Customer Technical Note

OOA 97

Issue:	*1.0*	Ref:	*KC/OOA/CTN/53*
Authors:	*Ian Wilkie, Colin Carter* *Paul Francis*	Date:	*May 1997*

KENNEDY CARTER

14 The Pines Broad Street Guildford GU3 3BH

Telephone: +44 1483 483200 Facsimile: +44 1483 483201

Email: info@kc.com Web Site: http://www.kc.com

Table of Contents

1. Introduction

At the 3rd Annual Shlaer-Mellor User Group Conference (SMUG 96), Kennedy Carter presented a series of proposals [Wilkie 96-2 - 96-5, Carter 96] for extensions to the OOA formalism. These extensions were proposed as the result of our consultancy and project experience over the last 7 years and in consultation with major users of the method.

Since SMUG 96 we have made some minor refinements to the proposals as a result of feedback that we have received and as a result of formalising support for them in the Intelligent OOA CASE tool[1]. It has been particularly gratifying to speak to users who had no previous contact with Kennedy Carter but who had had similar experiences of OOA/RD in practice and who had come up with very similar solutions. Readers who are familiar with the earlier documents will find no radical changes. Rather, they will find that some of the details have been clarified.

This document updates and gathers together these separate proposals as a single coherent “OOA 97” specification. This document therefore replaces these earlier documents. In addition, we have added a short section summarising the developments and various versions of the OOA/RD developed to date.

Many, but not all, of the concepts covered here are now supported in CASE tools. Readers are encouraged to consult their tool vendors for further details.

Document History

1.0	7th May 1997	First combined OOA 97 document

Acknowledgements

The ideas presented in this document are based upon ideas collected in many years of practical development work at Kennedy Carter, and the authors owe a debt to the many people who have been involved, in particular Mike Clarke, Dave Fletcher, Allan Kennedy, Adrian King, Chris Raistrick, Dave Walker, Ian Wall, Chas Weaver and John Wright.

The authors have benefited greatly from discussions with Hamish Blair, Howard Green, Neil Mason, Alex Morgado, Gary Newman, Don Stewart and Colin Tinker (all of GPT Limited), Chris Mayers of APM Limited, Steve Bucholtz, John Roder and Pat Sukrachand (all of GTE Government Systems), Alan Fletcher, Dave Reinfelds and Terry Ruthruff (all of Lockheed Martin) and Sally Shlaer, Steve Mellor and Neil Lang (of Project Technology Inc).

[1] Implementing support for a rigorous formalism, such as OOA, in a CASE tool that “understands” the method is a very powerful way of ensuring that all the elements of the formalism are consistent.

2. Summary of OOA/RD Method Versions

Since the publication of the first book on OOA by Shlaer and Mellor various clarifications and extensions have been issued. The intention of this section is to summarise these in order to set OOA 97 in context. Readers should note that it is not intended that this section in any way replaces the published definitions of these extensions and in case of doubt the original publication should be consulted.

A complication in defining clear versions of the method is that there have often been a continuous series of minor improvements that have been incorporated into, for example, training material or consulting work on real projects without there having been an "official" and published release. Nevertheless, it is useful and convenient to attempt to define such versions.

2.1 OOA 88

This method, described in [Shlaer 88] is confined mainly to Information Modelling with a minimal treatment of dynamic modelling and “design by translation”. It is unfortunate that even until very recently, many comparisons of different OO approaches have used this definition for the Shlaer-Mellor method and thus have failed to take into account the considerable developments since then.

2.2 OOA 91

The method as published in “Object Lifecycles: Modelling the World in States” [Shlaer 92], represents the first virtually complete description of OOA as we might recognise today. It differs from OOA 88 in the following respects:

- Minor Improvements to the Information Modelling formalism and notation such as Numbered Relationships.

- Substantial definition of dynamic behaviour in terms of interacting state machines executing models represented by State Transition Diagrams and State Transition Tables. Included with this was a treatment of the rules of synchronisation and concurrency within an OOA model.

- Introduction of the idea of domain partitioning with an outline treatment of bridges.

- A discussion of the Software Architecture that emphasised the idea of translation as system construction approach.

2.3 OOA 92

As published, OOA 91 left some gaps in the definition of the method. While developing CASE tool support that understood the formalism Kennedy Carter extended the definition and notation into the areas that were previously undefined. In particular:

Information Models

The following changes were made:

- The case of a general n-way relationship that had briefly been mentioned in OOA 88 but omitted from OOA 91 was withdrawn completely. Our experience was that such relationships are very hard to understand and the real-world issues that they represent are better described by a number of simpler relationships.

- The notion of an "attribute domain", which was informally described in OOA 91 was tightened to include the idea of a data type with optional constraint.

State Models

In order to make the STT representation a complete description of the state model the following were added:

- A row representing the state of an instance before its creation. This pseudo-state enables the STT to show creation transitions. In addition, it allows the analyst to distinguish between the arrival of an event targeted at a non-existent instance being an error ("Cannot Happen") and being expected ("Ignore").

- The addition of the effect "Meaningless" for events arriving for instances in a state where the instance is deleted.

Other changes were:

- The notion of a polymorphic event was formally introduced (although at least one example of such an event had been previously published in training courses). Events sent to supertype objects are automatically available to all subtype state models. The analyst must then specify whether they are used (i.e. cause some transition) or are simply ignored. The Object Communication Model was enhanced to reflect the polymorphic transmission of events.

- The idea of “synchronous services” were introduced. Such services specify processing that is executed synchronously with respect to the invocation and can return data to the invoking state action[2]. OOA 97 refines the ideas and introduces the formal association of such services with objects and with object instances.

- Self directed events go to the head of the event queue for an instance

Process Models

[2] Shlaer and Mellor have recently published a paper adopting the ideas of Synchronous Services developed by Kennedy Carter [Shlaer 96].

- The Action Specification Language (ASL) [Wilkie 96-1, Wilkie 96-6] was introduced as an alternative to Action Data Flow Diagrams (ADFDs) for specifying process models.

Domains and Bridges

- Introduction of an “OOA of Bridges” describing all the possible bridges mappings that can exist between OOA domains [Raistrick 94].

2.4 OOA 96

The “OOA 96 Report” [Shlaer 96] tidied up a number of a number of loose ends in the description of OOA 91 and introduced some additional concepts. Some of these concepts had previously been described in Project Technology training material, but not incorporated in an “official” description of the method.

In outline the areas addressed were:

Information Models

- Clarification of the ideas of mathematical and stochastic dependence of attributes. Introduction of the notation (M) for mathematically dependent attributes replacing the (D) notation used previously.

- Clarification of the idea of relationship loops and composed relationships.

- Clarification to the ideas of reflexive relationships and the introduction of a new special case of symmetric reflexive relationships.

State Models

- Notational distinction between identifying event parameters and supplemental data.

- Events can no longer be sent to terminators.

- Self directed events go to head of the event queue for an instance.

- Formal introduction of polymorphic events through the idea of a “Polymorphic Event Table”

- Clarification to the definition of the operation of the Finite State Machine mechanism.

- Occurrence of “Cannot Happen” at run time defined as an analyst error.

- Introduction of the new concept of “multiple assignors”

- Clarifications to the rules surrounding object instance creation and deletion

Process Models

- Process models are not longer allowed to access the “Current State” attribute of an active object (except in the special case of synchronous creation).
- Introduction of the “proper attribution” rule for transient data on ADFDs[3]
- Introduction into ADFDs of the ideas of “base processes” working on (possibly ordered) sets of data, thus formally supporting the idea of iteration.
- Changes to the allowable properties of different process types on ADFDs.
- Replacement of the “Timer” mechanism with a simpler “Delayed Event” mechanism.
- Introduction of the term “wormhole” to refer to the invocation of services provided by other domains.

2.5 OOA 97

OOA 97 is the main subject of this document. A number of enhancements are introduced by this:

- Refinement of the idea of synchronous services by formal association of services with domains, objects and instances of objects.
- Introduction of the additional FSM responses of “Hold” and “Shouldn’t Happen”.
- Introduction of exception handling mechanisms within the OOA formalism.
- Introduction of support from the formalism for both “Deferred” and “Dynamic” data types.
- Introduction of a comprehensive support for the definition of Bridges within OOA/ASL including the idea of “counterpart relationships”.

In order to support these ideas, the definition of ASL has been upgraded from ASL 2.4 to ASL 2.5 [Wilkie 96-6].

[3] Recent discussions on the electronic Shlaer Mellor Users Group have prompted Sally Shlaer to modify this rule to insist that all transient flows have well defined types (“Domain Specific Types”) rather than be attributes of objects as such. This modified rule is consistent with the approach taken by ASL.

2.6 Compatibility Between OOA Versions

With a few exceptions, every stage of the development of OOA can be seen as consisting of a set of *extensions*. Users of the method are free to choose whether to utilise each new feature or not. In particular, users are unlikely to use features that are not (yet) supported by their chosen CASE tool. Further, users must assess the risk of using a feature supported by their current tool but not by others. Some aspects of OOA 96 are not backwards compatible with earlier versions and users must decide the impact accordingly.

We believe that there are no features in OOA 97 that are not backwards compatible with earlier versions of the method.

3. Formalised Synchronous Operations

3.1 Introduction

In previous versions, OOA/RD concentrated on asynchronous communication between objects in a domain (in the form of events) and provided few firm details on the nature of the communication between domains. In [Wilkie 94] we introduced the idea of synchronous operations invoked between domains. These ideas were expanded in the course of consultancy to include such operations invoked within a domain and have been embodied in both OOA 92 and ASL.

Work on highly distributed systems in particular led us to conclude that these ideas required enhancement so that the precise nature of the services and their execution rules could be defined. This:

- Makes the formalism more "symmetric" with respect to the asynchronous event driven behaviour
- Supports the “distribution of intelligence” strategy for synchronous as well as asynchronous behaviour
- Allows more control over the efficient dispersal of behaviour in a distributed system

3.2 OOA 92 Support for Synchronous Behaviour

Support existed in OOA 92 for the ideas of synchronous operations in two forms:

- The ASL definition supported the notion of "functions" and "bridges"
- Within a domain, analysts were able to define "synchronous services"

In ASL, a "function" is a section of code that can be called via a parameterised interface. The caller waits until the code is executed and resumes execution after the point at which the call is made. If desired values may be returned through the interface. Functions can provoke further asynchronous behaviour by sending events, but the caller will not wait for the results of those events being processed.

A "bridge" in ASL is a special kind of function which can obtain visibility of another domain by means of the $USE directive. The bridge code can invoke functions provided by the other domain, and the caller of the bridge will wait for this execution. Bridges are discussed in much greater detail in later sections.

In the definition of ASL 2.4 there is not attempt formally to associate functions with objects or object instances although instance handles can be passed into functions. There is therefore no restriction on the naming of functions or the data that must be supplied to them. OOA 92 allowed users to define "synchronous services" a domain, which when using ASL became ASL functions. The definition of such services was associated with objects within the domain or associated with the domain as a whole. This distinction had no special significance in practice and was used as an organisational tool by analysts.

This situation was unsatisfactory for a number of reasons:

- There was no visible formalisation of the association of functions with objects in the ASL. This hindered the comprehensibility of the models.

- There was no formal notion of an "instance based" function in ASL or OOA. This meant that service invocations could not easily be dispatched correctly in an instance dispersed architecture with remote synchronous processing. Analysts were also failing to benefit from the clarity of modelling that could otherwise be obtained.

- Analysts could not take advantage of any idea of "external visibility" to control the maintenance of a domain.

3.3 Terminology

Within OOA the term "synchronous service" is used to indicate a section of processing that will on request execute, optionally return data, and then terminate. The caller of the service will wait while the service is being executed. In an OOA domain model there will be one "definition" and many "invocations" per "synchronous service".

Within ASL, "synchronous services" are implemented by "functions".

3.4 Synchronous Services Associated with Objects

Synchronous services associated with objects (as opposed to domains) are identified by:

<object key letter><service number>:<service name>

for example:

ROB7:find_number_of_idle_robots

SLOT5:find_number_of_free_slots

Subject to the rules:

- service numbers are unique for an object

- service names are unique for an object

- service numbers can be the same as event numbers for the same object

In ASL, synchronous services are implemented as functions with the same name as the service and the invocation of the service is shown with a syntax demonstrated by the following example:

```
     [idle_robots] =
ROB7:find_number_of_idle_robots[]
```

Where output parameters are shown in parentheses on the left on the assignment and input parameters on the right.

3.5 Synchronous Services Associated with Object Instances

Invocations of these synchronous services are directed to a specific instance of the object with which they are associated. The names and numbers of such services are within the same scheme as that for the previous section (in the same way as creation events and instance events).

The invocation of instance based services are required to supply the identification of the object instance to which it is directed.

In ASL, instance based service invocations appear thus:

```
     [<output parameters>] = \
          <KL><Number>:<Name>[<input params>] on
                                   <instance handle>
```

For example:

```
     [state] = ROB10:check_active[] on my_robot
```

Such functions require an enhanced definition as follows[4]:

```
     define instance function <function name>
     instance this:<object type>
     input <input 1>:<type 1>, ....
     output <output a>:<type a>, ....
          <asl using inputs, outputs and "this">
     enddefine
```

"this" is a handle pointing at the instance used in the "on" clause of the invocation.

[4] Function definitions such as this involve a slight asymmetry in ASL. For example, there is no "define state/enddefine" or "define instance state/enddefine" constructs. Such definitions are an implicit part of the OOA model. It is similarly intended that the analyst should be relieved of the burden of definitions for functions. It should be possible, within a CASE tool, to declare that the instance function exists and is based on "object X" without the need to write out an elaborate definition.

3.6 Polymorphic Synchronous Services

As with events directed at supertype objects, it is useful for analysts to invoke synchronous operations on supertypes and have these implemented by the correct subtype.

OOA 92 defines polymorphic event behaviour in the following way:

- Any event which is directed at a supertype object (i.e. with a key letter the same as that of a supertype) is automatically available to al subtypes on all hierarchies descending from the supertype.
- A subtype may choose to ignore such available events (by setting the whole response column in the STT to “ignore”) in which case the event will not be processed by the subtype at runtime.
- Any subtype which has not completely ignored the event will received and process the event at run time if the subtype instance is related to the supertype instance at which the event was directed when the event was generated.
- A single event generation (to the supertype) may result in multiple state machines responding at run time. This will happen for example if there are state machines at both the supertype and subtype level, neither of which ignore the event.

Synchronous operations provide similar behaviour with one important difference:

- Exactly one implementation of the invoked service will execute at run time. Since the synchronous operation can return data, it makes no sense to say that the service will run multiple times; There could be no sensible rules for the return of the data.

Thus:

- An instance based synchronous operation can be defined for a supertype. The implementation of this object can be either in the supertype, or in any of its subtypes in any hierarchy descending from it.
- The analyst must ensure that there is exactly one implementation that can be executed at run time for any possible pattern of subtype instances. Thus:
    - implementations must not exist in more than one hierarchy
    - within one hierarchy, implementations must not exist at more than one level in any descending branch

- At run time, when the service is invoked using the supertype handle, the implementation for the supertype or subtype (or subtype of subtype) is invoked depending on the model.

- The handle “this” is accessible in the implementation of the service and will be of the same type as the object with which the implementation is associated.

For example, consider:

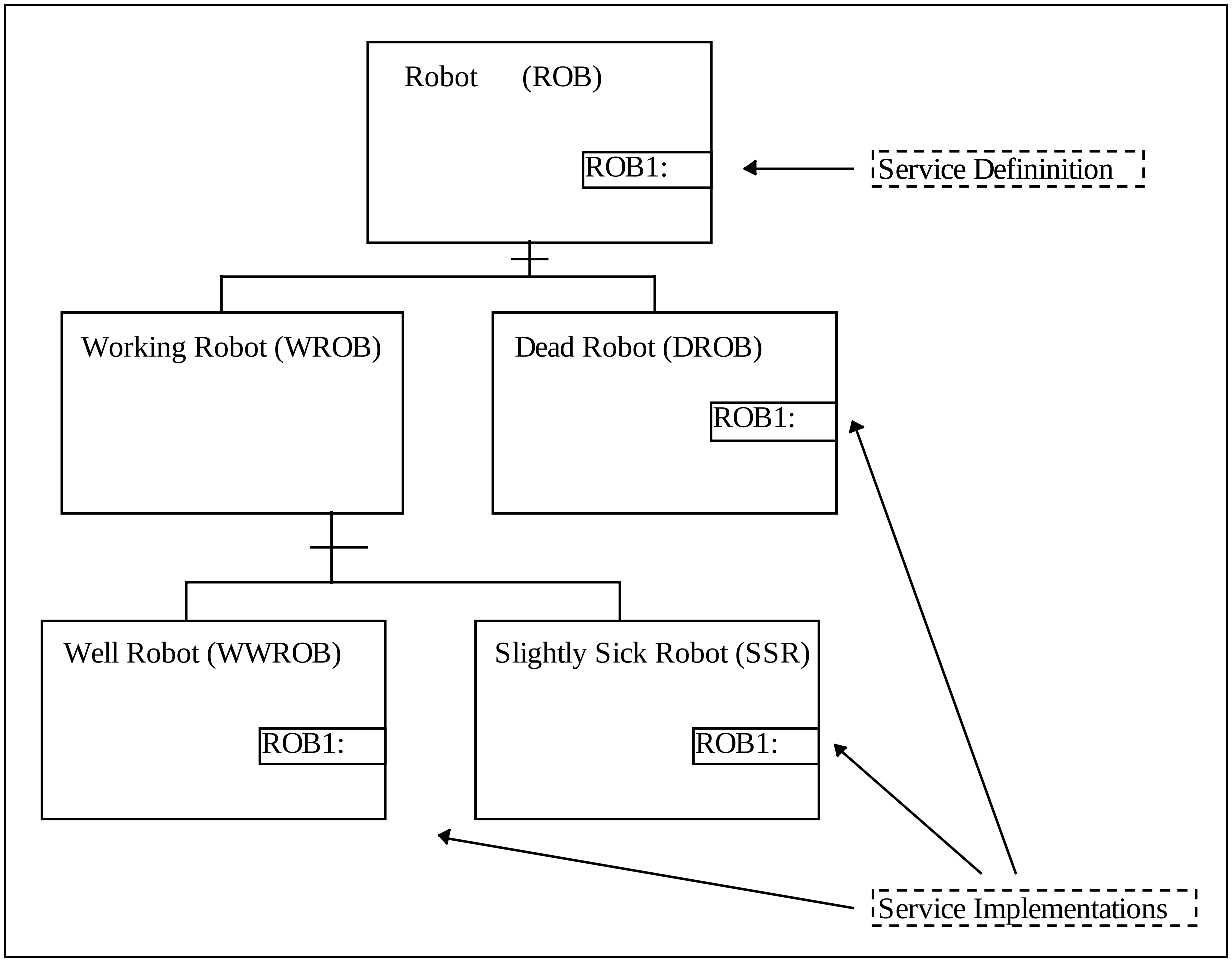


Figure 1 Example of a Polymorphic Synchronous Service

This shows (using an ad-hoc notation) a super-subtype hierarchy with an operation defined for the supertype object, which is implemented by a number of subtype objects. In this example, if a section of ASL performs:

```
my_robot = find-one Robot where .....
[x,y,z] = ROB1:Get_Position[] on my_robot
```

then if “my_robot” is a “Dead Robot”, the implementation of ROB1 defined for DROB will be called. In such a circumstance, the handle “this” in ROB1 will be a handle of type “Dead Robot” and will point at the instance of “Dead Robot” related to “my_robot” through the super/subtype relationship.

However, if “my_robot” is a “Working Robot” then one of the two implementations (for WWROB or SSR) will be called, depending on which of these latter subtypes exists.

In this example, if WROB also has an implementation of ROB1, then this is considered as a build time error.

3.7 Synchronous Services Associated with Domains

Some synchronous operations can be provided by a domain which are not associated with any specific object or instance within the domain. Indeed, it can be argued that all services provided by a domain (including event generation) should be invoked by such "anonymous" services. Such a style will promote loose coupling between the server domain and its use in particular project builds[5].

Within a domain, such services appear as functions with no restriction on their names (other than that they must be unique within the domain). Such functions have no concept of "this".

Note that domain synchronous services may, if required, stimulate asynchronous threads by generating an event.

The following syntax is used to represent a domain based service:

```
<Domain KL><service no>::<service name>
```

Where:

`<Domain KL>` is the “Key Letter” for the domain with which the service is associated (this is analogous to an object keyletter within a domain).

`<service no.>`is the number of the service within the domain

3.8 External Visibility of Services

It is desirable to formally control the visibility of synchronous services from outside a domain. If an analyst intends that a service be invoked from outside then the service must be shown as being invoked by a terminator on the OCM

[5] In object oriented architectures it is often required to associate all functions with objects. In such a case the architecture could have an object corresponding to the domain, providing all the domain based services as methods.

This allows the domain analyst to restrict access to the domain to known interfaces and thus reduce undesirable coupling.

3.9 Visual Representations

It is of course highly desirable that the overall pattern of synchronous as well as asynchronous communication between objects should be made as visible as possible. In OOA 97 the object communication model (OCM) is extended to include synchronous invocations[6]:

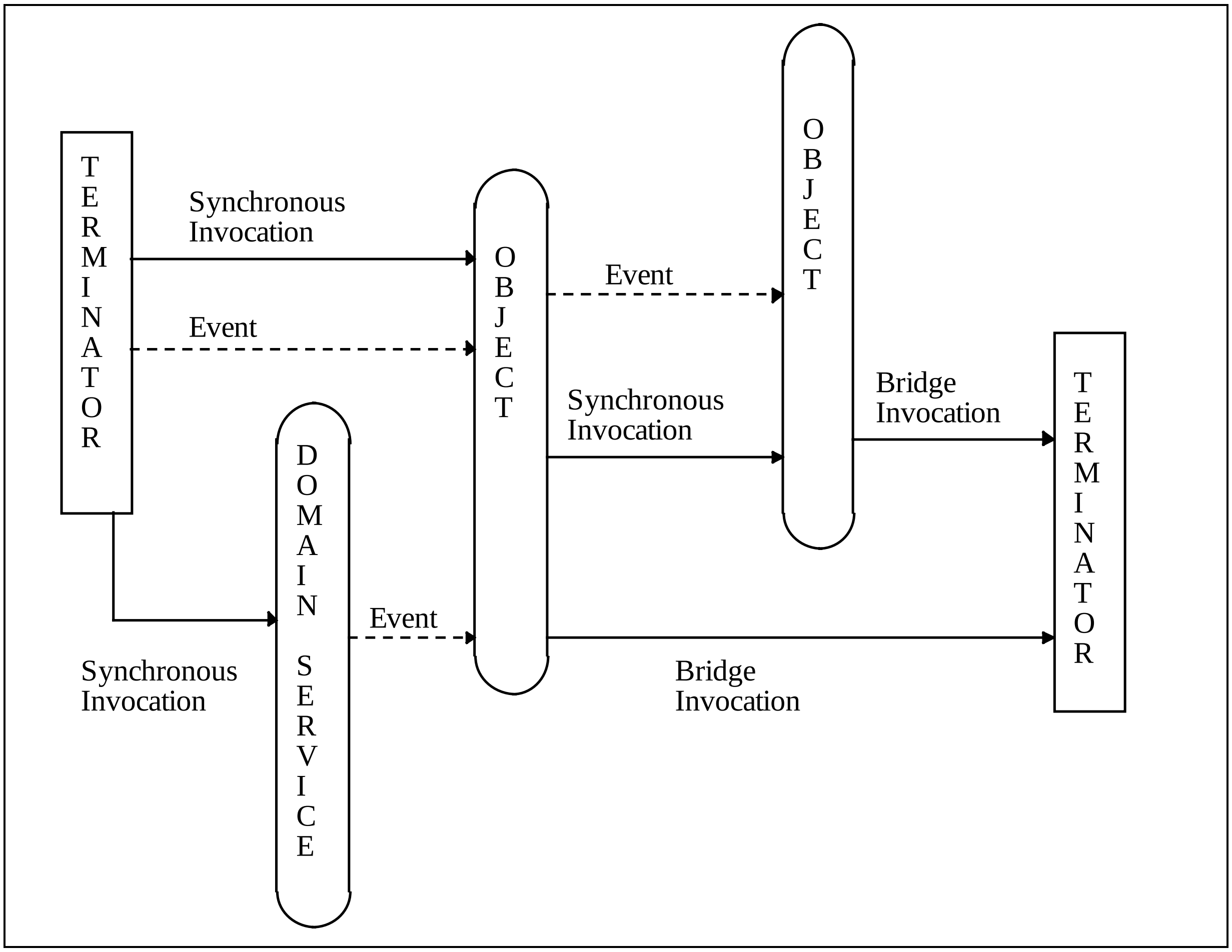


Figure 2 Synchronous Operations on the OCM

Note that it is not possible from this diagram to tell whether an event is being generated or a synchronous service being invoked from the state model of an object or from a synchronous service that it offers. Such knowledge will be contained

[6] This idea is not as simple as it might seem. For example, if Object A invokes a service provided by Object B that send an event to Object C, should the event be shown as being sent by Object A or Object B ? There is no "correct" answer. OOA 97 attempts to find a pragmatic compromise, taking into account the purpose of the OCM.

elsewhere in a model and must be consulted in order to interpret this picture. This situation already existed with the OOA 92 OCM, in that the STD had to be examined in order to determine exactly what would happen on receipt of an event.

Note also that object will appear on this diagram even if they do not have a state model.

3.10 Execution Semantics

The asynchronous mechanisms provided in OOA 92 imposed relatively few restrictions on allowed run time behaviour, and OOA 97 adopts a similar approach.

In “standard” OOA 97:

- Instance synchronous operations acting upon the same instance but called from different state actions will not operate concurrently with each other.
- An instance synchronous operation acting upon an instance and called from one state action will not operate concurrently with a different state action acting upon the same instance.
- Instance synchronous operations invoked from different external synchronous actions but operating on the same instance will not operate concurrently with each other.
- No restrictions are placed on object or domain based services as such.

Note:

- By “acting upon”, we mean the subject of the “to” clause for an event generation and the subject of the “on” clause for an instance synchronous operation.
- Although we have said “will not operate concurrently”, more specifically we mean that the OOA model must not be able to tell if the actions are operating concurrently. Thus a “clever” architecture may assess the pattern of read operations and write operations and allow the operations to run concurrently so that the results are identical to the situation that would be produced if they were run serially.
- Even this relaxed set of rules allows model to dead lock at run time.
- This set of rules allows single task, single thread OOA 92 architectures to be compliant using normal in-line synchronous call nesting to implement synchronous operations.

- By “external synchronous action”, we mean an action external to the system (perhaps being invoked through an implementation domain) that waits for the action to be completed.

- Analysts may not assume anything about the relative ordering of synchronous operations and event processing, other than the restrictions on concurrent activity.

We do not exclude the possibility that specific architectures may impose further rules on the execution of such services, although it would be potentially dangerous for analysts to rely on such rules for the correct operation of their models.

3.11 Backwards Compatibility

OOA 97 continues to allow the use within ASL of services and invocations that do not conform to the formalised naming and object association scheme. Such services are treated as domain based services within OOA 97.

4. Additional State Model Responses

This section deals with two additional responses of a state model to incoming events:

- A “Hold” response
- A “Shouldn’t Happen” response

These are in addition to the existing:

- Transition
- “Ignore”
- “Cannot Happen”

The “Hold” response has arisen out of a large number of considerations but in particular the need to reduce the complexity of state models in certain situations. It should be noted that we feel that there is nothing described in this section that cannot be modelled with OOA 92. However, sometimes this can be achieved only by considerable model complexity. This is discussed in the following sections.

4.1 The “Shouldn’t Happen” Response

The “Shouldn’t Happen” response has arisen out of the assertion in OOA 96 that the occurrence of a “Cannot Happen” event at run time represents an analyst error. Our experience is different. We observe that in OOA 92 analysts used the “Cannot Happen” effect for two purposes:

- The analyst considers that it is a logical impossibility that the specified event will be processed in the specified state. This is the OOA 96 style “Cannot Happen”.
- The analyst recognises that the event *could* happen but chooses not to model a response in this domain.

In any system there are typically a huge number of failure modes that are possible. These range from hardware failure to deliberate misuse of the system. For example, normal operation of a vending machine will not allow the door to open before it has been unlocked by the system. The STT for the door might therefore indicate that “Door Open” cannot arrive before “Door Unlocked”. However, a malicious (and strong) user might *force* the door open, thus triggering the “Door Opened” sensor. The domain analyst may simply choose not to deal with this contingency explicitly.

The “Shouldn’t Happen” response can be used to indicate such a choice. The response to this occurring at run time will be architecture dependant.

4.2 The Problem of State Model Complexity

The OOA 92 formalism provides for describing the dynamic behaviour of a domain in terms of interacting state machines operating concurrently. There is no provision for a state machine synchronously waiting, to the exclusion of all else, for an asynchronous thread to complete.

The execution rules for consumption of an event arriving for an object instance allow only for the event to be ignored, cause a transition or raise an exception[7]. The state machine will execute one of these responses as soon as the current state action has completed.

Consider this example from telephony:

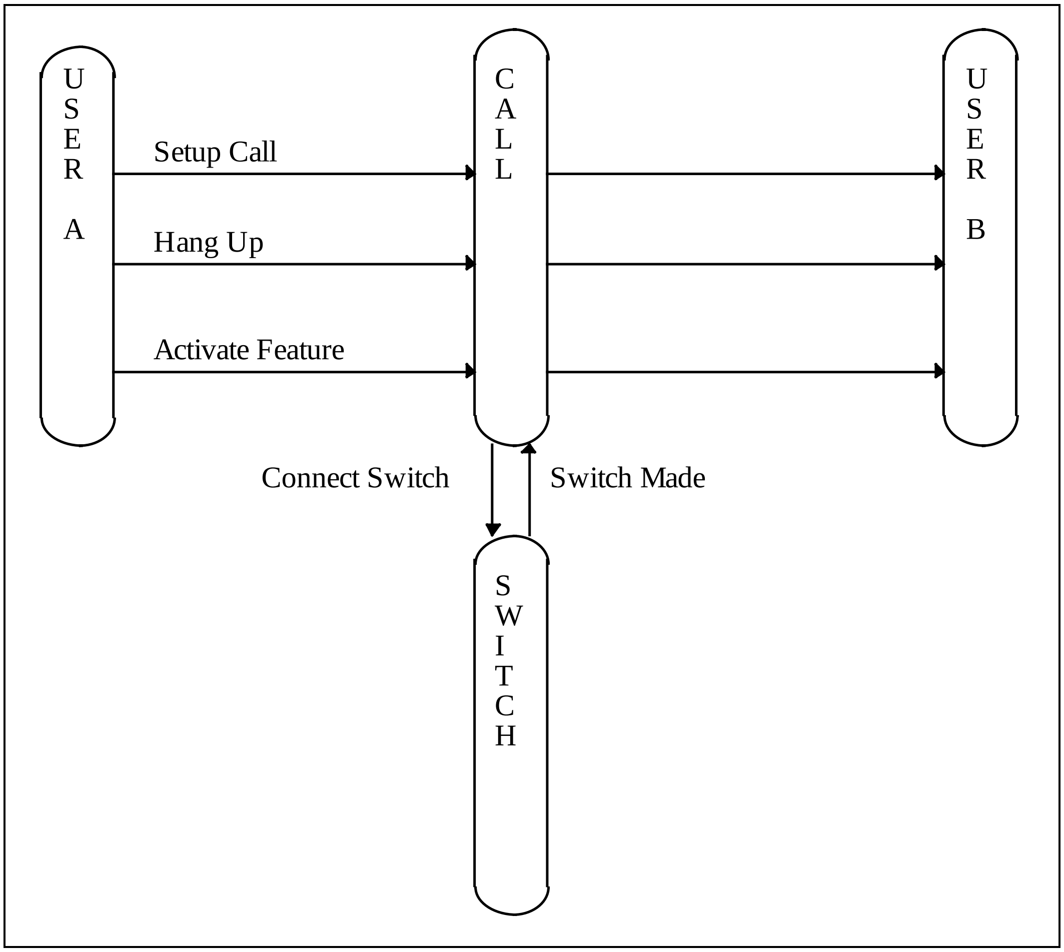


Figure 3 Example of State Model Complexity

[7] “Cannot Happen” or “Shouldn’t Happen”.

In this example, the state model for "Call" might look like:

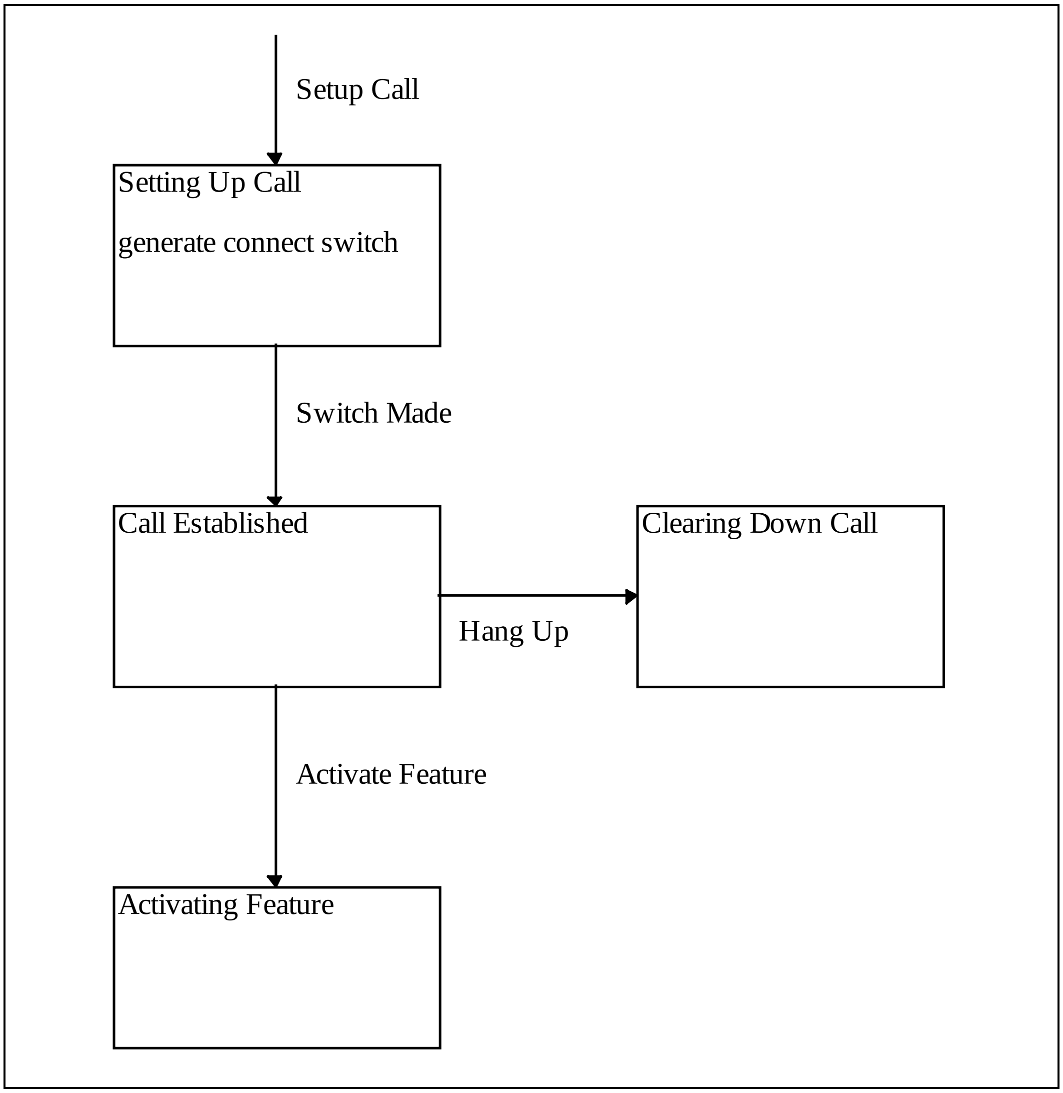


Figure 4 State Model for "Call"

Note that the dialogue with "Switch" must be an asynchronous one involving events, since the switch object must deal with the asynchronous nature of the switch fabric.

Now, consider what happens in the user hangs up while the state machine is in the "Setting Up Call" state. We would have to make a new state that could deal with clearing down a call when the state of the switch is not certain and new transitions. Similarly, if the user starts to activate a feature before the connection is fully made, we must somehow "remember" that this has happened in order to perform the correct operation when the switching is complete.

With a realistic telephony example, we could quickly suffer from a "state explosion". Instead, it would be nice to be able to wait until the switching is complete before dealing with any other activity. (Note that this would introduce a delay before "Hang Up" was responded too, but this might well be acceptable).

As a second example, consider the Project Technology OOA training course "car washing and fuelling" example which shows a very typical situation where we wish to move from State A to State B when both events 1 and 2 have occurred. However, Events 1 and 2 can arrive in either order (this is a facet of the problem domain). As a result the analyst must introduce at least 2 extra states and one internal event, the states being "1 has occurred, waiting for 2" and "2 has occurred, waiting for 1". This again introduces considerable complexity into a relatively simple situation.

As a third example, consider the Robot state model in the optical disk management system case study in which the Robot may be requested to carry out a disk transfer while in the middle of servicing an existing request. This requires an object, the "Disk Request" to remember the event and additional processing for the Robot as it terminates the previous request. Although the extra complexity added is not great, it is interesting to note that new OOA users often produce faulty solutions to this problem even if they are aware of it.

4.3 The "Hold" Response

In OOA 97 the possible responses in a cell in a State Transition Table (STT) are extended to include the response "Hold".

Consider:

Events States	E1	E2	E3
S1	S2	Ignore	Ignore
S2	Hold	S3	Hold
S3	Ignore	S1	S1

Suppose that when the instance is in State S2, the event E1 arrives. The queue will then contain:

E1

This event will not be removed from the queue for processing, but will remain. If event E3 arrives, then in the queue we will have:

E1 <- E3

Again, no event will be processed (the instance is still in state S2). Suppose then that E2 arrives:

E1 <- E3 <- E2

At this point event E2 will be removed from the queue and processed (even although it was not first in line). This will leave:

E1 <- E3

in the event queue. The result of processing E2 will have been to change the state of the instance from S2 to S3. As a result event E1 can now be processed (it is ignored) , leaving:

E3

on the queue and the instance still in state S3. Event E3 will now be processed moving the instance to state S1 and no more events in the queue.

Thus the execution rule for an object instance becomes: "When a state action is complete, the next non-held event on the queue will be consumed and acted upon".

We note that:

- The "hold" effect requires very little change to the notation and has a semantic that analysts find easy to remember.
- It is possible for analysts to develop models that might exhibit "live lock" type of behaviour, but this is no worse that the OOA 92 possibility of the analyst inappropriately ignoring events. In addition, the degenerate case of holding all events in a state can easily checked for.
- Whether an event is held or not is actually a property of the event/state combination rather than the event itself.
- There are a number of different methods by which an architecture might implement this behaviour and there are performance trade-offs to be made.

In the telephony example given in the previous section the analyst must simply hold the "Hang Up" and "Activate Feature" events in the "Setting Up Call" state to guarantee the correct behaviour.

5. Dealing with Errors and Exceptions

5.1 Motivation

All previous versions of OOA has the notion that models execute on a "perfect" architecture. Thus, events are always delivered, attributes are always available for access and so on. This assumption brings benefits in terms of simpler, more comprehensible and more maintainable models. However, it is not an idea which is fully sustainable in practice, especially when the software architecture is distributed.

A single task architecture can, of course fail. However, almost any (non recoverable) failure that can be imagined will result in the failure of the entire architecture. The problem of dealing with this therefore becomes one of dealing with a global re-start of the system.

In the case of a distributed architecture, there is the possibility that only part of the system may have failed while other parts may, in principle, carry on regardless. For example, we might have a situation where an instance of Object A is related to an instance of Object B. A failure of one task in the architecture might mean that the Object A instance is unable to access the Object B instance. In this circumstance it would probably be unacceptable to re-start the entire system since would partially defeat the purpose of having a distributed system in the first place. Instead, we would require the Object A instance to continue processing in a way that is appropriate to the failure of the other instance.

A further problem arises from the fact that part of the power in RD comes from being able, in principle, to map OOA models onto a wide variety of possible implementations employing different dispersal strategies. This means that in one architecture the Object A/Object B problem outlined above would never arise because the two instances are in the same task. In another architecture, however, the failure could occur and the model would be required to take some application specific action.

We must ask therefore, whether the domain analysts must be required to specify the response to a failure in every line of ASL, even although the 90% of them will never arise in practice with the chosen dispersal strategy. There is no simple answer. The aim of any support mechanisms inserted into OOA/RD must therefore be to enable users to achieve an optimal compromise.

5.2 Background to the OOA 97 Approach

In this section we deal in more detail with the various considerations that went into the choice of an exception mechanism in OOA 97.

5.2.1 The Need for an Explicit Mechanism

A question that must be answered initially is: "Do suitable mechanisms exist already within the OOA/RD formalism ?".

Consider the alternatives to a specific mechanism with defined semantics:

a. The architecture could send an event to the object instance that executed the ASL that provoked the exception.

b. The architecture could call an instance based synchronous operation provided by the object that executed the ASL.

Both of these suffer from the following problem:

- the state action or service call that provoked the problem will continue executing. This execution could be before or after the processing of a service call, depending on the characteristics of the architecture, but would certainly be before the reception and processing of an event. This may be unacceptable, and will almost certainly result in further problems being provoked.

In addition, “a” suffers from:

- the state model of the object (and potentially every object in the domain) would be required to respond to the “error” event in every state in the state model. Dealing with abnormal behaviour would therefore dominate the model.

This latter point might be helped a little by sending the event to a specific state model to deal with all exceptions. However, this would force the analyst to introduce an “error” object and either remove any ability to deal with the problem a context sensitive way or require the receiving state model to select an appropriate response depending on the contents of the event data. In a large domain this could lead to large switch statements in ASL doing a job that probably should have been done by the architecture.

Finally, any use of existing mechanisms would have the disadvantage that nothing would stand out in the model to say “this is special”.

5.2.2 The Choice Style of Exception/Error Handling

Having decided that some specific mechanism must be introduced, what is the best way to achieve this? Within the context of ASL, there are two ways that errors might be handled:

- Explicitly within the ASL using either return codes:

  ```
  status = link a R1 b
  if status = ERROR then
  ....
  ```

  or by setting some “global” error value:

  ```
  link a R1 b
  ```

```
if STATUS = ERROR then
....
```

- By some exception handling mechanism that operates “outside” the normal control flow of the ASL. For example in the ODMS Robot State Action for “Going to Source”

```
my_dt = find-any Disk_Transfer with \
            status = ‘Ready for Robot’
link this R9.”is executing” my_dt
source = my_dt -> R7.”moves disk from”
[delta_x,delta_y,delta_theta] =
find_displacement[this,source]
[] = move_robot[delta_x,delta_y,delta_theta]
my_dt.status = ‘In Progress’
```

Exception processing for “Going to Source”

```
# Something went wrong, try to get
# the robot back into a clean idle
# state. Since failure could have
# been in one of several places,
# we will need to try to undo all the
# things that might have been achieved.
my_dt = this -> R9
if my_dt != UNDEFINED then
  unlink this R9 my_dt
endif
.....
```

This exception code would be executed when an exception occurs as a result of any statement in the state action.

Of these, we consider that the second is preferable since it leaves the ASL for the state action without any “contamination” do deal with the errors.

The second approach also has the following advantages:

- the rules surrounding exception handlers can be set-up so that users are forced to deal with exceptions. Using the return code technique, it is much harder to verify that any kind of action has been specified.

- there is the possibility that the exception handler can be made sensitive to the actual problem that occurred without a risk that new exception types remain untrapped. By contrast,. if the return code approach was made more subtle by returning different enumeration values corresponding to different errors (e.g. ‘database access failure’, ‘memory allocation error’ etc.) there is a risk that when a new error type is introduced in the architecture (say ‘object store error’) some locations may fail to trap the error. At the very least we would have to modify many pieces of ASL code to recognise the new error.

5.3 Philosophy: When is an exception not an exception ?

OOA 97 introduces the concept of an “exception” or error within the OOA formalism. One must therefore ask for what activities might we expect an analyst or system designer to use it ? Specifically, should users be able to raise exceptions from within ASL ?

Consider a system which controls a nuclear power plant. Suppose that a state action of some state model is capable of detecting that the reactor core temperature has passed a safe limit. Would we expect the ASL to raise an exception and have the dropping in of the control rods activated by a piece of code in an exception handler ? Clearly not since, although the overheat is an undesirable and (hopefully) rare condition, reacting to it *must* be considered as part of the *fundamental operation* of the control system. By *contrast,* we have argued quite strongly in previous sections that software architecture problems can not be handled by the normal mechanisms.

In classic programming, it is often hard to “draw the line” and determine when it is appropriate to use an exception mechanism and when to use some other technique. We believe that OOA/RD provides the framework in which it become much easier to decide, and the remainder of this section will outline the justification for this.

We recommend that any model specific behaviour (whether “normal” or “abnormal”) relating to one domain should be addressed with the normal OOA techniques within that domain. However, where a problem is detected within one domain and that domain is incapable of taking the appropriate recovery action without knowing the context in which the executing activity is being used, then the problem can be reported to the invoking domain via an exception mechanism.

In other words, exceptions should be used for the situation where the contract[8] between the invoking and the invoked domains has been broken.

Consider some examples:

a. Software architecture failures. Here the SWA undertakes to implement the OOA formalism, which includes, for example, guaranteed delivery of events. During the operation of a distributed architecture it would be expected that individual event transmissions might be lost. However, the software architecture itself should not treat such a loss as an “exception”. Rather the architecture will perhaps re-try, or re-set some communications links or take other action. Only after the SWA has failed to be able to deliver the event will it raise an exception, to be caught by the application.

b. “Process Input Output” domain in the optical disk management system[9]. Suppose that application function “extend robot arm[robot id]” is mapped to

[8] See later sections on Bridges for discussions on the ideas of “contracts”.

the PIO service “set register[channel]”. The implementation of “set register”, could encounter a number of problems. Some problems such as:

- communications connection lost
- card off-line
- bus conflict

are likely to arise during the normal operation of the domain. When they occur, the PIO domain must take the appropriate action such as:

- re-open communications link
- put card on-line
- wait for bus contention to subside

and would be modelled as part of the normal operation. If the domain fails to remedy the situation, then an exception would be raised, indicating to the caller that the service provision has failed. In this way, the regular operation of a domain can become an exception for a domain invoking services from it.

Other problems such as:

- hardware failure detected
- architecture exception received

might cause the PIO domain to immediately pass on the problem to a higher domain. Indeed, a general policy for service domains might be to leave architecture exceptions unhandled, so that in the completed system they will be caught by the application domain.

5.4 Catching OOA 97 Exceptions

We use the term "catching an exception" to indicate the situation where an exception has been declared ("raised"), and some activity processes the result.

When an exception is raised, the normal processing of the model will cease (in the manner defined below), and an exception handler will be executed. On termination of

[9] This is the standard case study for the Project Technology authored OOA training courses. For more information contact Kennedy Carter.

the exception handler, the model will continue executing in some way, as defined in the rules that follow.

5.4.1 Execution rules for asynchronous threads

The execution rules are as follows:

- When an exception is raised, an exception handler will be executed and the remainder of the state action involved will be abandoned.

- Event queues will remain intact after the execution of the handler[10]. Specifically, any outstanding events for the instance that provoked the exception will be available for processing and any event already generated by the state action prior to the exception will be available for processing by their recipients.

- More than one exception handler in a system may be active at any one time. Whether this can happen in practice or not is architecture dependant.

- If an exception is raised while executing an exception handler, the current handler will be abandoned and the exception will be passed one level up in the hierarchy described in the next section. Thus, an unmanaged exception will never be silently ignored.

- Other processing in the model may well continue while the exception processing for the offending state action is carried out.

5.4.2 Exception handler definition for asynchronous threads

Handlers can be defined and associated with:

- A state

- An object

- A domain

When an exception is raised during the execution of a state, the handler associated with the state is executed. If this does not exist, that associated with the object is executed. If the object handler does not exist the domain handler is executed. If no handler is found then a default handler for the architecture will be activated. Note:

[10] Unless the operation of the handler causes an entire OOA transaction to be aborted and the instance data rolled back. The precise details of processing here are thus architecture dependant.

- Exactly one handler will be executing at one time for a given exception occurrence

- The execution rules are unaffected by whether the state, object or domain handler is called

- The execution rules for the default architecture handler are architecture dependant

- Handlers may call architecture dependant mechanisms to change the execution rules (for example to abandon an entire thread and roll back a transaction)

- As indicated in the previous section, if an exception is raised during execution of a state exception handler, the handler for the object will be called. This applies on up the hierarchy until an exception raised during execution of a domain handler will cause invocation of the default architecture handler.

The following information is passed to the various handlers:

- State handler:

  - Raised exception type

  - "this" for executing state machine (unless the state is a creation state)

- Object handler, as state handler plus:

  - State being executed

- Domain handler, as object handler plus:

  - Object executing offending state

  - "this" is not available

- Default architecture handler, architecture dependant but probably:

  - as domain handler plus offending domain

5.4.3 Synchronous Operations Called from Asynchronous Threads

When an exception occurs during the execution of a synchronous operation called from a state action the following execution rules apply:

- The synchronous operation is abandoned, and the exception handler is called for the operation.

- If a handler is not found, then the exception is passed to the caller. If the caller is another synchronous operation, the handler for that is invoked and so on.

- If the exception is passed back to the calling state, then this is treated as for state action exceptions described in an earlier section.

The information passed to a synchronous operation handler is:

- "this" (for instance based synchronous operations only)

Unlike handlers for asynchronous behaviour, synchronous handlers can pass back output parameters to their callers. Handlers can thus pass back:

- The output parameters defined for the operation that raised the exception.

Failure of the handler to pass back the parameters will result in the exception being passed back to the caller.

5.4.4 Synchronous Operations Called from Bridges

Exception handlers can be associated with bridges, in the same way as for synchronous operations. When an exception is passed up from the synchronous operation:

- The exception handler for the bridge will be called

- The remainder of the bridge is abandoned and control returns to the calling domain

- If the bridge has no exception handler defined, then the exception will be passed on to the calling state action or synchronous operation in the calling domain.

5.4.5 Asynchronous Operations Called from Bridges

If a bridge transmits an event into a domain, then only failure of the event transmission itself can be caught by the exception handler of the bridge. Once the event is sent, then exceptions are handled in the manner described under "Execution rules for asynchronous threads". Any failure of the contract between the invoking and invoked domains in this context must therefore be transmitted back to the invoking domain as an event.

5.5 Contents of OOA 97 Exception Handlers

Exception handlers can execute any ASL that is compatible with the context in which they are called. Certain data will be available to the handlers in a similar way to event data for state actions.

For example, exceptions raised from instances and handled at the service, state or object level will have access to "this". The type of "this" will be appropriate to the type of the object for which the handler has been called. Thus it will make sense for such a handler to execute:

```
this.colour = 'Green'
```

Since such code will exist in a handler associated with an object of which the attribute "colour" is an enumeration and of which 'Green' is a valid value. However, "this" cannot be made available to a domain handler since there is no valid ASL that the analyst could write such that the type of "this" can be reliably known.

Other data is available as described in previous sections:

EXCEPTION_TYPE	all handlers
EXCEPTION_STATE	all handlers except synchronous operations and bridges
EXCEPTION_OBJECT	all handlers

Of these, the state and object variables are enumerations with values determined by the structure of the model. Thus code such as:

```
if EXCEPTION_OBJECT = 'Reactor Core' then
     # Oops!.... do something quick!
     [] = scram_reactor[]
endif
```

is valid.

The "EXCEPTION_TYPE" variable is an enumeration that will always have the value 'Undefined' unless an explicit alias has been set up to map from the exception types that can be raised in server domains. This feature will allow handlers to take action that is sensitive to the nature of the exception without involving undesirable coupling between domains. Thus a handler in a particular domain might have code such as:

```
if EXCEPTION_TYPE = 'Re-Tryable Failure' then
     # Retry ....
endif
```

In a particular system build, this domain might invoke services from a RS232 comms domain which can raise a 'Timeout' exception. In this case, 'Timeout' could be mapped to 'Re-Tryable Failure'. In a different system build, the domain might use the services of another that has no ability to detect a timeout or other error that would indicate a Re-Try was appropriate. In this case, the condition shown above would never be true, and the re-try action would never be attempted. Like other cross-domain issues, we expect that as in this example, specific exceptions in server domains will be mapped to more generalised concepts in the client domain.

Exception handlers for states or objects are allowed to set the current state of the object instance directly. For example:

```
this.Current_State = 'Idle'
```

Note that the state action for 'Idle' will not be executed. Although an operation such as this would not normally be expected in ASL and indeed is incompatible with OOA 96, it is necessary in order to avoid state models that are dominated by exception processing.

Finally, it is expected that specific architectures will provide specific services to aid in exception processing. An example might be:

```
[] = abort_current_thread[]
```

Such architectural services might well override the standard execution rules for exception handling by, for example undoing the action of an entire OOA thread and flushing the event queues.

5.6 Raising Exceptions from ASL

As we have already indicated is considered undesirable to raise an exception within a domain and have it caught within the same domain. To enforce this idea, it is necessary to distinguish between operations called from bridges and those that are not. A construct such as:

```
if .... then
     raise 'motor failure'
endif
```

can thus only be used in the ASL of a service that is called from a bridge. Analysts must define an enumeration type (of which 'motor failure' is one value) describing all the possible exceptions raised by the domain in question. Execution of the "raise" statement causes the exception mechanisms to be invoked.

5.7 Exceptions raised by the Architecture

Exceptions raised by the architecture are on two types:

- Those that might be termed "analyst errors", such as a "cannot happen" effect being detected in a state machine, or an attempt to navigate from an undefined instance handle.
- Internal architectural problems such as a communications failure when sending an event.

Both of these classes of errors will be treated in the same way:

- On detection, the architecture will raise an exception
- Since there is no run time "bridge" between the OOA models and the architecture, the exception will be passed directly to the state action or synchronous operation provoking the problem.

Note that although exception handlers can be defined for object or instance based synchronous operation defined in a model, they cannot be defined for individual "intrinsic" ASL synchronous operations. Thus, although the construct:

```
my_dog.age = new_age
```

can be viewed as the invocation of a synchronous operation "update age" on the object "DOG", we do not allow definition of a specific exception handler for it. Thus, should an architectural exception occur while executing this, it will be passed to the invoking state or synchronous service.

6. Deferred Data Types

OOA 91 incorporated the idea of attribute "domains". Such domains capture, somewhat informally, the range of values that attributes can take on. Typical attribute domain descriptions might be:

"Magnet Current: 0-100 amps"

"Subscriber Address: Any address acceptable to the UK mail service"

OOA 92 formalised this with the specification of Data Types for attributes and ASL carries this through to all elements of the process modelling. In OOA 92 a number of base types are recognised along with user defined constrained versions of these.

This section deals with the idea of a “deferred type”, where a domain has attributes and data items of that type but does not carry the definition of the type or the operations allowed on it.

6.1 Motivation for Deferred Types

The idea behind deferred types is that it can often be useful in one domain to define the existence of data and the execution of operations on the data, without needing to specify the implementation of it. This has the following benefits:

- The domain using the type will be simpler and easier to understand
- A number of different implementations of the type and its operations can be considered without changing the original domain

Consider the following section of ASL in a hypothetical weapons management domain:

```
....
our_position = this.position
target = this -> R7."is currently targeting"
target_position = target.position
[range_to_target] =
        separation[our_position,target_position]
if range_to_target < weapon_range then
....
```

There are many possible ways in which the position could be held (polar co-ordinates, Cartesian co-ordinates etc.) and many possible reference frames (absolute, relative to own ship etc.). Each of these have their own advantages and disadvantages. However, in this domain we do not really care as long as we can find the linear separation of two positions.

It is convenient therefore to declare that the type of the "position" attribute is deferred, with the understanding that the implementation of its storage and the operation "[] = separation[]" is carried out by another domain.

Further, it can be even more convenient to overload existing operators within ASL, so that the above code section could be re-stated as:

```
     ....
     our_position = this.position
     target = this -> R7."is currently targeting"
     target_position = target.position
     range_to_target = target_position -
our_position
     if range_to_target < weapon_range then
          ....
```

Where, the "-" operator has been overloaded and has a special meaning in the context of the two deferred type operands.

6.2 Definition of Deferred Type Requirement

When we are performing the analysis of the “weapon management” domain in our example, it is possible that we will realise the requirement for the type before we have found an implementation. It will thus be necessary to document the nature of that requirement. Thus, in the weapon management” domain we have to:

- Document the nature and meaning of the type itself
- Document the nature and behaviour of the "separation" operation
- Document the nature and behaviour of the overloaded "-" operation

All of these are achieved through textual descriptions.

6.3 Realisation of Deferred Types in Implementation Domains

Often, deferred types will be used when there is an existing implementation for that type (for example a predefined message type). This section describes the issue involved in such implementation. It is important to note that the actual form of the implementation will depend on the architecture and target language being used. This section describes, therefore, the issues involved and an example in a hypothetical ‘c’ based architecture.

For each deferred data type that is to be implemented in an implementation domain, the system builder must:

- Define the data type implementation

- Define the operation implementations

For example:

Deferred Type	*Implementation*
`Position`	`struct *position_type`
`User_ID`	`int`

The architecture will then create an attribute of that type as part of the data structure for the containing object. Whenever an instance of the containing object was created, the architecture will call a user specified routine to be realised by the implementation domain. This routine will be expected to supply a value of the correct type that can be assigned to the attribute in the containing object.

Position

```
struct *position_type create_position() {
/* malloc structure and return pointer */
}
```

User_ID

```
int make_id() {
/* simply return value zero */
}
```

Overloaded operators are mapped:

Position "="

```
copy_position() {
/* Dereference Pointers and copy */
/* structure elements           */
}
```

User_ID "="

```
copy_id () {
/* copy integer arguments */
}
```

and so on. When the containing instance is deleted, an appropriate routine will be called:

Position

```
delete_position() {
/* Free storage */
}
```

User_ID

```
delete_id () {
/* Do nothing */
}
```

We note that:

- The exact format of the definition of these mappings is architecture dependant. Such implementation can bring benefits by exploiting existing function and class libraries, or by improving performance on certain kinds of operations. However, the obvious impact is that the implementation is likely to be non-portable between target environments. The most notable example of this is between the simulation and target environments. As a result, it may be necessary to build multiple implementations of the deferred type operations, with different system builds binding in different implementations. A possible compromise here could be that a simple ASL implementation could be used for simulation (see below), and something else used for the target. Such an ASL implementation might be fully functional (but slow), or might return only simulated answers.

- The architecture must call the creation and deletion routines for event parameters. The deletion routine will be called upon event consumption. The creation routine will probably be called upon entry to the generating state or function, since we may wish to avoid the overhead of copying the parameters as the event is transmitted. Further, the user will have to supply pack/unpack routines to enable the architecture to transmit the data over networks.

- The architecture must similarly support local variables of deferred type. Creation routines for all local variable of deferred type will be called on state or function entry (or immediately prior to first use) and deletion routines called upon state or function exit.

- The architecture must similarly support structure members

- It will be desirable if the code generation could be optimised so that "do nothing" routines (such as delete_id above) are not actually called at runtime. It will also be desirable if some of the simpler routines could be expanded "inline" in the generated code.

6.4 Realisation of Deferred Types in OOA Domains

In this situation we would have an attribute in one domain of a deferred type which becomes an instance handle in another domain. The attribute thus "points" at the counterpart in the other domain. Operations invoked on the attribute thus become operations on the counterpart instance in the other. This is very similar to the idea of explicit counterpart relationships described later in this document. We intend to explore this issue in later releases of the OOA 97 specification.

6.5 Realisation of Deferred Types Directly in ASL

In principle ASL can also be used to implement a deferred data type directly, without the need for an explicit OOA model. For example, we might have a attributes of type "widget factor", defined to be deferred. Thus will be implemented in another domain containing the statement that the actual type is "Integer" and where the various operations are implemented using ASL functions. For example:

```
define operation widget_factor_addition
input operand_1:Integer,operand_2:Integer
output result:Integer
     result = operand_1 + operand_2
enddefine
```

where the operation "+" in the client domain is mapped to this function. Since there are no explicit memory allocation or persistent data features in ASL (other than the creation/deletion of object instances), the software architecture must manage such issues entirely automatically.

7. Dynamic Data Types

It is common for service domains to have generic data handling properties. In these types of service domain a number of basic operations (e.g. =, >, <, ordered by) are required but the types upon which they operate are of no particular consequence to the domain.

The OOA 92 limitation of static typing both in the models and in ASL means that ASL must be written to handle every data type that may be encountered by the generic domain. Typically this adds supertype/subtype trees where the subtypes are data type specific, and adds large switch statements to the process models.

This section describes a scheme of dynamic typing which removes the need for these cumbersome structures in the OOA models.

7.1 Motivation for Dynamic Types

The idea behind dynamic types is similar to that of deferred types. Namely, that it can often be useful in one domain to define the existence of data and the execution of operations on the data, without particular concern for the data types being manipulated. This has the following benefits:

- The domain using the dynamic type will be simpler and easier to understand
- The domain can be maintained, modified, or reused with minimal effort

Consider the following section of an OOA model in a hypothetical data manipulation domain:

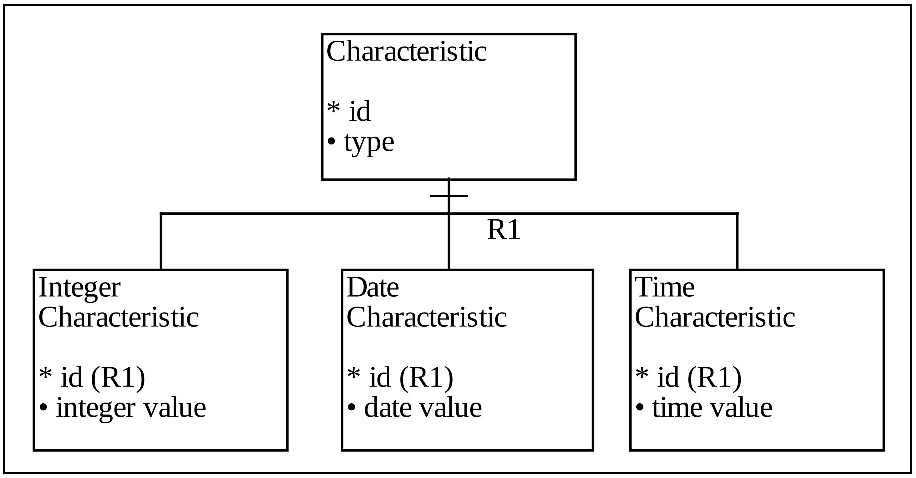


Figure 5 Hierarchy Required by a Generic Data Handling Domain

Assume the following ASL fragment appears in a state of the “Characteristic” object:

```
switch the_characteristic.type

case ‘integer’
     the_int = the_characteristic ->
                 R1.Integer_Characteristic
     the_int.integer_value = received_integer

case ‘date’
     the_date = the_characteristic ->
                 R1.Date_Characteristic
     the_date.date_value = received_date

case ‘time’
     the_time = the_characteristic ->
                 R1.Time_Characteristic
     the_time.time_value = received_time
```

```
   endswitch
```

The event which causes a transition into that state must carry event data for received_integer, received_date and received_time even though only one of them will be valid in any one event occurrence. The addition of a new type such as real (which may be dictated by a client domain) requires that the information model must change to add a new subtype; and every state which manipulates the objects must add an additional branch to the switch statement.

7.2 Characteristics of Dynamic Data Types

With dynamic data type:

- Analysts can create user-defined types which are dynamic
- Analysts can declare an attribute as being of a dynamic type
- ASL can access the current type of a dynamic data item

Consider the previous example, where the hierarchy is replaced by a single object:

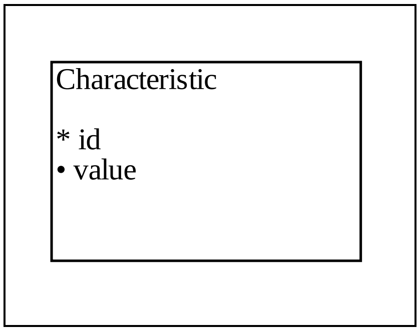


Figure 6 Revised Information Model for Generic Domain

In this domain the following are defined:

`a_dynamic_type` which is declared as a dynamic type composed of existing static types: Integer, Date and Time.

`value` The attribute `value` has the type `a_dynamic_type`

The ASL from the previous example would then condense to:

```
    the_characteristic.value = received_value
```

No subtypes are needed and no switch statement is required. The event parameter simply becomes received_value which is of dynamic type (a_dynamic_type).

A new operation, the “type-of” operation, is provided which returns the type of a data item as an enumeration value. Thus operations can be performed such as:

```
  {int_characs} = find Characteristic where \
                        type-of (value) = 'Integer'
```

The value returned by “type-of” is an enumeration type with a range of values determined by the type to which the operation is applied. Thus, the declaration of a dynamic type results in a further type being automatically derived which expresses the dynamic type as an enumeration type. This automatically derived is called `a_dynamic_type_enumeration` in this example. If required the analyst may create attributes of this type and therefore store the type of one attribute as another attribute.

The type-of operation can be applied to static data types, in which case it will return only one possible value.

7.3 Impact on Bridges

Use of dynamic data types generic data handling domains is only effective if bridges to and from more specific domains can be realised. For example:

A client domain may contain a synchronous service invocation of the form:

```
     [] = do_temp [temperature]
```

The corresponding bridge is defined:

```
     define bridge client:do_temp
     input temp : Real
     output
     $USE Data_Manipulation  # the generic domain
       [] = operate_on [temp]
     $ENDUSE
     enddefine
```

In the Data Manipulation domain:

```
     define function operate_on
     input the_value:a_dynamic_type
     output
       ......
       # obtain instance handle for the characteristic
       the_characteristic.type_of_value = type-of
the_value
       the_characteristic.value = the_value
      ......
     enddefine
```

In this example, the parameter of type Real has been passed to a function expecting a Dynamic Data Type. The architecture will therefore create a dynamic type in the server domain, populated with a real value. (The definition of the dynamic type must of course allow for a real value to be used).

7.4 Restrictions on the Operations Available in the Generic Domain

Any operations used within the generic domain must be available on all of the types within the enumerated list of types. Assignment (=) is available on all types. The comparison operators are available on many types and are commonly used in these types of domain. The ASL operation “ordered by” is also commonly required and available on a range of data types.

If arithmetic operations are to be used then the set of applicable data types must be limited to pre-defined arithmetic types (integer and real) and to user-defined types based upon integer and real.

7.5 Implications of Dynamic Types

- The new type raises no upward-compatibility issues. All OOA 92 ASL will still be valid.

- This addition introduces a meta-level concept (“type-of”) into OOA and ASL. Whilst meta-level capability is very powerful and can result in considerable benefits it also carries risks if it is abused.

- Additional runtime errors are now possible.

- Generated code size will be less and architectures should be able to achieve improved performance.

8. Bridges in OOA/RD

8.1 Introduction

As originally described [Shlaer 88, Shlaer 92], OOA 91 contained only a brief outline of the nature and properties of bridges. A later work [Raistrick 94] provided a comprehensive exposition of possible bridge mappings in terms of the OOA of OOA, and [Shlaer 94] covers some of the same ground in more detail. Finally, the Action Specification Language (ASL) provided a mechanism by which bridges of all types could be expressed [Wilkie 96-1]. However these documents have concentrated on the mappings to be defined at the expense of the relationship to the domain models involved. In particular, with the exception of ASL, few clear connections are made to the process models in the domains, and little attention is paid to the process by which domains are built.

This section attempts to pull a number of threads together, and present a coherent story of the nature and structure of bridges, methods for describing them and the processes by which they are built.

The discussion makes frequent reference to the Action Specification Language (ASL) and a number of the examples use ASL. However, the ideas expressed in this document are equally applicable with any other process modelling formalism such as Action Data Flow Diagram (ADFD's) or other specification languages.

Inevitably, this paper also covers some aspects of CASE tool support for the ideas described. This has been done with reference to Kennedy Carter's Intelligent OOA (I-OOA) product suite, but the principles are appropriate to any product.

Versions of ASL

Throughout this section we have shown examples that utilise ASL. For the description of the current support (Section 8.2), the ASL used is Version 2.4 as described in [Wilkie 96-1]. However, for the enhanced bridge support described later in the section we have used two sets of enhancements:

- Formalised synchronous services as described in Section 3
- Explicit manipulation of counterpart relationships in bridges (link_counterpart etc.) as described throughout the text.

These enhancements are part of ASL 2.5.

8.2 Bridges in OOA 92

Bridges and associated concepts were introduced into OOA/RD in Shlaer and Mellor's "Object Lifecycles" book [Shlaer 92]. The discussion in the book and also in Project Technology's course material is conceptual but does not provide much detail.

Kennedy Carter were well aware of this weakness in the method's coverage and took steps to provide more detailed supplementary information and examples. In 1993 Kennedy Carter enhanced its tool support for OOA with the introduction of I-SIM, the simulator product. Unique among OOA case tools it had support for bridges and allowed multi-domain simulations to be built and executed. As we described earlier, we have termed this level of support as being for OOA 92. The following sections illustrate the support for bridges in OOA 92.

8.2.1 Support for Bridges in OOA 92

A bridge function is specified for each outbound operation of a domain, that is an event or a synchronous service invocation.

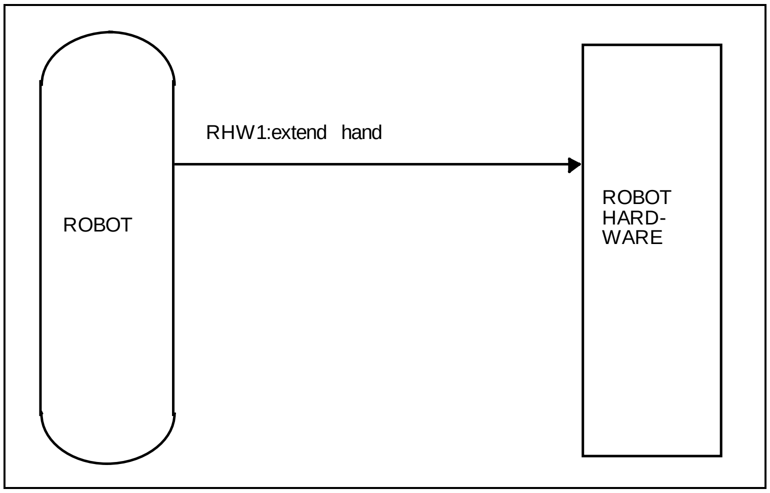


Figure 7 An Outbound Event

An outbound event from a domain will appear as an event directed to a terminator on the Object Communication Model (OCM). See for example Figure 7.

An outbound synchronous service call appears as a function invocation in the ASL of a state action or other synchronous service. There is no graphical representation of synchronous service invocations.

The bridge function is defined using ASL as follows:

```
define bridge ODMS:RHW1_extend_hand
input id_of_robot : Integer
output

$SINGLEDOMAIN
     # provide a stub for single domain testing
     the_robot = find-only Robot \
                         where robot_id =
id_of_robot
     generate ROB2:robot_move_complete () to
the_robot
$END

$MULTIDOMAIN
$USE PIO
     # A register set is the counterpart of a robot
     the_reg_set = find-only Register_Set \
                         where set_id = id_of_robot
     # The extend hand operation corresponds to
     # operation 5 on the register set
     selected_operation = 5
     generate
RSET3:update_reg_set(selected_operation)
                                   to the_reg_set
$ENDUSE
$END

enddefine
```

The header for the bridge function defines:

- the bridge operation that is implemented (RHW1_extend_hand)
- the domain invoking the bridge operation (ODMS)
- input and output parameters (id_of_robot). (Output parameters are not permitted for events) The input and output formal parameters must match the actual parameters in the invocation of the bridge function or the generation of the event to the terminator.

The body of the bridge function can contain up to two regions:

- A single domain region
- A multidomain region

The single domain region ($SINGLEDOMAIN/$END) allows the domain to be tested stand-alone. This region may contain any ASL and is only compiled in a singledomain build. In this example the single domain region is acting as a stub returning the event expected back from the PIO domain. This allows the thread of control in the ODMS domain to continue as if the PIO domain were present and behaving correctly. The single domain region may be left empty. The single domain region is not compiled during a multi-domain build.

The multi-domain region ($MULTIDOMAIN/$END) contains the formal bridge mappings. This region has some restrictions on the ASL it can contain. It also has a special ASL statement - the use clause. The use clause specifies the domain the bridge will map to (in this case use PIO). Inside the bracketed use clause, model items within the destination domain can be referred to. Such items are objects, attributes, events, functions and relationships. Inside the use clause model items from the source domain are no longer in scope. Input and output parameters and local variables remain in scope both inside and outside of the use clause. The multi-domain region is not compiled in a single domain build and is only compiled in a multi-domain build. The multi-domain region may contain many use clauses each specifying a different domain. Further, any individual domain may be the subject of more than one use clause within the multidomain section.

The above example illustrates a counterpart instance mapping and an event mapping. The example shows a common way of achieving counterpart instances. The Robot object within ODMS and the Register Set object within PIO are counterparts. As, at least, one of these has an arbitrary identifier then their identifiers can be arranged to match. Thus, for example, the Robot with robot_id = 1 is the counterpart instance of the Register Set with set_id = 1. As the identifier for the Robot (id_of_robot) is provided as an input parameter to the bridge function it is a simple find that is required to locate its counterpart instance.

The other mapping in this example is the event mapping. Some event mappings are one to one but in this case the event “extend_hand” maps to the general event “update_reg_set” with a specified parameter value. The event “close_gripper” in ODMS maps to the same general event with a different parameter value. The use of ASL allows a wide range of bridge mappings to be expressed.

Bridge functions appear the same regardless of the client/server relationship between the domains. For example:

```
define bridge
PIO:USER2_all_registers_in_set_are_zero
input id_of_reg_set : Integer
output
$USE ODMS
  the_robot = find-only Robot where robot_id = \

      id_of_reg_set
  generate ROB2:robot_move_complete () to the_robot
$ENDUSE
enddefine
```

Note that there is not a one-to-one mapping between the bridge function described here and the arrow between domains on a domain chart. In fact the mapping is many-to-many because one arrow may relate to many outbound operations from the client domain to the server domain and many outbound operations from the server domain to the client domain. In addition, as we have already indicated, one bridge function may contain many use clauses each to a different domain. It is for this reason that the bridge function is not attached to the bridge arrow in the I-OOA toolset.

Using this form of ASL bridge definition has allowed thousands of bridge functions to be successfully implemented using a variety of mappings in a broad range of projects. However, we recognise that further enhancements to bridge support are possible and desirable.

8.2.2 Limitations of Bridges in OOA 92

The use of ASL to define bridges means that all of the mappings are resolved at compile time. So the only mappings that can be expressed in the bridge function are static mappings, event-to-event, object-to-object, attribute-to-attribute, etc. Dynamic mappings cannot be expressed in the bridge function. The earlier example illustrated an instance-to-instance mapping which could be static or, more typically, dynamic. The instance-to-instance mapping was not held in the bridge function but was achieved through the use of compatible identifiers and identical identifier values in the counterparts. This can be thought of as a coincidental relationship between the counterpart instances. Whilst this mapping works, it suffers from several problems:

- It introduces an undesirable coupling between the domains - if an analyst adds an attribute to the identifier of one of the counterpart objects then the bridge will stop working.
- The use of the “find” operation will, in most architectures, produce very inefficient run time code.
- The analysts are required to write explicit mapping code for each operation to be mapped. This is time consuming and introduces additional possibilities for mistakes.

It would be more desirable to hold the counterpart instances as a property of the bridge itself rather than relying on matched properties of the two domains.

A further problem is that the definition of separate bridge functions hides the fact that sets of bridge functions may be related. For example the domain that invokes the bridge function “extend_hand” expects something in return. The behaviour of the domain is only valid if after invoking “extend_hand” it receives “robot_move_complete” some time later (or alternatively an error event to indicate that the robot failed to perform its task). This introduces the idea of bridge contracts (as opposed to contract bridge!) which is discussed in Section 8.4.

The partitioning of the bridge functions into single-domain and multi-domain regions has proved very useful. The value comes from the need to have different bridge properties at different stages in the project build. In the early stages it is necessary to build domains in isolation; in the later stages of a project all of the domains need to be put together. However, the use of two forms of bridge has proved to be inadequate for large projects where a number of intermediate forms of bridge function will be required. This has lead to a focus on project builds which identifies the set of domains and the relevant bridge functionality pertinent to a particular build. In its development a project may go through many builds - certainly more than two. Section 8.3.5 examines project builds in detail.

8.3 The Domain Chart

When developing large complex software systems the analyst typically has to deal with a number of distinctly different and unrelated subject matters. The strategy we use in OOA to organise these different subject matters throughout the software development process relies on the concept of a domain.

The definition of an OOA domain is:

> A domain is a separate real, hypothetical, or abstract world inhabited by a distinct set of objects that behave according to the rules and policies characteristic of the domain [Shlaer 92].

We can visualise each domain and how it interacts with the other domains in the system using a domain chart.

When one domain makes use of mechanisms and capabilities provided by another, we say that a bridge exists between the two domains. Traditionally, this has been represented on a domain chart as an arrow between the *client* domain (the user of the services) and the *server* domain (the domain providing those services). The direction of the arrow is from client to server - see the domain chart shown in Figure 8.

What this arrow effectively identifies, of course, is a *dependency* by one domain on the services provided by another. We can describe this dependency in very general terms, for example:

- **Air Traffic Control - Alarms**:

  The Air Traffic Control uses the Alarms domain to manage occurrences of trouble conditions.

- **Alarms - User Interface**:

  The Alarms domain uses the capabilities of the User Interface to present reports of trouble conditions to the operator. (The arrow going from User Interface to Alarms will be discussed in Section 8.3.3).

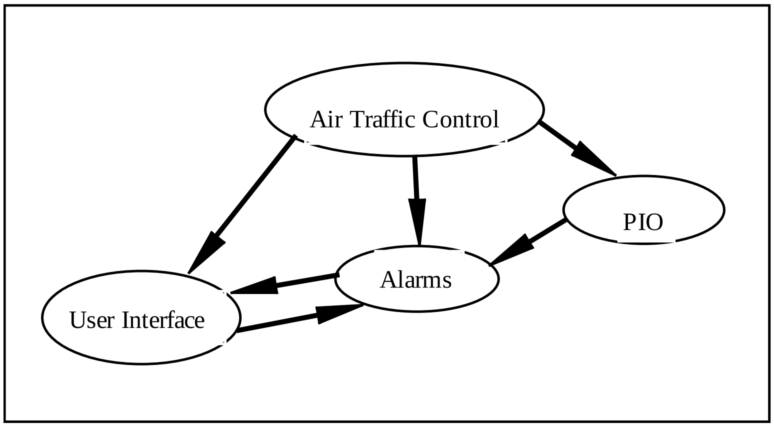


Figure 8 An Example of a Domain Chart

8.3.1 Dependencies

It must be recognised that each arrow on the domain chart is merely a ‘summary’ of the typically numerous and often complex mappings that hold between two domains. Furthermore, experience in developing real systems has shown that a single operation in a ‘client’ domain can sometimes be required to map to multiple operations in one or more service domains - an obvious implication of this being that it is not possible to associate an **actual** bridge mapping with a **single** arrow on the domain chart.

So what does the arrow on the domain chart actually represent ?

The answer is that it simply represents the general dependency by one domain on the services provided by another. A complex mapping that involves a single service invocation in the client domain to multiple services provided in more than one server domain therefore has to be represented as multiple arrows on the domain chart. This is reasonable since the client depends on the services of more than one domain.

OOA 97 therefore promotes the term ‘dependency’ (over the more conventionally used ‘bridge’) when referring to the arrow on the domain chart. The term ‘bridge’ is then retained to mean a specific mapping between a service request by a client and it’s satisfaction by one or more services provided by one or more domains. It should be noted that using this terminology, the “client” for a particular bridge operation may be at the “destination” end of the arrow on the domain chart.

8.3.2 Types of Dependency

We observe that there are a number of different types of dependency that exist. Consider the domain chart in Figure 8.

8.3.2.1 Service Dependency

When an application or service domain makes use of services provided by another domain we say that a service dependency exists between the two domains.

For each dependency on the domain chart:

- one domain will be the requirer of the service(s), and
- the other domain will be the provider of the services.

The domain requiring the service does not know (or care) how the provider of the service implements a service. The requiring domain is concerned only that the providing domain meets the requirements of the bridge contracts associated with the dependency. The idea of contracts is explored in Section 8.4.

Note that at run time, specific service invocations can be stimulated from either the requiring or the providing domain. Thus the Air Traffic Control domain will require the services of a GUI to get information to and from the operator, but at run time the GUI domain might well be the source of an unsolicited event that is delivered to the Air Traffic Control domain. In other words, the arrows on the domain chart do not show the direction of OOA thread propagation at run time. Rather they simply show the dependency that domains have on the existence of others in order to function correctly.

8.3.2.2 Architecture Dependency

Dependencies between the application/service domains and the Software Architecture domain are unlike those between the application and service domains themselves.

One difference is that architecture dependencies are typically executed at compile/build time (as opposed to run time for service dependencies). However, a more important difference is that architecture dependencies are specified with respect to the formalism of OOA, rather than that the elements of the domain models themselves.

Thus, architecture bridge mappings, in effect, make statements such as:

- OOA object becomes Class called <object name>

rather than:

- Robot becomes Class called “Robot”
- Online_Location becomes Class called “Online_Location”
- etc.

Further, the application and service domains will all have an architecture dependency. As a result, for the sake of clarity in complex systems involving many different domains, the architecture dependencies are often not explicitly shown on the domain chart.

Architectural bridges are not addressed in any further detail in this document.

8.3.3 Mutual Dependency

Where two domains require the services provided by each other they are said to be mutually dependent. However, their dependencies must, by definition be de-coupled from each other. As such, they are represented as separate arrows on the domain chart, and an informal description written for each one describing its nature.

For example, an Alarms domain will require the services of a User Interface in order to report alarm conditions. At the same time the User Interface domain may be required to manage the failure of display units and report these through the alarms mechanisms.

In these cases it is not helpful to refer to either domain as the ‘client’ or the ‘server’ since its role changes depending upon which dependency is being referred to. These terms are also much overloaded and have many different, often unhelpful interpretations. Thus we shall use the terms ‘service requirer’ and ‘service provider’ when referring to domains ‘linked’ via a dependency. Obviously the roles of a domain will still migrate between that of a requirer and a provider depending upon the context of the dependency.

8.3.4 Virtual Dependency

Sometimes a dependency exists between two domains that in a particular implementation (build), must be realised using two or more other dependencies on the domain chart. Consider the example in Figure 9.

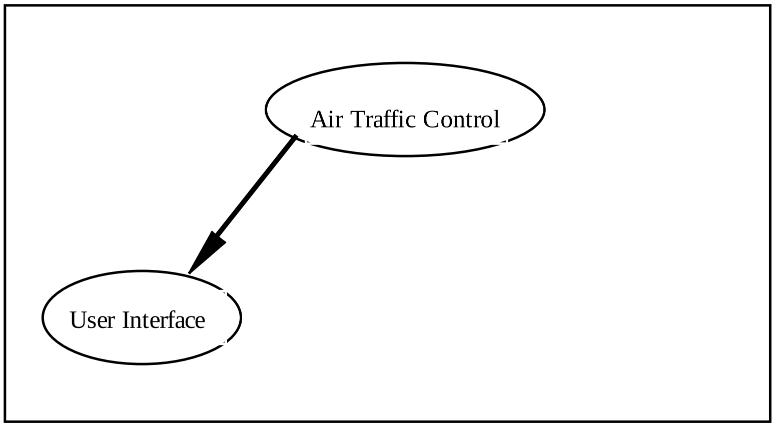


Figure 9 Dependency Requiring Virtual Implementation

The dependency on this domain chart might be informally described as:

- **Air Traffic Control - User Interface**:

  The Air Traffic Control domain uses the User Interface domain to present reports of air traffic flow to the operator.

This describes the real-world problem. However, let’s say that due the nature of the hardware architecture that the Air Traffic Control and User Interface domains are distributed across multiple processors connected together via a communications link, and that the software architecture itself cannot yet support such inter-process communication in a transparent way. In this situation the real-world dependency cannot be directly realised. In this case it is actually realised using the interaction of a third domain - the Remote Communications domain. As shown in Figure 10 this domain is responsible for sending and receiving messages via the communications link.

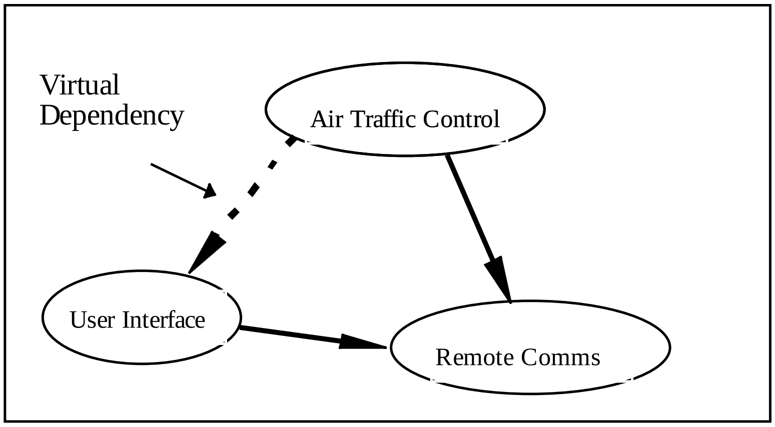


Figure 10 Implementation of Virtual Dependency

So for example, when the Air Traffic Control domain needs to update an operator's view of air traffic flow it does this by invoking (say) the 'update air traffic view' operation which gets mapped to the 'send message' service provided by the Remote Communications domain. This results in a message being transmitted across the communications link to the respective operator's terminal where (a different instantiation of) the Remote Communications domain receives the message and invokes (say) the 'new message received' operation which gets mapped to the 'update screen' service provided by the User Interface domain.

Thus, the real-world dependency is realised by a set of alternative dependencies and a number of additional domains. We use the term 'Virtual Dependency' for a dependency that is actually realised in such a way.

Although it can be argued that this is typically addressing an inadequacy of the software architecture, we have found it to be a useful pragmatic device.

Note that the dependency is not intrinsically virtual, rather, in certain builds it will be implemented as such. Equally there will be builds (for example in the early testing stages) where the dependency is directly implemented. The idea of project builds are discussed more fully in Section 8.3.5.

8.3.5 Project Builds

Experience in developing complex systems using OOA/RD over a number of years has emphasised the need for building such systems in a phased incremental fashion.

A number of factor influence the strategy that must be used in supporting an incremental build:

- Complex systems typically consist of multiple domains. In the early stages of a project however, it is unlikely that the system is understood well enough to be partitioned into a set of well defined domains - rather the system developers have some notion of a set of ‘candidate’ subject matters that the system must support. These of course will need to be realised as an integrated set of domains in the final delivered system.

- As the system developers incrementally add functionality within each build it is likely that the set of domains included within each build will correspondingly change.

- In addition to phase delivery, developers will often need to test or simulate individual domains or groups of domains. For example, in Section 8.2 we introduced the idea that the ODMS domain uses the services provided by the PIO domain to control the robot. Suppose that the actual robot hardware is not scheduled to be available at the point when testing of this functionality is planned, and in anticipation of this the system developers have built a ‘simulated robot hardware’ domain and a set of alternative bridge mappings between it and the ODMS domain (see Figure 11). The proposed build in this case includes domains and bridges that will not be included within the delivered system but nevertheless there is a need to specify the inclusion of them in one or more phases of the build strategy.

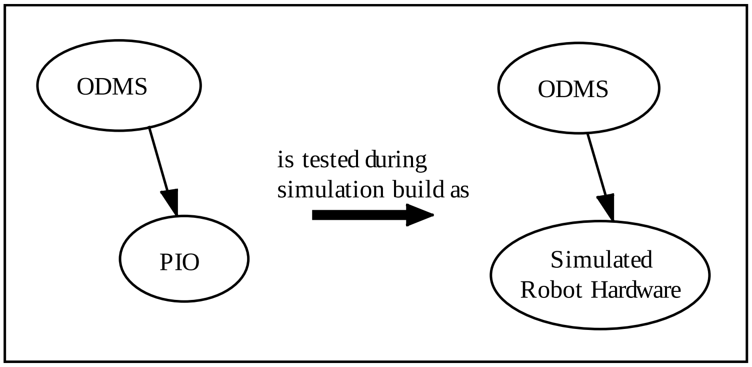


Figure 11 Alternative Domain Builds

We have noticed that project teams have adopted a number of different approaches to incremental system build and, upon examination, have realised that all exhibit features from two fundamental forms:

- Analyse, build, and test each domain separately. On completion the domain is delivered for (phased) integration with the other domains using the implemented bridges;

- A ‘thread-based’ approach is adopted whereby the incremental builds are geared around the implementation of a series of OOA threads. An OOA thread may result in processing in many domains, the thread of control crossing the domain boundaries via the bridges. Using this strategy the analysis models for each domain are built and tested in an incremental manner - each increment supporting additional OOA threads. The bridges are thus built in a timely fashion to support each incremental build.

Typically of course, properties of both of these approaches can be found on most projects. Thus for example, the integration of individually tested domains may not be with a single evolving domain set but rather there may be evolving groups of domains which are integrated with each other at a later stage.

It is, of course, important that the scope and content of each build be understood and that the correct bridge mappings be created and managed at each stage. Although the management of this process is not strictly a method issue, it is nevertheless recognised as being vital to all projects.

The remainder of this section discusses a framework in which the management of this process can take place.

8.3.5.1 The Specification of a Project Build

Essentially each incremental build of a system is made up of:

- a number of domains (each at a distinct version level), and
- a number of bridge mappings (each at a distinct version level)

In OOA/RD we already have a means of graphically identifying a set of domains and their inter-dependencies - it is the Domain Chart. So can we use it to specify the scope of a system build ?

Typically one would expect to see a single domain chart per version of a project. For that project version it would identify the set of domain versions and their dependencies (the bridges). Obviously what we need is a series of domain chart-like diagrams (which sit alongside the project domain chart), each of which defines the scope of the system for each incremental build level.

The rôles played by each diagram type are obviously very different, and in order to distinguish between them we propose to use the following terms:

- The Project Domain Chart
- Build Charts

8.3.5.2 The Rôle of the Project Domain Chart

The project domain chart fulfils the rôle of the conventional Shlaer-Mellor domain chart. Thus, it provides a concise graphic representation of a set of co-operating domains and their dependencies as required for the completed system.

8.3.5.3 The Rôle of a Build Chart

A build chart identifies a set of domain versions and their dependencies - note the important emphasis on the word ‘version’ here. Incremental builds of a system may well contain similar sets of domains, with some or all of those domains at different version levels.

It is important that each domain version can be analysed and ‘executed’ autonomously and as such it will require a set of ‘stubbed’ bridge implementations to be provided.

Against any specific build state of a project the system developers may develop formal bridge implementations which will be used to replace the stubbed bridges. In this way, alternative formal implementations of the bridges may be used in each build if required. Any bridge mapping not provided with its formal implementation will remain stubbed.

Thus, for any particular build state, the system developer may specify:

- the version of each domain on the build chart (each of which comes with a complete set of stubbed bridge implementations), and
- formal bridge implementations that replace some or all of the stubbed bridges supplied with the domain versions included within that build.

8.4 Contracts

When constructing a system composed of a number of domains, it will frequently be the case that a domain will be analysed that contains a number of outgoing service invocations on terminators and where there may be some time before the bridge to the providing domain is implemented, perhaps even by a different team. Similarly, analysts may wish to re-use an existing domain some considerable time after it has been created and documented.

It is therefore important that the nature of the interfaces both in and out of a domain are clearly documented. Often this will not simply be a question of documenting a single call or event, but can involve a sequence of invocations and returned events.

A *bridge contract* formally specifies the nature of the service requirement (in the case of an outgoing service invocation) or the service provision (in the case of an incoming service). This is distinct from a bridge *implementation* that specifies how the service requirements are mapped to service provisions.

Bridge contracts “belong” to a single domain, whereas bridge implementations “belong” to a combinations of domains in the context of a project build[11].

8.4.1 Contract Types

In order to make appropriate definitions of bridge contracts we must first understand the nature of the contracts that can exist.

We consider that there are three types of contract for a service requirement:

- Open
- Closed, Non-Blocking
- Closed, Blocking

An *Open Contract* is one where the invoking domain requests that the contract be honoured, but does not care or wish to be informed when the requested service has been executed. We might term this a “Fire and Forget” contract. Note, however, that:

- the invoking domain does trust that the service domain will *eventually* execute the service as required[12]

[11] For the purposes of single domain simulation, bridges are also implemented in the context of a single domain.

[12] In a distributed architecture, this trust may be violated. We will not address this issue here, but it is discussed in some detail in [Wilkie 96-4]

- the “Open” aspect of the request indicates simply that the operation of the invoking domain will not be affected by when the service is actually executed

An example of such an open contract invocation might be expressed as:

```
[] = OP1:inform_operator_of_train_arrival[train_id]
```

Here, the train management domain is simply sending a notification to the operator, and does not wish to be informed when the message has been displayed on the screen or the bell rung or whatever the chosen mechanism is. We should note that there may well be time constraint that emerge from the requirements for the application that say things like “*All messages must be visible on the operator’s display console within 1 second of the condition being detected by the system”*. Clearly, such requirements must be honoured and might well be part of the contract specification. However, the contract is still considered to be Open since the invoking domain is not notified of completion.

A *Closed Contract* is one where the invoking domain requires to be informed when a contract has been honoured (or indeed when it has failed). The invoking domain thus expects that:

- the service will be executed and the caller will be reliably informed of the outcome

An example of such a closed contract invocation is:

```
    [] = RHW1:extend_hand[]
```

where the caller expects to receive the event:

```
    ROB2:robot_move_complete()
```

when the hand has fully extended. In this case the invoking domain clearly does depend on receiving the response since it must not make a further action on the robot until the move is complete. To do otherwise would damage the robot mechanism and thus cause failure of the operation of the calling domain.

This example is actually a *Non-Blocking Closed Contract*, since the execution of the caller will continue (the invoking state machine will process events) while the robot is moving. The closure notification is delivered asynchronously to the caller. This delivery can be achieved either with the injection of an event into the calling domain or with the invocation of a synchronous operation on the original calling domain[13].

[13] This still constitutes an asynchronous delivery of the closure notification since the operation will have been called from an asynchronous thread in the domain providing the original service.

However, for simplicity will refer to the closure being notified by event for the remainder of this discussion.

Alternatively, the calling domain may require a blocking contract, where the invocation will not return until the requested operation has been completed. In the robot hand example above, the calling domain would expect the hand to be fully extended by the time the invocation:

```
      [] = RHW1:extend_hand[]
```

is returned. Note that we cannot tell that this is a closed blocking contract simply by looking at the invocation. We require a formal contract specification to tell us.

Only closed contracts can return data relating to the completion of the contract to the caller. However, this can be achieved either by a blocking or a non-blocking contract. In the blocking case, data may be returned as output parameters from the invocation and in the non-blocking case data may be returned in the closure event.

In summary then:

- In an open contract the caller is not informed of contract completion. No data can be returned to caller.

- In a closed non-blocking contract, the caller is notified by return of a closure event when the contract is complete. The closure event may carry data if required.

- In a closed blocking contract, execution of the calling state action is suspended until the contract has been completed. Data may be returned from the call if required.

These discussions apply equally well to services provided by domains. Such domains will publish available services, each of which will conform to one of the three contract models:

- An open service provision will not inform the invoker of completion

- A closed non-blocking service provision will return an event on service completion

- A closed blocking service provision will have completed the contract by the time the invocation returns.

8.4.2 Implementation of a Service in the Providing Domain

In the previous section we described three contract types that might be expressed both for a service requirement and for a service provision. We note that service domain analysts have some freedom in how such service provision contracts are met:

- An open contract provision can be implemented either by injecting an event into the service domain or by making a synchronous call that performs the required action. In the latter case the action will have completed by the time the call returns. In effect the service would be over-provided since the open contract does not require this to be true. However, such over provision still meets the contract and allows the analysts to change the implementation without affecting users.
- A closed non-blocking contract provision, can choose either to inject an event into the domain and have the resulting thread return the closure event or could perform the operation synchronously and generate the closure event immediately.
- A closed, blocking provision contract must be implemented by a synchronous operation on the providing domain.

8.4.3 Matching Required Service to Provided Service

Rather interestingly, it is not necessarily the case that a required service with a contract of a particular type must be implemented with a service provision of the same contract type. In fact:

- An open required contract can be met by either an open service provision or a closed blocking service provision.
- A closed non-blocking service requirement can be met by a closed non-blocking or by a closed blocking service provision.
- A closed blocking service request can, of course, be met by a closed blocking service provision. It can also, in principle, be met by a closed non-blocking service provision. However, this latter case would require the architecture to support the blocking of the calling invocation until the closure event is received from the providing domain. In effect the architecture would be implementing the state model for a “service invocation” object. Such a state model is shown in Figure 12.

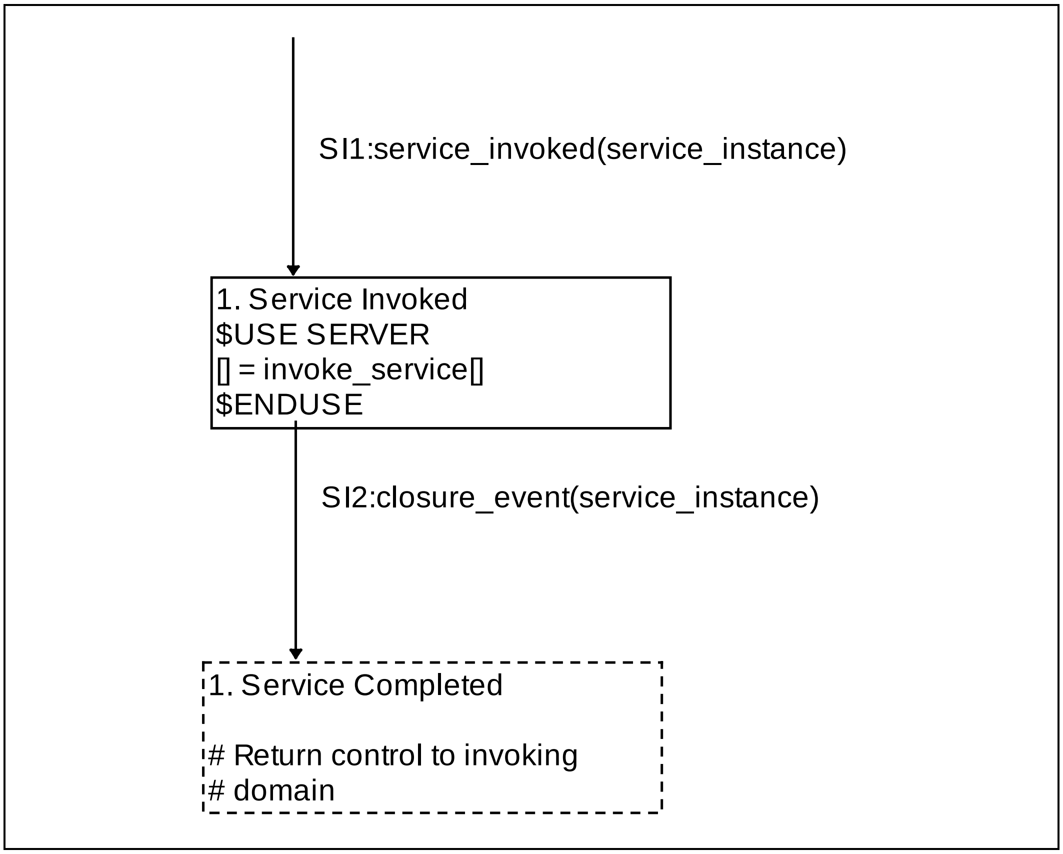

SI1:service_invoked(service_instance)
1. Service Invoked
$USE SERVER
[] = invoke_service[]
$ENDUSE
SI2:closure_event(service_instance)
1. Service Completed
# Return control to invoking
# domain

Figure 12 State Model for a Synchronous Invocation of an Asynchronous Event-Response Pair

In practice, we have not seen a real architecture which has implemented such a capability.

This flexibility of mapping means that one cannot tell from looking at the interaction with a terminator within a domain model whether the service is implemented in the providing domain(s) by synchronous operations or by asynchronous event passing. Indeed, different versions of the service domains may choose to take a different approach while still meeting the terms of the contract. It is for this reason ASL 2.5[14] requires that all outgoing service invocations in a domain are shown in the single style:

```
[] = <Terminator_Key_Letter><No.>:<Service_Name>[]
```

and not as events to terminators. Note that the above invocation should not be considered as a synchronous service call. Rather it is *a bridge service invocation*.

[14] Due allowance is made for backwards compatibility with older models.

That is: events are not sent to terminators[15] and neither are synchronous service calls made on terminators[16]. To do either if these so might make an unjustified implication in the mind of the analyst regarding the implementation of the service in the providing domain. Only by looking at the bridge contract can we be certain as to what happens when the service is invoked.

This flexibility also raises the question of what type of contract the invoking domain should specify for a given service. Clearly, the invoker should not impose unnecessary constraints on the service provision. Thus:

- If possible, the invoker should require an open contract

However, if the contract must be closed then should it be blocking or non-blocking ? Here we must be driven by the requirements of the requesting state model and the domain that it resides in:

- If the requesting state model must respond to asynchronous requests (events) while waiting for the service to complete then a non-blocking contract must be used. Otherwise a blocking contract can be used.

Using a blocking contract will simplify the nature of the invoking state model which may be an advantage.

An important issue which must be taken into account at this point is performance and responsiveness. Clearly in a blocking call, the invoking state machine will "block" until the call returns. This may be acceptable for the state model concerned, but may have a damaging impact on the domain or even the entire system depending on the architecture employed. If the architecture (or portions of it) are single threaded, then many object instances, and possibly the entire system may block while waiting for the call to complete. If the call is likely to be completed very rapidly this may be acceptable, but a long call could cause serious problems. Indeed, such a situation would violate the spirit of the OOA rule that state actions take their inputs, perform the actions and then end. In other words state actions should not persist for any extended period of time.

Thus:

- In deciding whether to use a blocking or non-blocking contract, system builders must take into account both the likely completion time of the service and the nature of the target software architecture.

[15] This idea is also in OOA 96.

[16] It can be argued that it would be better to define a new syntax to show bridge service invocations. However, on discussion with analysts it was felt that the syntax defined here should be used.

The “safe” default that would avoid this kind of domain pollution would be to decree that all closed contracts be non-blocking. However, such a stance might significantly complicate the invoking domain models.

Alternatively, we might insist that blocking contracts be used only where the entire execution of the service provision takes place with no interaction with the external world. However, such a choice would also represent a form of domain pollution, since knowledge of the precise implementation of the service domains is reflected in the invoking domain.

We recommend that analysts and system designers reach a pragmatic compromise on this issue.

8.4.4 Formal Contract Definitions

Clearly, the concept of a contract can capture a great deal of information about the nature of the required binding between domains. It will be necessary, therefore to capture this in some formal or semi-formal context. Such specifications will carry informal descriptions as well as more formal information such as the contract type, closure criteria and so on.

8.4.5 Definitions for Service Requirements

As we have previously indicated, both “ends” of a dependency will require a contract definition. In the invoking end, we must document the requirements of the service that is being requested. Such documentation is described in the following sections.

8.4.5.1 Open Contract

Open contracts will have a specification as shown in the following example:

Service Name:	OP7:train_arrived		
Input Parameters:	Train_ID	Integer	The identity of the train
	Arrival_Time	Time-of-Day	Official arrival time
Type:	Open		
Description:	Causes message to be displayed to operator. Message constructed from the input parameters.		

8.4.5.2 Closed, Non-Blocking Contract

Closed, non-blocking contracts will have a specification as shown in the following example:

Service Name:	RHW1:extend_hand		
Input Parameters:	Robot_ID	Integer	The identity of the robot
Type:	Closed, Non-Blocking		
Description:	This will cause the robot (identified by Robot_ID) to extend its hand to its full length. The calling domain will be notified when the arm is fully extended so that the next robot move can be initiated. Note that the robot hardware can fail, in which case the caller must be appropriately notified.		

Closure Notifications:

ROB2:robot_move_complete	Robot has successfully completed the action.
ROB99:robot_has_failed	Any failure to complete the action.

Note that the closure notifications make reference to services (ROB2 and ROB99) that are *provided* by the calling domain. These will have provided service contract definitions in their own right[17].

[17] These second contracts providing closure will themselves most likely be open or at least closed, blocking. However, we do not exclude the possibility that such closure contracts be closed, non-

8.4.5.3 Closed, Blocking Contract

Closed, blocking contracts will have a specification as shown in the following example:

Service Name:	USER4:get_pin		
Input Parameters:	ATM_ID	Integer	The identify of the cash dispenser
Type:	Closed, Blocking		
Description:	This will cause the cash machine to display a prompt message and solicit the users PIN.		
Output Parameters:	PIN	Integer	The entered PIN
	Cancel	Boolean	Indicates that the user pressed the cancel key.

8.4.6 Definitions for Provided Services

We now consider the definition of the other "end" of the dependency. In the providing end, we must document the service that is being provided by the domain. Such documentation is described in the following sections.

8.4.6.1 Open Contract

Open provisions will have a specification as shown in the following example (the :: notation signifying a domain based service):

Service Name:	TUI14::show _text_message		
Input Parameters:	Display_ID	Integer	Identity of the screen
	Message	Text	The message to be displayed
Type:	Open		
Description:	Causes the specified text to be displayed on the screen..		

This service provision provides an open contract because the server domain does not guarantee to have displayed the message by the time the service invocation has returned, and neither will it provide a return event to indicate success. Note that we are showing this domain based service name in the syntax suggested in section 3.

8.4.6.2 Closed, Non-Blocking Contract

Closed, non-blocking provisions will have a specification as shown in the following example (again domain based):

blocking. In this case there will be a third contract for the closure notification for this second contract and so on. Obviously great care must be taken to ensure that such threads eventually terminate.

Service Name:	PIO13::set_bit		
Input Parameters:	Shelf_ID	Integer	Shelf number in system
	Card_ID	Integer	Card number in shelf
	Register_ID	Integer	Register in card
	Bit_ID	Integer	Bit number in Register
Type:	Closed, Non-Blocking		
Description:	Sets the specified bit to 1.		
Closure Notification:			
CLIENT1:bit_set		The requested bit has been set.	
CLIENT2:bit_not_set		The requested action has failed	

Note that the closure notifications make reference to services (CLIENT1 and CLIENT2) that are *required* by the providing domain. These will have provided service contract definitions in their own right.

8.4.6.3 Closed, Blocking Contract

Closed, blocking provisions will have a specification as shown in the following example:

Service Name:	UI4::get_keypad_value		
Input Parameters:	Pad_ID	Integer	The identify of the keypad
	Num_Digits	Integer	Number of digit required
Type:	Closed, Blocking		
Description:	This will read the specified number of digits from the selected keypad and return them to the caller.		
Output Parameters:	Value	Integer	The entered number
	Status	op_enum	Indicates the status of the result.

8.4.7 Specification of the Mapping Between a Required Service and a Provided Service

Given an outgoing service request from a domain and one or more services provided by a number of service domains, we must specify the precise mapping from one to the other.

Such a mapping must take into account the requirements in the contract, in particular:

- The problem domain issues captured in the description

- The type of service contract (Open, Closed etc.) with due regard to the issues discussed in Section 8.4.3

- Any required closure conditions

- Mapping of data items.

The following example shows a mapping of the service requirement RHW1:extend_hand in section 8.4.5.2 to the provided service PIO13::set_bit in Section 8.4.6.2. The example uses ASL to show the mapping. We might equally show some tabular representation of such a mapping.

```
$USE PIO
shelf = 1
card = 1
register = (Robot_ID * 3) + 1
bit = 7
[] = PIO13::set_bit[shelf,card,register,bit]
$ENDUSE
```

Note that:

- This mapping is defined in the context of a “project build” as discussed in Section 8.3.5.

- We have not fully satisfied the contract specified for RHW1:extend_hand since we have not defined the mapping that will cause the desired closure events to be generated.

- This is a fixed rather than a counterpart instance mapping. We will show examples later which expand on this.

8.5 Terminators

So far we have used the concept of a "Terminator" without fully exploring the nature of the abstraction that it represents. In this section we will describe the basis for abstraction of a terminator and examine some of the implications.

Consider a domain that controls an industrial robot[18]. The domain will require capabilities of the robot hardware, such as the ability to move a hand and so on. We might represent this on an OCM as:

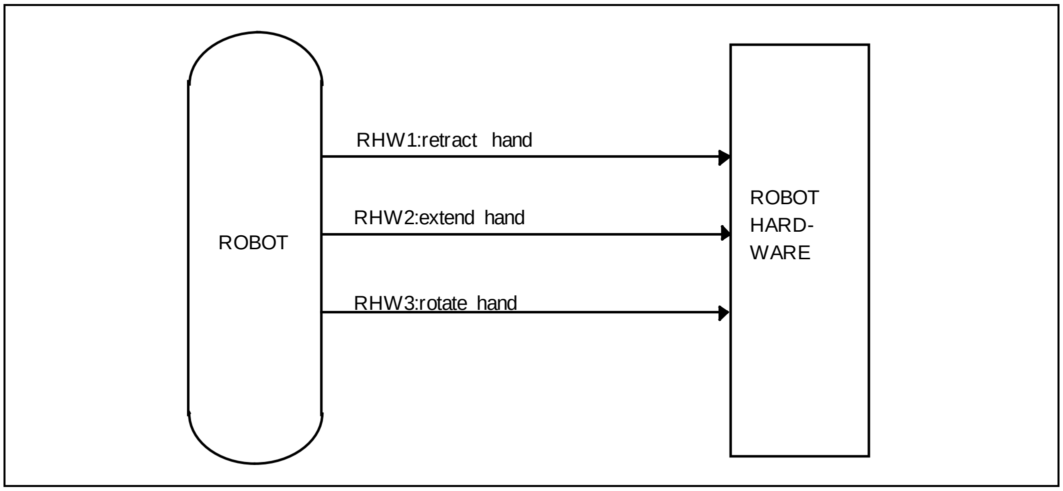


Figure 13 Interaction with a Terminator

Here we have captured the assembly of hardware that the domain controls as "ROBOT HARDWARE" and represented it by a terminator. The terminator has a name and a description (of which more later) and also a "keyletter" short form of the name in a similar manner to objects. The keyletter is used as part of the name of bridge service invocations. As was discussed earlier in section 8.4.3 the invocations (shown by the arrows on the OCM) represent bridge service calls rather than event transmissions.

8.5.1 Abstraction Rules

The following rules capture the basis for terminator abstractions:

- The terminator is an abstraction of an external entity being controlled or monitored by the domain.
- The terminator is named using the language of the subject matter of the domain.

[18] for example the Optical Disk Management System (ODMS) domain from the Project Technology case study.

- The terminator represents the ultimate destination or source of interactions with the problem domain and thus specifically does not relate to domains used to implement the communication with the external entity[19]

Thus, terminators capture the external entities in the problem domain from the perspective of that domain and do not make explicit reference to any other domains.

Thus we would expect to see:

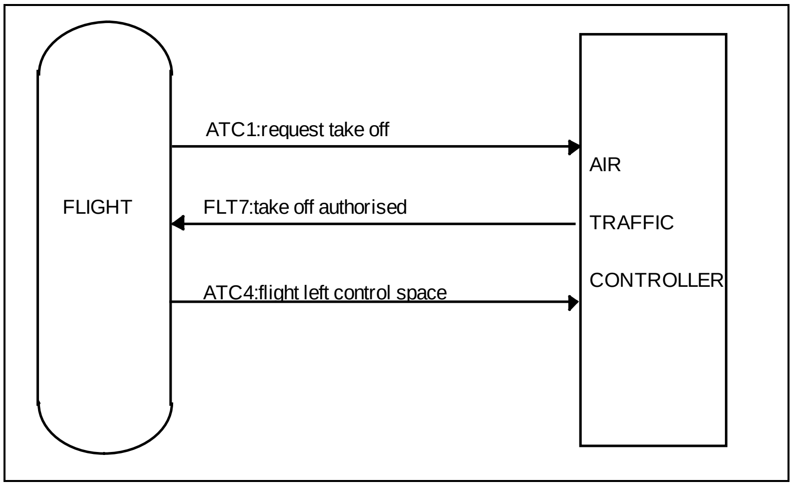


Figure 14 Terminator at Domain Level of Abstraction

rather than:

[19] This has an interesting parallel with the ideas of “ultimate source or sink” of data on an essential model context diagram in the Structured Analysis (SA) method. In SA, users were exhorted not to show details of implementation technology in the “essential” (analysis) model. However, in practice this was difficult to achieve since it required analysts to push understanding of complex implementation technology into the “design” phase. In OOA/RD the problem has been solved by requiring that a fully developed domain model be produced for the “implementation technology” domain.

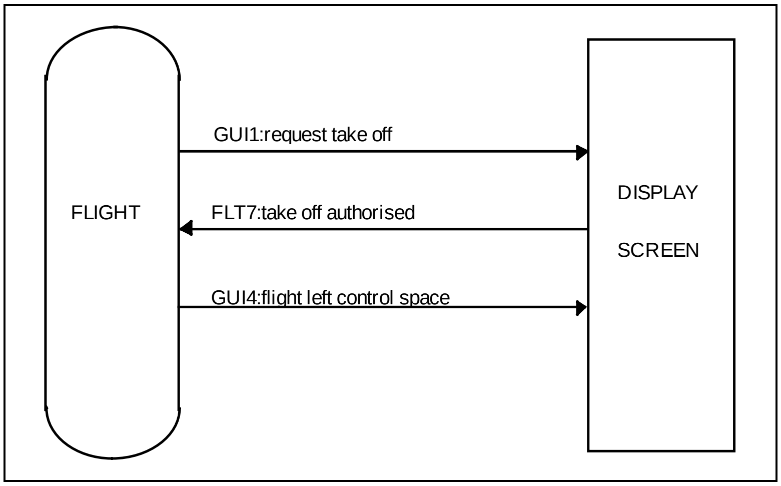


Figure 15 Terminator at Inappropriate Level of Abstraction

since this latter example make explicit reference to the Graphical User Interface domain.

The example we have shown so far relate to “real world” oriented domains where the terminators correspond to “kickable” entities. However, many of the domain models that we construct will deal with highly abstract subject matters and, particularly in service domains will have only a very limited view of their clients. In such cases we expect that terminators could be shown simply as “client” (with key letter “CL”):

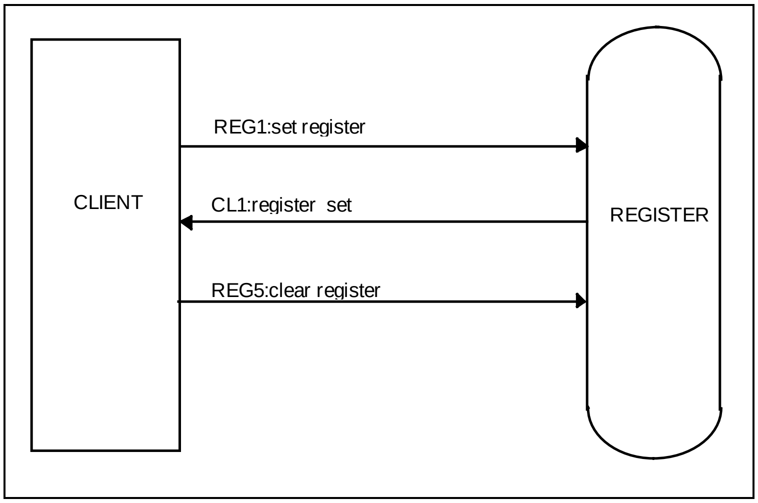


Figure 16 Anonymous Client Terminator

Thus, if we followed a thread through a number of domains, it might look something like:

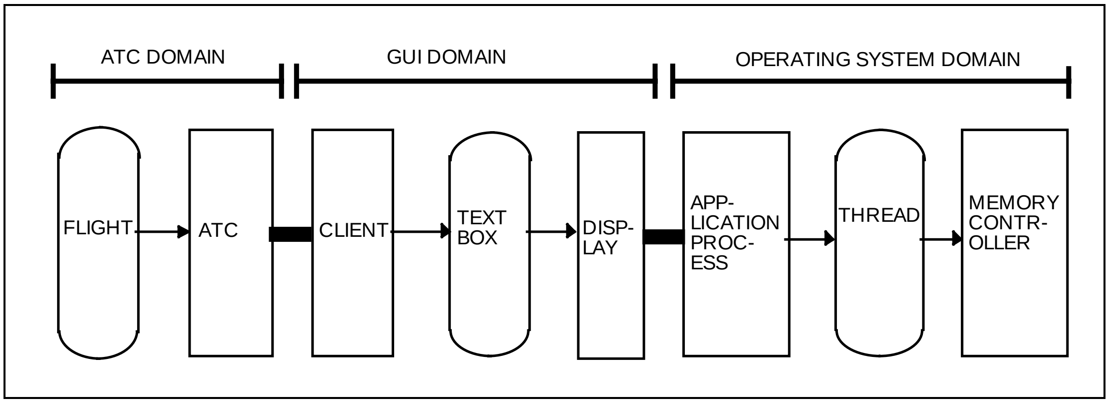


Figure 17 Layers of Abstraction

The names of the terminators linked through bridges in this diagram clearly shows the semantic shift that occurs between domains.

8.5.2 Benefits of the Terminator Abstraction

By divorcing a client domain from the details of the implementation domains we immediately benefit from a number of possibilities:

- System builders are free to change the nature of the domain used to implement a service without changing the nature of the client model. This will be acceptable as long as the contract required by the client continues to be honoured. Thus, in the Air Traffic example above we could replace a GUI domain by a Text Terminal domain without affecting the behaviour of the ATC domain.

- Similarly in the control of a robot, we may find that the robot hardware is sufficiently unreliable that it is necessary to detect and recover from hardware failures. In order to continue to meet the requirements of the contract with the application domain we could construct a "reliable robot" domain that can be inserted into the system without the knowledge of the original application.

- We can simulate the behaviour of an implementation without changing the client model. For example, we may wish to test out the Optical Disk Management domain by providing a "simulated robot" domain that simply records the desired new position of the robot and after some time delay returns a "Robot Done" event.

8.5.3 Terminator Descriptions

Just as with objects, we expect that analysts produce informal "descriptions" of terminators. These capture the basis of the abstractions as well as the outline characteristics of the terminator. These descriptions will be used by system designers along with bridge contracts in order to find or implement suitable service domains.

Here is an example terminator description:

> Terminator: ROBOT
>
> Key Letter: ROB
>
> Description:
>
> The external robot hardware controlled by this domain. The hardware consists of an assembly of components that are capable of moving disks around inside the cabinet. This domain may send commands to the robot to move various components (such as its hand), and some time later the hardware will return an event indicating completion of the move.
>
> Note: We expect that the robot is perfectly reliable and will always complete its move successfully.

This description addresses many of the same issues as the contracts for the individual terminator operations.

8.6 Bridges Mapping Types

So far we have considered only the straight forward bridge mappings of the type that are already supported by ASL. In this section we will consider the different types of mappings that can be used. These range from the simple mappings already discussed to the more sophisticated ideas of counterparts.

8.6.1 Simple Bridge Mappings

We characterise a simple mapping as one in which an outgoing invocation is mapped to some service provision in another domain, with the mapping involving a simple transformation and mapping of data parameters. An example of this is the mapping of the “extend hand” operation to “set bit” shown in Section 8.4.7.

This is to be compared with the ideas of counterpart mappings discussed below.

8.6.2 Counterpart Instance Mappings

In many systems involving a complex assembly of domains, much of the communication between domains will be between related pairs of instances. Such related pairs are called counterpart instances. In such cases it is advantageous to formalise the pairing for a number of reasons:

- The specification of the related instances can be made simpler since they need not manage the relationship explicitly
- The connected domains are not polluted with knowledge of each other
- Architectures can use the formalised knowledge to produce more optimised code

Since we are discussing cross-domain relationships, it will be evident to the reader that there must be both “supertype/subtype” relationships, and straight forward relationships. We term these as “Generic/Specific” and “Peer to Peer” respectively, and these are discussed over the next two sections.

8.6.3 Generic/Specific Counterparts

It is frequently the case in OOA/RD that service domains capture generic abstractions that apply to objects in many other domains in the system[20]. Sometimes developers will realise the need for such domains as the initial domain chart is developed[21].

[20] Such abstractions are at the heart of many OO based approaches. The notion of a domain in OOA/RD provides the context for the abstraction and provides a clear definition of the services on offer.

[21] In the telecommunications industry a very common abstraction is the notion of a “managed object”.

Often however, the need will become apparent as developers recognise the same pattern of object characteristics and behaviour in multiple places within each domain and across multiple domains.

Consider a system which controls a number of hardware items (Robots, Assembly Lines), uses a number of items as a part of its implementation (Displays, Serial Line Controllers) and has a software architecture that allows "on-line" adjustment and modification to components of the architecture (Tasks, Nodes). Suppose that it is a requirement of the system that we must be able to take any of the above items in and out of service for maintenance purposes. Further suppose that we wish to provide a common set of behaviour for such maintenance activities and a common interface.

In order to achieve the desired effect we might construct a maintenance domain containing objects thus:

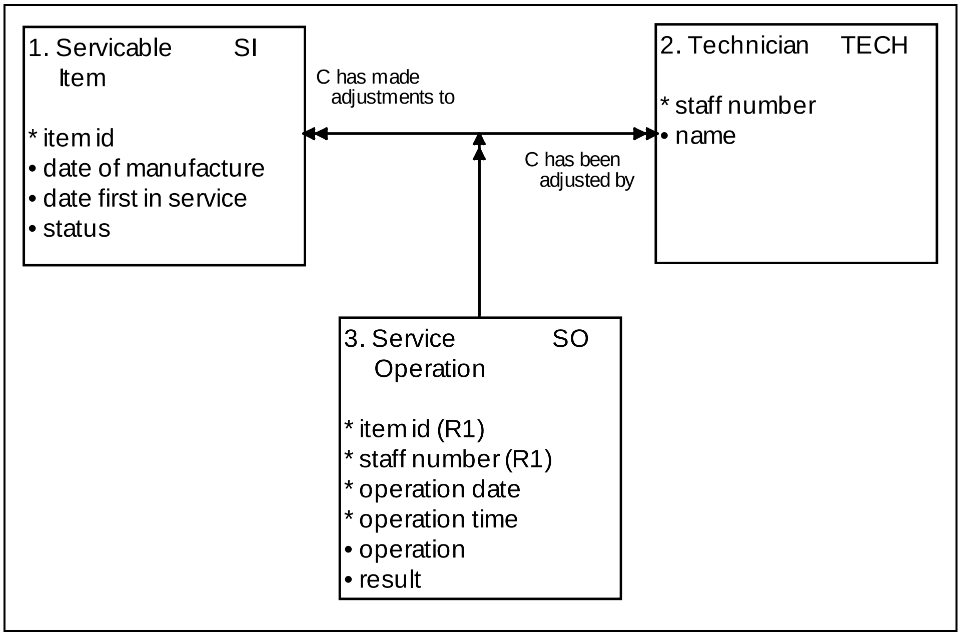


Figure 18 Information Model Fragment for a Generic Maintenance Domain

This domain deals with the generic abstraction of a "Serviceable Item", examples of which would be Robots, Displays and Tasks. Within the Maintenance Domain the Serviceable Item will describe the generic behaviour surrounding, for example, taking an item out of service.

Assembling this together on a pseudo-domain chart:

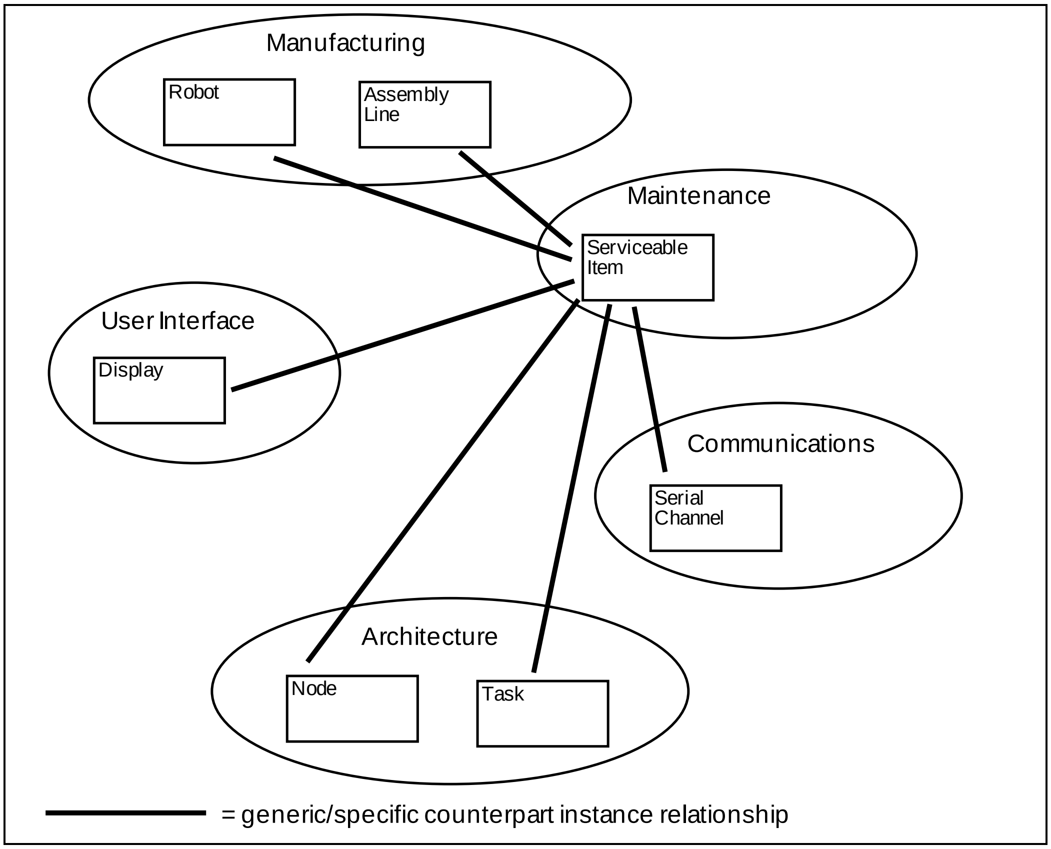


Figure 19 Domain Chart using Generic Domain

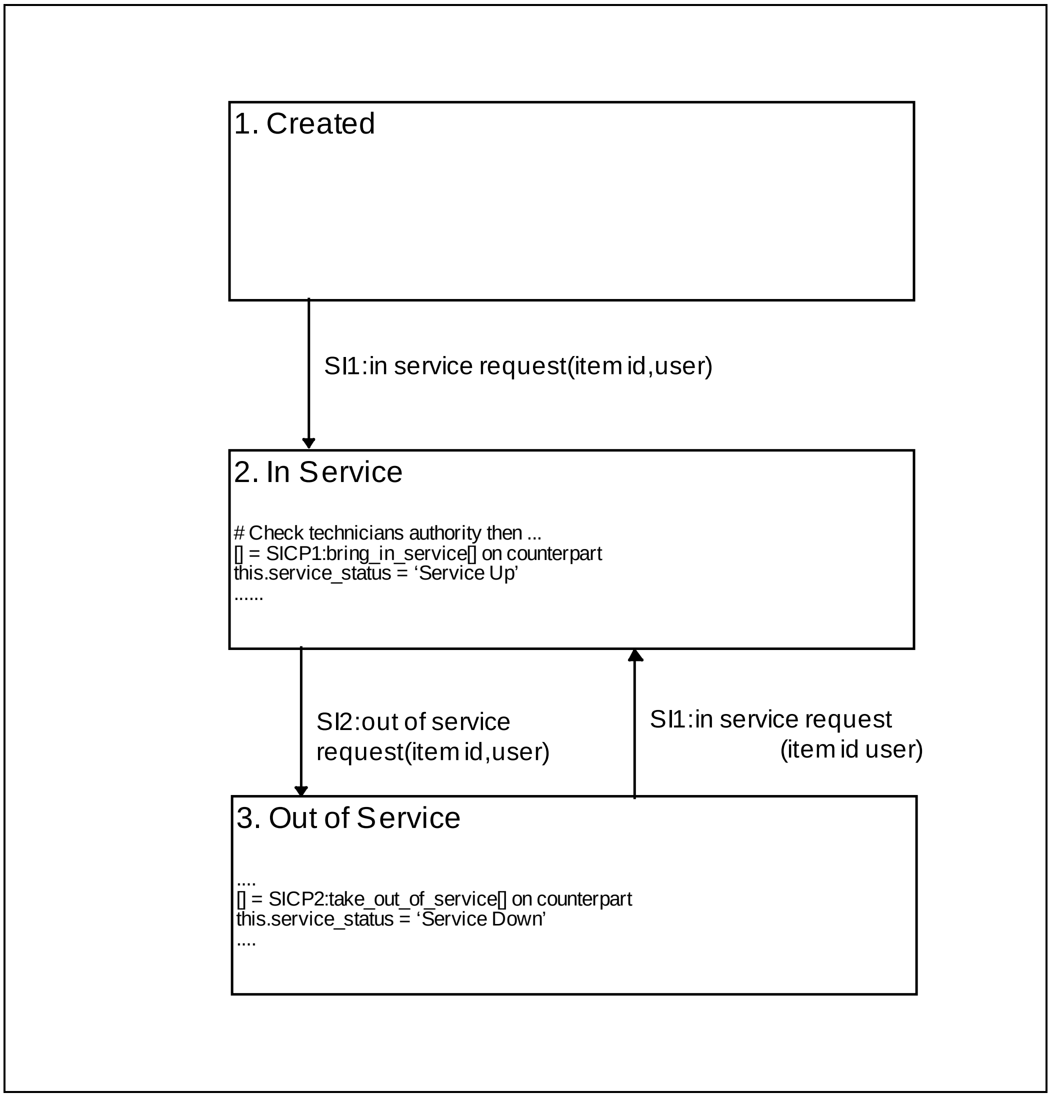
This diagram captures a form of Supertype/Subtype relationship between objects in different domains. Just as with such relationships within a single domain we can assert that it is mandatory in both directions. Thus, for example, every instance of “Robot” must have a corresponding instance of “Serviceable Item”. Similarly, every instance of “Serviceable Item” must have a corresponding instance of either “Robot” or “Display” etc. Just like intra-domain relationships, this inter-domain relationship must be identified. For this we will use the syntax:

CPR<relationship number>

and this example we will call “CPR1”. Such inter-domain relationships will be defined in the context of a project build, and the format of the definition is covered in section 8.6.3.6.

The nature of this relationship implies that whenever an instance of a specialised object is created, then a corresponding instance of “Serviceable Item” must be created. This must be similarly true for deletion. This is issue explored later in this section.

The true utility of this concept becomes apparent if we consider a possible lifecycle for “Serviceable Item”:


1. Created
SI1:in service request(item id,user)
2. In Service
SI2:out of service request(item id,user)
SI1:in service request (item id user)
3. Out of Service

```
# Check technicians authority then ...
[] = SICP1:bring_in_service[] on counterpart
this.service_status = ‘Service Up’
......
```

```
....
[] = SICP2:take_out_of_service[] on counterpart
this.service_status = ‘Service Down’
....
```

Figure 20 State Model for “Serviceable Item”

The state “In Service”, for example, deals with maintenance aspects of the problem (checking authority, recording information etc.) and also causes the item itself (the counterpart) to be brought into service via the call to “SICP1:bring_in_service[ ]”. However, this domain does not “know” how to switch on a Node or activate a Robot. In this domain “SICP1:bring_in_service” has a definition of its interface but not of any implementation. We require, therefore, that every counterpart object of “Serviceable Item” provides an implementation of “SICP1:bring_in_service”.

At run time, when the “Serviceable Item” enters the “In Service” state, the appropriate implementation of “SICP1:bring_in_service” for the related counterpart instance, of whatever type, will be invoked. Thus, if we decide that other objects are to become “Serviceable Items”, we need only declare the counterpart mapping and provide the implementations of the necessary operations.

Note that the operation “bring in service” is associated with a terminator “Serviceable Item Counterpart”, with key letter “SICP”. We return to this in section 8.6.3.7.

In the next few sections we will cover these ideas in greater detail.

8.6.3.1 Creation and Deletion of Counterpart Instances

Since the nature of this counterpart instance relationship is similar to a super/subtype relationship, we will require that instances at either end of the relationship always have counterparts at the other. This implies that whenever an instance is created, the counterpart must also be created. Similarly deletion must also be propagated.

Creation or deletion can be stimulated from either end of the relationship. In order to achieve this, the analyst must define and publish creation routines for those objects that are automatically created, and the architecture must arrange for the creation routine to be called. We term these *counterpart creation services*.

Counterpart creation services must:

- Cause the creation of the counterpart instance

- Return the instance handle of the created instance to the architecture

These requirements imply that the instance must be created synchronously, although the creation service may also send an event to the newly created instance to progress it through the lifecycle.

Here is an example, where creation is stimulated from the specialised end of the relationship.

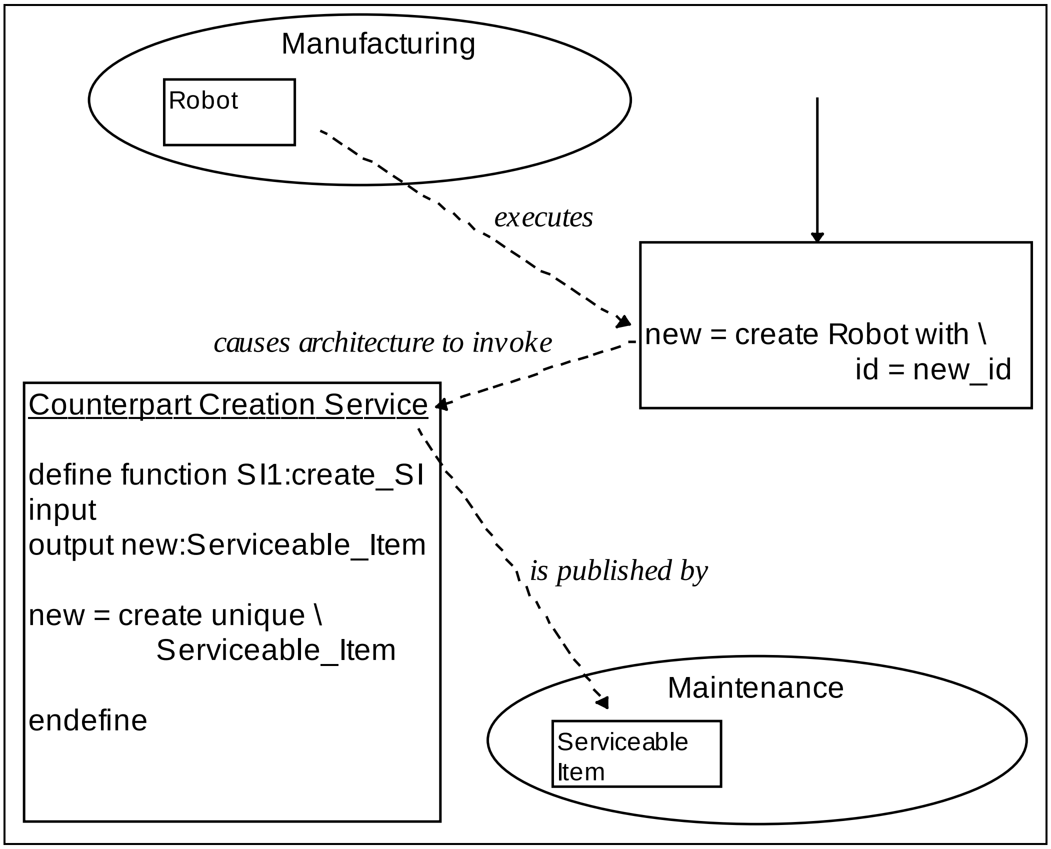

Manufacturing
Robot
executes
new = create Robot with \
id = new_id
causes architecture to invoke
Counterpart Creation Service
define function SI1:create_SI
input
output new:Serviceable_Item
new = create unique \
Serviceable_Item
endefine
is published by
Maintenance
Serviceable
Item

Figure 21 Counterpart Creation from Specialised End

Here the analyst has specified that:

- Robot is a counterpart of Serviceable Item (CPR1)
- SI1:create_SI is the counterpart creation service for Serviceable Item

The architecture will execute code *as if* the following ASL had been executed in the creation state of “Robot”:

```
....
new = create Robot with id = new_id
[] = create_CPR1_counterpart[new]
....
```

and *as if* the following bridge ASL had been executed:

```
define bridge MANUFACTURING:create_CPR1_counterpart
input new_robot:Robot
output

$USE MAINTENANCE
[new_si] = SI1:create_SI[]
link_counterpart new_robot CPR1 new_si
$ENDUSE
```

where “link_counterpart’ is an ASL operation that allows linking of instances in different domains. Note that:

- The analyst is not required to write this bridge ASL or modify the original state action. The architecture translation must take care of this automatically.
- The service “create_CPR1_counterpart” is an implicit service created by the architecture. For this reason we have not associated it with any explicit terminator, nor given it a key letter designation.

The counterpart creation service (SI1:create_SI):

- Must return a handle to the instance of Serviceable Item that it creates
- Cannot return any other data
- Cannot accept any other inputs unless an explicit mapping is provided by the analyst.

We do not intend to explore this latter point further here other than to state that the mapping would have to be made between (in this case) attributes of Robot and input parameters to SI1:create_SI.

If creation is stimulated from the generalised end of the counterpart then the analyst must provide additional information that tells the architecture *what type* of counterpart is to be created. In this case, given that a “Serviceable Item” is being created, is the counterpart to be a “Robot” or a “Display” etc. ?

We expect that in this case the object at the generic end of the relationship will carry an attribute that indicates which type of counterpart it has.

Modifying the previous example:

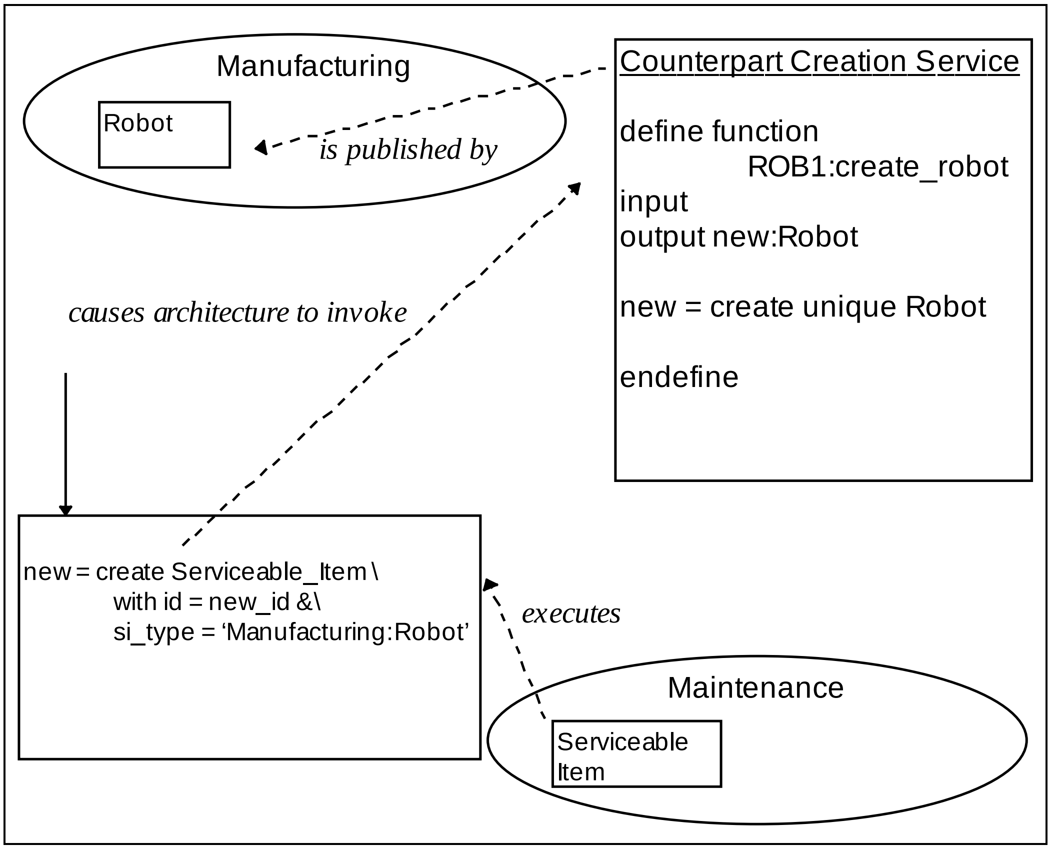


Figure 22 Counterpart Creation from Generic End

Here the analyst has specified that:

- Robot is a counterpart of Serviceable Item (CPR1)
- ROB1:create_robot is the counterpart creation service for Robot
- si_type indicates the type of counterpart object

The architecture will execute code *as if* the following ASL had been executed in the creation state of “Serviceable Item”:

```
....
new = create Serviceable_Item with id = new_id &\
                   si_type = ‘Manufacturing:Robot’
[] = CL1:create_counterpart[new,si_type]
....
```

Where an additional service call has been inserted after the analyst written “create” operation on the “Serviceable Item”. CL1:create_counterpart is a bridge operation defined as:

```
define bridge MAINTENANCE:CL1_create_counterpart
input new_si:Serviceable_Item,si_type:SI_TYPE
output

switch si_type
case 'Manufacturing:Robot'
     $USE MANUFACTURING
          [new_robot] = ROB1:create_robot[]
          link_counterpart new_robot CPR1 new_si
     $ENDUSE
case 'Manufacturing:Assembly_Line'
     ....
     ....
endswitch
enddefine
```

Note that:

- As before, the analyst is not required to write this bridge ASL or modify the original state action. The architecture translation must take care of this automatically.

- Unlike the specific end of the counterpart relationship, the generalised end "knows" about the existence of the counterpart. For this reason the service "CL1:create_CPR1_counterpart" has been explicitly associated with a "Client" terminator in this domain.

- The type of the attribute "si_type" must be a deferred type[22], implemented by the software architecture. As shown in this example, this type must capture both the object and the domain in which is resides. Normally such internal "structure" of deferred types would not be visible in this domain. However, in order to illustrate the concept we have used an enumeration constant in the state action code, and this has made the "structure" visible. In practice generic domains such as "Maintenance" would never have hard codes attribute values such as this.

It can be imagined that in a given system, creation may be stimulated from either end of the relationship depending on the circumstances. In such a case, the architecture must ensure that a creation "loop" does not occur.

Deletion of counterparts would be handled in a similar manner, with (in the above example) the analyst specifying Counterpart Deletion Services for the Robot or the

[22] See [Wilkie 96-3] for a discussion of the ideas of deferred types.

Serviceable Item depending on which end stimulates the deletion. As with creation, the architecture must ensure that a “loop” does not occur.

8.6.3.2 Propagation of Behaviour Between Counterparts

Having set up the correspondence between instances, we will wish to propagate behaviour from one end to the other. The nature of the super/subtype abstraction between domains is such that we expect that:

- The generic object is “aware” of the existence of the specific objects although it does not “know” exactly how those objects behave. In the example given at the start of this section we showed the generic object (Serviceable Item) invoking a service that is implemented by the specific objects (Robot etc.). We expect to see the Maintenance domain invoke such services explicitly since the domain has no function without real items to maintain. However, the Maintenance domain does not know what the Robot, for example, does to take itself “out of service”. Rather the Maintenance domain expects that the Manufacturing domain interprets the meaning of “out of service” in the context of a Robot. This implies a clear contract definition for the “out of service” call.

- By contrast, the objects at the specific end of the relationship will not be “aware” of the generalised concept that is connected by a counterpart relationship. The behaviour of “Robot” within the manufacturing domain will be driven by the requirements of that domain and not by the fact that it is also a “Serviceable Item”. We expect then that the Robot will not invoke explicit operations on its counterpart. Instead, operations will be invoked on the counterpart when something *happens* to the specific object instance, in other words when an event is received by it.

Taking the example of the *explicit counterpart service invocation* first:

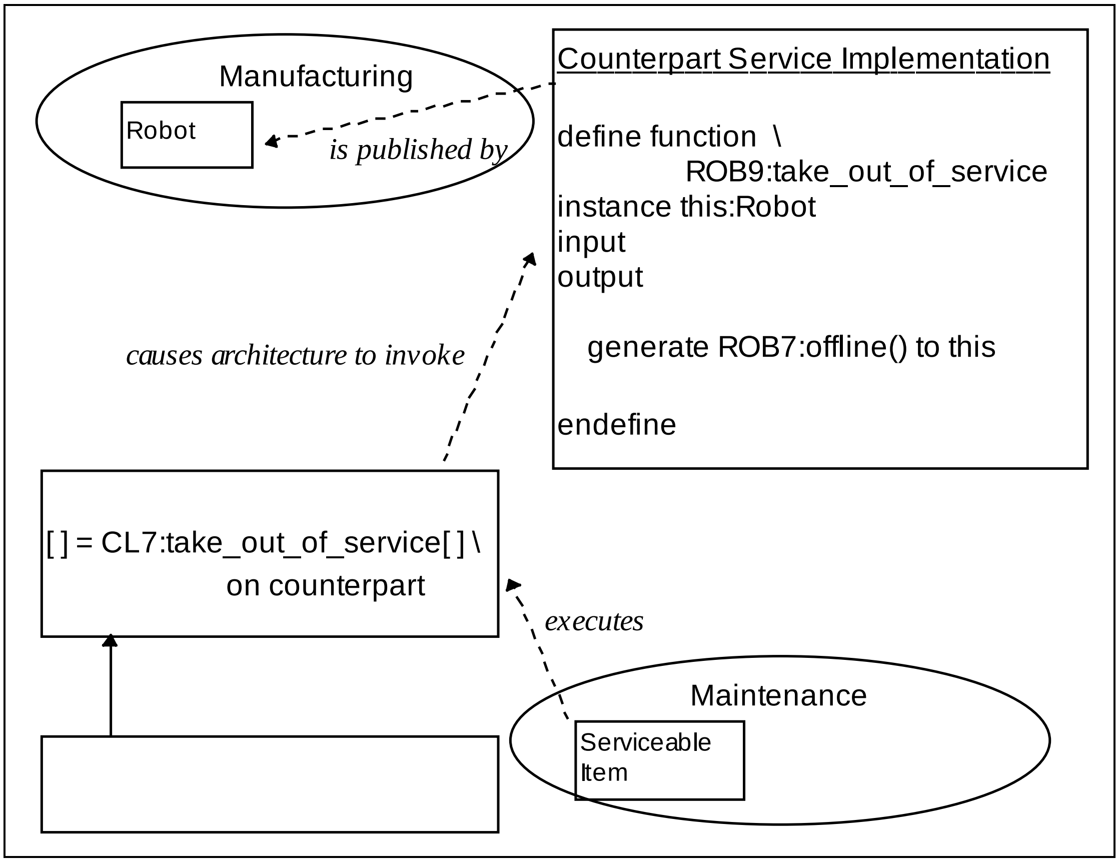


Figure 23 Service Invocation by Generic End

Here the analyst has specified that:

- Robot is a counterpart of Serviceable Item (CPR1)

- CL7:take_out_of_service is a counterpart operation on the Client terminator in the Maintenance Domain.

- ROB9: take_out_of_service is the implementation of CL7 for a Robot in the Manufacturing domain. This is an “instance based” service in the manner described in section 3.

The architecture will execute code *as if* the following ASL had been executed for the bridge mapping:

```
define instance bridge \
               MAINTENANCE:CL7_take_out_of_service
instance this: Serviceable_Item
input
output

cp = this -> CPR1
cp_type = type-of(cp)
switch cp_type
case 'Manufacturing:Robot'
     $USE MANUFACTURING
          []=ROB9:take_out_of_service[] on cp
     $ENDUSE
case 'User_Interface:Display'
     $USE USER_INTERFACE
          ....
     $ENDUSE
case 'Manufacturing:Assembly_Line'
     ....
     ....
endswitch
```

Again, it should be noted that the analyst is *not required* to write such ASL. This ASL is shown here to illustrate what will happen when the call is made. Note that the ASL in this bridge has some special features:

- The construct "-> CPR1" refers to the navigation of the counterpart instance relationship.

- This is an "instance based" bridge in an analogous idea to the instance based synchronous service. In the bridge however, an instance in one domain is mapped to the corresponding instance in the counterpart domain. As shown in the state model, this bridge is invoked from a state of "Serviceable Item", with the call "[] = CL7:take_out_of_service[] on counterpart". The "on counterpart" clause in effect supplies "this" to the bridge.

- The type of "cp" returned by the navigation is "dynamic" and can therefore at run time be any of the types of instance handle for objects that have been declared as counterparts of "Serviceable Item". The operation "type-of" is described in section 7.2.

By contrast, if we wish behaviour to be propagated from the specific end to the generic end, we believe that the analyst should map the occurrence of an event in the specific domain to the invocation of a service in the generic domain. This is in line with the idea that the description of the behaviour of the specific domain is unaffected by the requirements of the generic domain.

Such a mapping is shown in the following example:

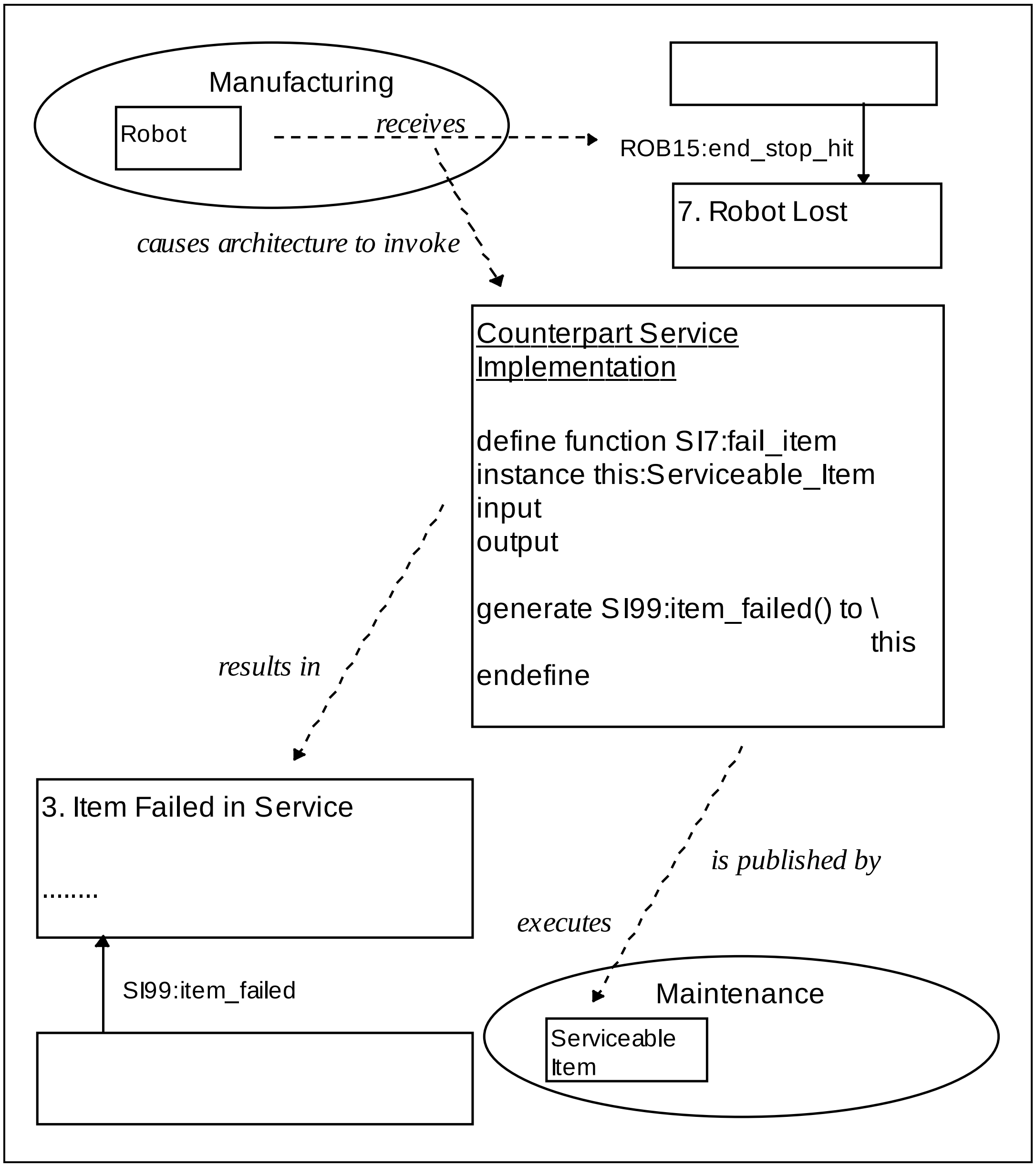

Manufacturing
Robot
receives
ROB15:end_stop_hit
7. Robot Lost
causes architecture to invoke
Counterpart Service Implementation
define function SI7:fail_item
instance this:Serviceable_Item
input
output
generate SI99:item_failed() to \
this
endefine
results in
3. Item Failed in Service
....
is published by
executes
SI99:item_failed
Maintenance
Serviceable Item

Figure 24 Implicit Service Invocation Stimulated from Specialised End

Here, the analyst has specified that:

- Robot is a counterpart of Serviceable_Item (through CPR1)
- When event “ROB15:end_stop_hit” is received by the Robot, then the service “SI7:fail_item” is to be invoked on the counterpart

- In this case, the definition of the counterpart service implementation sends an event which causes the “Serviceable Item” to enter the “Item Failed in Service” state

The architecture will execute code *as if* the following ASL had been executed wherever the event ROB15:end_stop_hit is received by an instance of Robot:

```
 [] = fail_robot[] on counterpart
```

where the “counterpart” is the counterpart of the instance receiving the event. Note that this would be executed even if the Robot state model actually ignored the event. The service “fail_robot” would be equivalent to the following bridge:

```
define instance bridge MANUFACTURING:fail_robot
instance this: Robot
input
output

cp = this -> CPR1
$USE MAINTENANCE
[] = SI7:fail_item[] on cp
$ENDUSE
enddefine
```

Note that:

- The analyst is not required to write this bridge. It is implicit in the support provided by the architecture. The service “fail robot” is implicit and thus we have not associated it any explicit terminator in the Manufacturing domain.

- We have not defined exactly how the analyst specifies that ROB15 is mapped to SI7. Such information will be captured by a CASE tool.

The example shown here was that of an event transmission in the specific domain being mapped to a service invocation in the generic domain. Similarly, the analyst would be able to declare that an invocation of a synchronous operation in the specific domain would result in the invocation of an operation in the generic domain.

8.6.3.3 Parameter Mapping and Data Types

In the examples that we have shown so far, there have been no parameters passed to any of the operations that the analyst has access to.

If a counterpart service requires parameters, then these must appear in the formal definition in the invoking domain and appear in every implementation of the service for every counterpart object.

Suppose that the Serviceable_Item object in the Maintenance domain invokes a service on its counterpart:

```
[a] = CL15:do_something[b,c] on counterpart
```

with the formal definition for the service being:

```
instance this: Serviceable_Item
input b:my_integer,c:my_real
output a:my_real
```

where the types my_integer and my_real are User Defined Types based on Integer and Real respectively. The implementation of the service for the Robot object in the "MANUFACTURING" domain might be:

```
define function ROB21:do_something
instance this: Robot
input b:my_integer,c:my_real
output a:my_real
     # ASL using a, b, and c
enddefine
```

Then, following the earlier explanations of counterpart bridge mappings, the type casting rules are those *equivalent* to the execution of the following construct:

```
define bridge MAINTENANCE:CL15_do_something
instance this: Serviceable_Item
input b:my_integer,c:my_real
output a:my_real

cp = this -> CPR1
$USE MANUFACTURING
     [a] = ROB21:do_something[b,c]
$ENDUSE
enddefine
```

As described in [Wilkie 96-6], the input parameters to the bridge revert to base type within the $USE clause and are subject to the rules for such types. Similarly the output parameters are returned as base types from the call and become the corresponding UDTs when outside the $USE clause. Any constraint violation in this process should cause an exception to be raised. Note that:

- To avoid the analyst having to specify mappings explicitly, the parameters are mapped between CL15 and ROB21 on the basis of position. Thus the analyst need only specify that CL15 and ROB21 are mapped counterpart operations.

- The base types of the parameters to CL15 and to ROB21 must be commensurate in the manner described in [Wilkie 96-6].

8.6.3.4 Counterpart Attribute Mappings

In addition to mappings of operations, analysts may wish to map attributes, such that value in the two domains are kept synchronised. Suppose, for example, that we wish the “Serviceable Item” to have an “Actual Status” attribute. The value of this will reflect the detailed status of the actual counterpart of the item. This attribute might be used for display or logging purposes. The values that this status attribute can take on will depend on the type of the counterpart since a “Robot” will have a different lifecycle from an “Assembly Line”. It will be up to the analyst of each domain to indicate the mapping between an attribute in the specific object and in the generic object.

The analyst must therefore declare:

- The attributes to be mapped
- The mapping/transformation between them

Having done this, the architecture is free to implement the mapping in the most efficient way. A number of choices are possible:

- Both attributes share the same memory location. Accessors in either domain utilise a transformation based on that provided by the analyst.
- The attribute is stored in one domain only. Accessors in the other domain are replaced by a counterpart navigation and attribute access.
- Attributes are stored in both domains. Write accessors are augmented by a counterpart navigate and update.

The choice will depend on the relative update and read rates in each domain, memory requirements and the way in which counterparts are implemented.

8.6.3.5 Meta-level Operation Mappings

A common generalised domain is one that deals with user interfaces. Such a domain might contain the following model fragment:

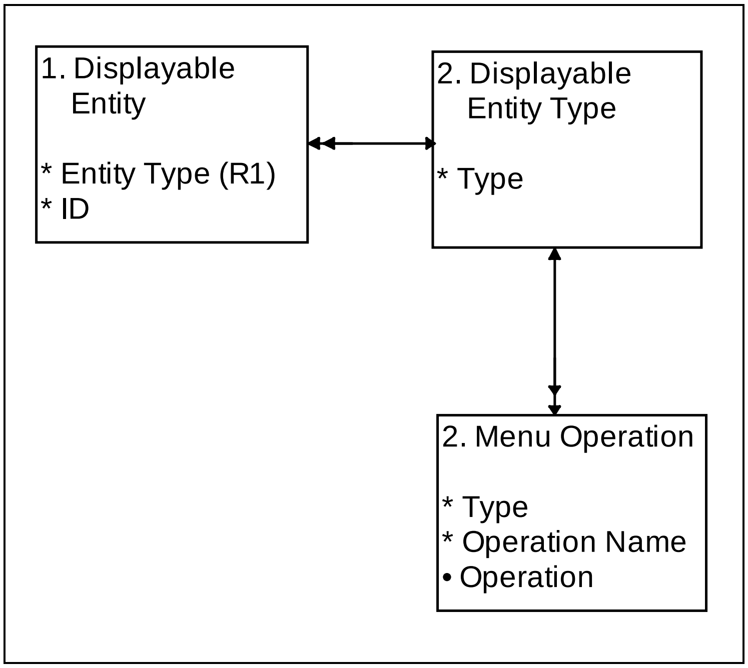


Figure 25 Model Fragment from Generic UI Domain

This information model captures rules about menu contents in relation to the type of item being displayed. The state models will specify the behaviour that causes those menu operations to be invoked. At a certain point, we will wish the "Displayable Entity" to be able to invoke the actual operation specified by the "Operation" attribute of "Menu Operation" directly on its counterpart. That is "Displayable Entity" will invoke the operation:

```
[] = CL1:execute_operation[chosen_op] on counterpart
```

Where CL1 is an operation associated with the "Client" terminator in this UI domain. It would be highly desirable to avoid the analyst having to write a bridge mapping of the following form:

```
switch selected_operation
case 'Schedule Train'
[] = TR15:schedule[]
case 'Delete Train'
[] = TR17:delete_train[]
....
```

and so on. Instead, the type of the “Operation” attribute should be “OOA Operation” (a meta-type), and the bridge specification in (ASL) should say:

```
$INVOKE selected_operation
```

This will require an enhancement to ASL and may also have to deal with parameter passing. We intend to return to this issue in the future.

8.6.3.6 Specific/Generic Bridge Contracts

Contracts must be specified for the various services involved in this type of counterpart mapping. We will require a contract describing the requirements of the generic object. In the example we have been using so far we will require a contract for “Serviceable Item/CPR 1”:

Counterpart Relationship: CPR1 Serviceable Item

Relationship Type: Generic/Specific

Description: This captures the maintenance behaviour common to all counterparts, for example in taking an item in or out of service.

Required Services: CL7:take_out_of_service

CL8:put_in_service

Provided Services: SI7:fail_item

This provides a summary of the obligations of any client that wishes to be a counterpart of a “Serviceable Item”. In addition, both the required and the provided services will require contracts. These provide a detailed picture of the obligations of the client. For example:

Service Name: CL7:take_out_of_service

Input Parameters: <none>

Type: Closed, Blocking

Description: This will cause the counterpart to go out of service. The precise interpretation of “out of service” will depend on the context. An item that is out of service is not in a state where it can perform its normal functions. In this state there will be no unsolicited events occurring that are propagated to the Serviceable Item. The counterpart will remain in this state until explicitly put in service.

Output Parameters: <none>

Note that this description refers explicitly to the state behaviour of the client. Implicit in this is that to fully understand the obligations of a client, the analyst must have visibility of the State Model of the generic counterpart.

8.6.3.7 Representation on the OCM

In the examples that we have shown in previous sections we have seen counterpart operations in the Generalised domain explicitly associated with a terminator (frequently the “Client” terminator). We would therefore expect to see such communication represented explicitly on the OCM. For example:

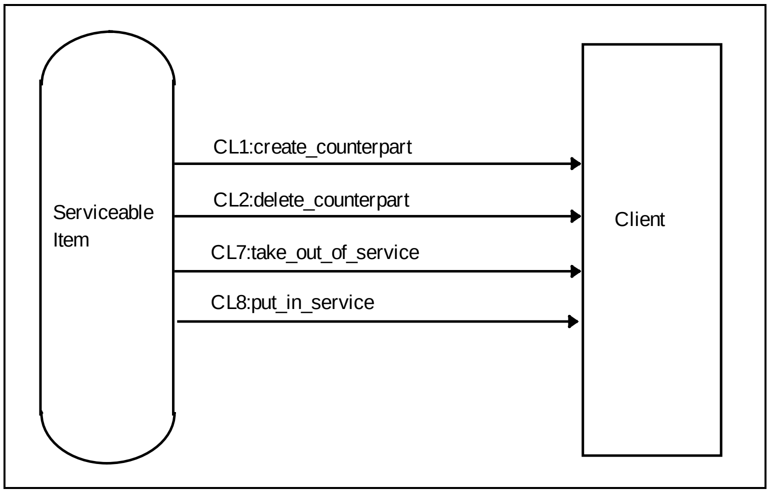


Figure 26 Counterpart Operations on the OCM

However, in the case of the specialised end we will not see such communication on OCM, since one of the benefits of this type of mapping is to remove the complexity dealt with in the generalised domain from the various specialised domains.

8.6.4 Peer-to-Peer Counterpart Mappings

In the generic/specific mappings we dealt with the situation where:

- the specialisations are not “aware” of their connection to generalised behaviour
- much behaviour is propagated implicitly between the counterparts
- several different specialisations may be involved in the relationship
- at the instance level, one-to-one mandatory mappings are involved

However, we will sometimes require to maintain relationships between counterparts where the above conditions do not apply. We term these “peer-to-peer” mappings and are analogous to ordinary relationships in OOA.

For such relationships the analyst will be required to manage the instance and operation mappings explicitly. Suppose we have a domain that communicates with a terminator called “Remote System”. We might wish to set-up and maintain a counterpart relationship (CPR2) with the following contract:

Counterpart Relationship:	CPR2 Remote System
Relationship Type:	Peer-to-Peer
Description:	This captures the connection between functioning of items in this domain and that of a remote system.
Required Services:	RS1:online
	RS2:offline
	RS3:get_status
	RS4:connect_to_remote_system
Provided Services:	XT5:remote_system_ready

This provides a summary of the obligations of the domain at the other end of the relationship. Each service will require a contract, for example:

Service Name:	RS4:connect_to_remote_system		
Input Parameters:	required_system_type	system_type	Provides a definition of the characteristics of the remote system required.
Type:	Closed, Blocking		

Description: This will cause the connection of the calling entity in this domain to a corresponding entity in the remote system. If a suitable system does not exist, then one will be created for the purpose.

Output Parameters: <none>

As part of a system build, the analyst will have to specify mappings for all the services, for example, the mapping for RS4:create_remote_system might be:

```
$USE RSD
[suitable_remote_system] = \

RSD1::find_or_create_system[required_system_type]
link_counterpart this CPR2 suitable_remote_system
$ENDUSE
```

Note that:

- the instance handle “this” is valid here, since the operation RS4:create_remote_system is defined as a counterpart operation.

- the fact that the handle “this” is valid implies that the terminator itself is, in effect, an “instance based terminator”. That is: the analyst must define which object in the domain containing the terminator is going to use the services of the terminator and thus participate in the counterpart relationship.

- that the analyst will explicitly use the extended ASL operations such as “link_counterpart” “unlink_counterpart” and counterpart navigation that were used in earlier sections to illustrate the specific/generic mappings.

8.7 Implementation Using ASL

Throughout this section we have used an ASL 2.5 to provide examples and demonstrate the bridge mappings. This extended ASL incorporates ASL 2.4 [Wilkie 96-1] plus the following enhancements:

- Formalised Synchronous Operations as described in section 3
- Dynamic Data Types as described in section 7
- Deferred Data Types as described in section 6
- The “on counterpart” clause in service invocation
- Explicit counterpart relationship manipulation in bridges (link_counterpart, unlink_counterpart and ->)

The use ASL 2.5 has appeared in two roles in this section:

- In the analyst written ASL (operations invoked “on counterpart” and explicit peer-to-peer manipulations in bridges)
- In *implicit* ASL associated with generic/specific counterpart relationships (in the text, we have used the term “at run time the system will behave *as if* the following ASL had been executed”)

This latter role serves two purposes:

- Use of ASL provides a clear and unambiguous definition of the behaviour of such complex and implicit mappings.
- By actually generating the implicit ASL shown, such counterpart operations can be supported in architectures that support only the enhanced ASL outlined at the start of this section.

Thus, if a CASE tool or a utility developed for a particular project can generate the required ASL from the counterpart relationship information in the model, then the requirement load on each and every back end architecture is reduced.

Of course, support using the generated ASL might have an impact on performance in which case *some* architectures would need to ignore the generated ASL and provide more direct support for the counterpart constructs. This could be achieved, for example, using inheritance in C++.

The approach of using “implicit” ASL to support aspects of the method is, we feel of general utility[23]. It allows a pragmatic compromise of convenience and performance.

[23] Another example of this is with initial instance populations in a non-persistent architectures. Conveniently, such data can be expressed as tables. Any architecture can choose to load such tables (in whatever format) quickly and efficiently. Alternatively, the tables can be translated into ASL which specifies the data (using “create” and “link” constructs). Such ASL can be executed by *any* architecture that supports ASL without the need for “special” architectural features for loading such data. This saving is, of course, achieved at the expense of binding code into the application that is executed once only at start up.

9. References

[Shlaer 88] Sally Shlaer and Stephen J. Mellor
Object Oriented Analysis: Modelling the World in Data
Prentice Hall, 1988

[Shlaer 92] Sally Shlaer and Stephen J. Mellor
Object Oriented Analysis: Modelling the World in States
Prentice Hall, 1992

[Shlaer 94] Sally Shlaer
Building Bridges: A Preliminary Report
Project Technology, 1994

[Shlaer 96] Sally Shlaer and Steve Mellor
Synchronous Services
Project Technology, 1996

[Raistrick 94] Christopher Raistrick
A Practitioners Guide to the Use of Bridges in Shlaer-Mellor Recursive Development
Kennedy Carter, 1994

[Wilkie 94] Ian Wilkie
Synchronous Services
Kennedy Carter, 1994 (KC/OOA/TN/44)

[Wilkie 96-1] Ian Wilkie, Adrian King, Mike Clarke and Chas Weaver
The Action Specification Language (ASL) Reference Manual
Kennedy Carter, 1996 (KC/OOA/CTN/06 V2.4A)

[Wilkie 96-2] Ian Wilkie
Formalisation of Synchronous Services
Kennedy Carter, 1996 (KC/OOA/CTN/44)

[Wilkie 96-3] Ian Wilkie
A Proposed Policy for the Implementation of Deferred Data Types using non-OOA Domains in OOA/RD
Kennedy Carter, 1996 (KC/OOA/CTN/43)

[Wilkie 96-4] Ian Wilkie

A Proposed Policy for the Handling of Errors and Exceptions in OOA/RD

Kennedy Carter, 1996 (KC/OOA/CTN/42)

[Wilkie 96-5] Ian Wilkie
A Proposal for Modified Event Processing Rules in OOA/RD
Kennedy Carter, 1996 (KC/OOA/CTN/46)

[Wilkie 96-6] Ian Wilkie, Adrian King, Mike Clarke and Chas Weaver
The Action Specification Language (ASL) Reference Manual
Kennedy Carter, 1996 (KC/OOA/CTN/06 V2.5)

[Carter 96] Colin Carter and Ian Wilkie
A Proposed Policy for the Implementation of Dynamic Data Types in ASL
Kennedy Carter, 1996 (KC/OOA/CTN/45)