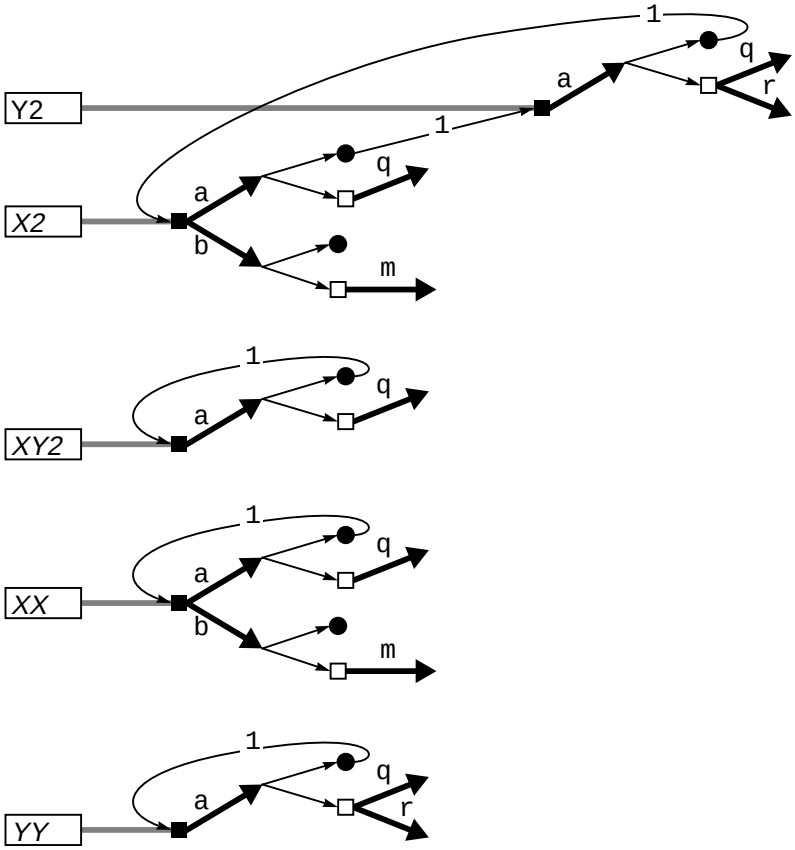


Table 5.1 shows which of these types conform to which others. Each cell contains Y if the type for the row conforms to the type for the column and N if it does not.

Table 5.1: Conformance between types

	X2	XY2	XX	YY	Y2
X2	Y	Y	Y	Y	Y
XY2	N	Y	N	N	Y
XX	N	N	Y	N	Y
YY	N	N	N	Y	Y
Y2	N	N	N	N	Y

Conformance for these cases can be determined by combining the treatment of cycles in the graph, which has just been explained for result parameters, with the treatment of argument parameters which was explained earlier. Three cases will be explained here in order to illustrate this.

X2 conforms to *Y2*; in other words, any interface of type *X2* can be used as if it were of type *Y2*. Up to the point where the argument to the operation named a is considered, this is exactly like *X* and *Y* of Figure 5.1. Using the interface as if of type *Y2* requires that the interface offered as the argument to the

operation named a be usable as if of type *X2*. An interface of type *X2* will use this argument as if of type *Y2*. This passing of an argument cannot lead to an interaction error unless there was some other way to reach that interaction error.

XY2 conforms to *Y2*. This is similar to the previous case but here the interface whose operation is to be invoked is of type *XY2* and will use the argument as if of type *XY2*. The interface offered is known to be usable as of type *X2* and so it is necessary to show that *X2* conforms to *XY2*. Once again, the new part of the problem is in dealing with the argument. The possible nested invocation will be supplied with an argument known to be usable as of type *XY2* and will actually be used as if of type *Y2*. There can be no potential interaction errors reachable only by this path and so it has been shown not only that *XY2* conforms to *Y2* but also that *X2* conforms to *XY2*.

XX does not conform to *YY*. There is no path *YY*a•1.b nor is there a path XXa•1.a❒r. Both of these represents interaction errors that can arise if, using an interface as if of type *YY*, the operation named a is invoked with an argument of type *YY* but not *XX*. The server is expecting an argument of type *XX* in the request for a and may attempt to invoke the operation b which is not present if the type of the argument is *YY* - the absence of the path *YY*a•1.b indicates this potential error. If the server invokes the operation a it will not expect the response named r which could be made by the a operation of an interface of type *YY* - the absence of the path *XX*a•1.a❒r indicates this potential error.

5.3 A conformance testing algorithm

Chapter 4 presented a collection of types for interfaces that can be used to represent types at least for the purposes of conformance testing. This section presents interface constructors that create interfaces of those types. The operations defined below combine to form a type conformance tester.

The examples given below include mechanisms to construct strings that explain why the types are not related by conformance and the type Reason used in the definitions in Chapter 4 is used as a synonym for String. Indeed, the program from which the pieces of code, both here and in Chapter 4 were extracted, includes the form:

```
Reason = String;
```

5.3.1 Constructor for Specification

```
specNode = interface
    ( new(xi:Integer xtc:TypeContext):(Specification)
      [ interface
        ( index():(Integer) (xi)
          context():(TypeContext) (xtc)
          conformsTo(y:Specification):()->no(Reason)
            [ ytc = y.context();
              yt xt = after
                      ( ytc.find(y.index())
                      | xtc.find(xi)
                      ) handle
                      ( notFound() [ ->no("Bad type node") ]);
              xt.conformsTo(yt, xtc, ytc, noPairs, noPairs)
            ]
        )
      ]
    );
```

This is a definition of a specification node factory. Invoking the operation named new constructs and returns a specification interface. A possible use would be:

```
x   = specNode.new(1,xycontext);
y   = specNode.new(2,xycontext);
x.conformsTo(y);
```

The index and context operations of a specification node interface are straightforward. The conformsTo operation takes another interface of type Specification as its argument. It finds the two interface type nodes by invoking the find operation of each context with the corresponding index - note that no constraint need be applied to the order in which these finds are performed but that both results are required. The real work is done by invoking the conformsTo operation of the interface type node.

5.3.2 Constructor for index pair lists

```
Pair =
    interface
    ( end():(Pairs)
        [ interface
          ( look()->end() [->end()]
            find(x y:Integer)->notFound() [->notFound()]
          )
        ]
      add(x y:Integer p:Pairs):(Pairs)
        [ interface
          ( look():(Integer Integer Pairs) (x,y,p)
            find(x1 y1:Integer) :() ->notFound()
              [ after
                  ( after ( x.equal(x1) ) handle
```

```
                    ( true() [ y.equal(y1) ] )
                  )
                handle
                ( true() []
                  false() [ p.find(x1,y1) ]
                )
              ]
          )
        ]
    );
 noPairs = Pair.end();
```

This is the definition of the index pair mechanism and the empty set of pairs which was used twice above.

5.3.3 Constructor for type context nodes

```
TCNode =
  [ endNode = interface( find(i:Integer)->notFound()
                           [->notFound()] );
    interface
    ( zero():(TypeContext) [ endNode ]
      more(n1:Integer i:TINode tc:TypeContext):(TypeContext)
        [ interface
          ( find(n2:Integer):(TINode)->notFound()
              [ after ( n1.equal(n2) ) handle
                ( true()  [i]
                  false() [ tc.find(n2) ]
                )
              ]
          )
        ]
    )
  ];
```

The type context factory is very simple and can be used as follows:

```
xycontext = TCNode.more(2,yINode,
            TCNode.more(1,xINode, TCNode.zero()))
```

This definition uses a simple list of {integer, interface type node} pairs to support the ability to look up an interface type node given an integer index.

5.3.4 Constructor for interface type nodes

```
INode =
  [ endNode = interface
      ( find(n:String)->notFound() [->notFound()]
        isConformedToBy(iw1:TINode
                        cx cw:TypeContext
                        xwp wxp:Pairs):()->no(Reason) [ ]
        conformsTo(iy1:TINode
                   cx cy:TypeContext
                   xyp yxp:Pairs):()->no(Reason)
          [ iy1.isConformedToBy(endNode, cy, cx, yxp, xyp)]
      );
    interface
    ( zero():(TINode) [ endNode ]
```

```
        more(nx1:String sx1:TSNode ix2:TINode):(TINode)
          [ asString = nx1.append("(&)");
            thisNode =
              interface
              ( find(n:String):(TSNode)->notFound()
                  [ after ( n.equal(nx1) ) handle
                    ( true()  [sx1]
                      false() [ ix2.find(n) ]
                    )
                  ]
                isConformedToBy(iw1:TINode cx cw:TypeContext
                                xwp wxp:Pairs):()->no(Reason)
                  [ after
                      [ iw1.find(nx1)
                           .conformsTo(sx1, cw, cx, wxp, xwp) ]
                    handle
                    ( notFound() [->no(asString)]
                      no(r:Reason)
                        [->no(r.append(" in ").append(asString))]
                    );
                    ix2.isConformedToBy(iw1, cx, cw, xwp, wxp)
                  ]
                conformsTo(iy1:TINode
                           cx cy:TypeContext
                           xyp yxp:Pairs):()->no(Reason)
                  [ iy1.isConformedToBy(thisNode,
                                        cy, cx, yxp, xyp)]
              )
          ]
      )
    ];
```

This defines an interface type node factory. The use is straightforward:

```
xINode = INode.more("b",xbSNode,
         INode.more("a",xaSNode, INode.zero()));
yINode = INode.more("a",yaSNode, INode.zero()));
```

The representation as a list of operation name, signature node pairs is straightforward as is the finding of a signature given an operation name. The `conformsTo` operation simply invokes the `isConformedToBy` operation of its argument supplying itself as the argument and with the appropriate exchange of positions of the contexts and sets of index pairs.

The end of list node represents the type of an interface with no operations. Since every interface is usable as if it had no operations, the `isConformedToBy` operation of the end node always reports success.

Other interface type nodes contain an operation name and a signature. The `isConformedToBy` operation tests for presence of the operation name in the argument interface type node and conformance of the signatures. If that test succeeds then the rest of the operations are tested by invoking the `isConformedToBy` operation of the tail of the list.

5.3.5 Constructor for signature nodes

```
SNode =
  interface
  ( a(args:TLNode):(TSNode)
      [ interface
        ( args():(TLNode) [args]
          response()->none() [->none()]
          conformsTo(sy:TSNode
                     cx cy:TypeContext
                     xyp yxp:Pairs):()->no(Reason)
            [ after
                [ sy.response();->no("unexpected response")]
              handle
              ( none() [] );
              sy.args().conformsTo(args, cy, cx, yxp, xyp)
            ]
        )
      ]
    i(args:TLNode response:TRNode):(TSNode)
      [ interface
        ( args():(TLNode) [args]
          response():(TRNode) [response]
          conformsTo(sy:TSNode
                     cx cy:TypeContext
                     xyp yxp:Pairs):()->no(Reason)
            [ after
                [ response.conformsTo(sy.response(),
                                      cx, cy, xyp, yxp) ]
              handle
              ( none() [->no("no response")] );
              sy.args().conformsTo(args, cy, cx, yxp, xyp)
            ]
        )
      ]
  );
```

The signature node factory can construct either an announcement node or an interrogation node. The definitions of the operations present no difficulties but note the reversal of the conformance requirement for arguments.

5.3.6 Constructor for response type nodes

```
RNode =
  [ endNode = interface
      ( find(n:String)->notFound() [->notFound()]
        conformsTo(r:TRNode cx cy:TypeContext
                   xyp yxp:Pairs):()->no(Reason)
          [ ]
      );
    interface
    ( zero():(TRNode) [ endNode ]
      more(nx1:String lx1:TLNode rx2:TRNode):(TRNode)
        [ thisNode =
            interface
            ( find(n:String):(TLNode)->notFound()
                [ after ( n.equal(nx1) ) handle
                  ( true()  [lx1]
```

```
                    false() [ rx2.find(n) ]
                  )
                ]
              conformsTo(ry1:TRNode
                         cx cy:TypeContext
                         xyp yxp:Pairs):()->no(Reason)
                [ after
                    [ lx1.conformsTo(ry1.find(nx1),
                                     cx, cy, xyp, yxp) ]
                  handle
                  ( notFound() [->no(thisNode.asString(""))]
                    no(r:Reason)
                     [->no(thisNode.asString(r.append(" in ")))]
                  );
                  rx2.conformsTo(ry1, cx, cy, xyp, yxp)
                ]
              asString(prefix:String):(String)
               [ prefix.append("->").append(nx1).append("(&)") ]
            )
        ]
    )
  ];
```

Response type nodes are lists of (termination name, list of indices) pairs. The definitions are similar to those used for interface type nodes but in this case there is no need to introduce a reversed conformance testing operation.

5.3.7 Constructor for list of indices nodes

```
LNode  =
  [ endNode = interface
      ( look()->end()[->end()]
        conformsTo(l:TLNode cx cy:TypeContext
                   xyp yxp:Pairs):()->no(Reason)
          [ after[ l.look();
                   ->no("different number of parameters")]
            handle
            ( end() [] )
          ]
      );
    interface
    ( zero():(TLNode) [ endNode ]
      more(ix1:Integer lx2:TLNode):(TLNode)
        [ interface
          ( look():(Integer TLNode)(ix1,lx2)
            conformsTo(ly1:TLNode
                       cx cy:TypeContext
                       xyp yxp:Pairs):()->no(Reason)
              [ after
                [ iy1 ly2 = ly1.look();
                  after ( xyp.find(ix1, iy1) ) handle
                  ( notFound()
                      [ cx.find(ix1)
                        .conformsTo(cy.find(iy1),
                                    cx,cy,
                                    Pair.add(ix1,iy1,xyp),yxp)
                      ]
```

```
                    );
                    lx2.conformsTo(ly2, cx, cy, xyp, yxp)
                  ] handle
                  ( end() [->no("different number of parameters")]
                    notFound() [->no("type name not in context")]
                  )
                ]
              )
            ]
          )
        ];
```

The list of indices nodes are where the lists of index pairs are both created and used. It is also where the type contexts are used. Once the pair of head indices from the two lists have been found, these are looked up in the appropriate list of pairs. If the pair is found then a previously visited node has been reached. If the pair is not found then the interface type nodes are retrieved from the contexts and the conformance testing operation is invoked but with an additional pair of indices added to the list of pairs.

5.3.8 Using the constructors

The types X2 and Y2 of Figure 5.7 can be defined as

```
X2 Y2 = ( type( a(v:Y2)->q()
                b()->m() ),
          type( a(w:X2)->q()->r()) )
```

The code to construct interfaces to represent these types would be:

```
l0  =  LNode.zero();
l1  =  LNode.more(1, l0);
l2  =  LNode.more(2, l0);
r0  =  RNode.zero();
i0  =  INode.zero();
ix2=   INode.more("b",SNode.i(l0,RNode.more("m",l0,r0)),
       INode.more("a",SNode.i(l2,RNode.more("q",l0,r0)), i0));
iy2 =       INode.more("a",SNode.i(l1,RNode.more("q",l0,
                                      RNode.more("r",l0))), i0);
context = TCNode.more(2,iy2,
          TCNode.more(1,ix2, TCNode.zero()));
x2  = specNode.new(1,context);
y2  = specNode.new(2,context);
```

An invocation of `x2.conformsTo(y2)` would deliver an anonymous result thus indicating that *X2* conforms to *Y2*. An invocation of `y2.conformsTo(x2)` would deliver a result with termination name no and reason "->r(&) in a(&)" because the operation named a in an interface of type *Y2* may deliver a response with termination name r which is not acceptable for an interface of type *X2*.

References

[APM1002]

A Model for Interface Groups, *APM1002.01*, APM, Cambridge (June 1993).

[APM1003]

The ANSA Naming Model, *APM1003.01*, APM, Cambridge (June 1993)

[APM1004]

ANSA Atomic Activity Model and Infrastructure, *APM1004.00* APM, Cambridge, (June 1993).

[APM1009]

Distributing Objects, *APM1009.01*, APM, Cambridge (June1993).

[APM1010]

Using path expressions as concurrency guards, *APM1010.01*, APM, Cambridge (June 1993).

[APM1014]

DPL Programmers' Manual, *APM1014.01*, APM, Cambridge (June 1993).

[BLACK 86]

Black A, Hutchinson N, Jul E, Levy H; **Object Structure in the Emerald System**, *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 86). ACM Sigplan Notices 21,11:78-86* (November 1986)

[BLACK 87]

Black A, Hutchinson N, Jul E, Levy H, Carter L; **Distribution and Abstract Types in Emerald**, *IEEE Transactions on Software Engineering SE-13,1:65-76* (January 1987)

[LISKOV 88]

Liskov B; **Distributed Programming in Argus**, *Communications of the ACM 31,3:300-312* (March 1988)

[SALTZER 78]

Saltzer J H; **Naming and Binding of Objects**, *Operating Systems An Advanced Course (LNCS 60)*, Springer-Verlag (1978)