Understanding the Behaviour of Real-Time and Distributed Systems Constructed from Object-Oriented Frameworks

R. J. A. Buhr
Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, Ontario, CANADA, K1S 5B6
email: buhr@sce.carleton.ca
web: http://www.sce.carleton.ca/faculty/buhr.html
tel: (613) 520-5718
fax: (613) 520-5727

Alex Hubbard
Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, Ontario, CANADA, K1S 5B6
email: alex@sce.carleton.ca
web: http://www.sce.carleton.ca/rads/alex/alex.html
tel: (613) 520-2600 x5728
fax: (613) 520-5727

April 22, 1996

Abstract

An emerging approach to developing software for real-time and distributed systems is to build them with standard components and patterns embodied in frameworks such as Schmidt's public-domain C++ framework, ACE. A framework such as ACE enables developers to reuse good solutions to implementation problems discovered through the experience of others (the problem-solution pairs are called patterns). This is effective for simplifying and speeding up the development process but has the side effect of hiding the overall structure and behaviour of particular systems implemented with the framework behind a complex screen of disconnected, general pattern concepts and of customization details widely scattered in code files. Pattern experts and developers will form their own understanding of overall system structure and behaviour based on experience, but this leaves the understanding locked in their minds. There is a need for a way of diagramming the overall structure and behaviour of systems constructed from frameworks in a high-level way to allow all to share this understanding. This paper describes a diagramming technique called use case maps that satisfies this need. The technique is not restricted to systems constructed from frameworks but this paper focuses on such systems through

an ACE case study.

1.0 Introduction

The approach of using standard components and patterns from frameworks like Schmidt's Adaptive Communication Environment (ACE) [1] provides an effective way of constructing new real-time-and-distributed systems with good properties quickly. However, the effect on system understanding is a bit like trying to understand from an automobile parts catalog how a particular automobile constructed from the parts will look and drive: complex relationships obtained from scattered details must be held in the mind's eye in a high-level way. The problem is actually worse than for automobiles for a number of reasons: we do not have any physical models of software systems or parts to keep in the mind's eye, so the system view we want will not be a concrete thing like an automobile but an abstraction that may seem quite distant from code; the patterns embodied in frameworks that supply the system parts are themselves described abstractly, so getting a system view requires distancing ourselves two levels of abstraction away from code; and a software system is, while running, like an automobile that can "morph" into different forms while you are driving it (because its structural form may be fluid due to the creation, destruction, and changing interrelationships of operational parts such as processes and objects).

We need a conceptual model with the following properties for understanding software systems: the model can easily be held in the mind's eye and documented for use by others; it can represent both large-scale behaviour patterns and changing structural forms in one package; it is a first-class model that is not dependent on details of pattern customization or code; and it provides a framework for combining patterns. This paper introduces and illustrates by example a model with these properties called *use case maps* [2]. The model combines high-level (or large-scale) structure and behaviour in a coordinated way to give a characterization of a software system as a whole. This characterization may be called *architecture* in the sense that term is commonly used to mean a description of a system (or family of them) at a large scale (in this sense, design patterns of the kind embodied in ACE are not "architecture", although the term is sometimes applied to them in the object-oriented community).

Because the paper applies use case maps to an existing framework (ACE) that was developed without them, it necessarily presents the approach from a reverse engineering perspective. However, we hope that the reader will be able to perceive that there is no reason to limit the application of use case maps to reverse engineering. Although not illustrated by this paper, they can equally well be used as architectural patterns to guide forward engineering, in which case they would become part of documentation.

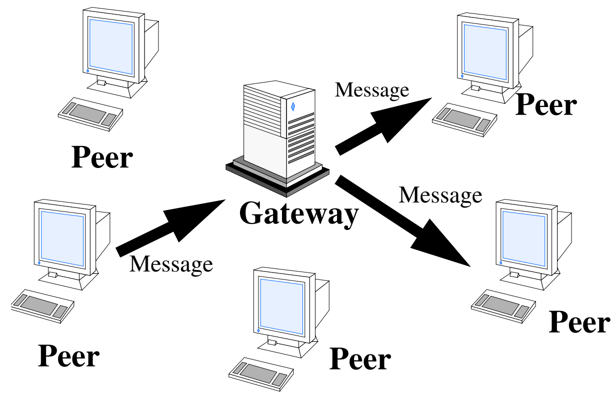


Figure 1: The Gateway System

2.0 ACE and the Gateway Application

ACE is a framework for constructing distributed applications such as the Gateway application indicated by Figure 1. The code for this application is provided as an example with ACE. This application is a small distributed system in which peer workstations interact through a gateway computer. From the point of view of the gateway, some of the peers act as input peers which generate messages and others act as output peers which consume messages. Messages are sent from any one input peer to the gateway which then routes the messages to one or more output peers. The diagram and the above prose describe only how the hardware of the gateway system works. To understand how the software works, one must first understand the ACE patterns that are used to implement it.

ACE patterns are described in a number of papers [3][4][5][6][7]. Each pattern is described in an abstract way in isolation from other patterns with a combination of prose and standard object-oriented diagramming techniques. Their embodiment in C++ code in ACE is nontrivial to understand, because C++ features are used to the full and much detail in different places in the code must be understood. How these and other patterns should be chosen or combined to create an application is left up to the ACE user. The patterns are more general than any particular ACE application, so any particular specific application is visible only in the customization details. Therefore, piecing together a simple, overall picture of the gateway software like that of Figure 1 for the hardware requires chasing and mentally integrating a lot of detail in both abstract descriptions and code.

After doing this one discovers that the patterns used in the gateway are as follows:

- A service configurator pattern supports creation and management of service objects, such as the ones required in the gateway computer to provide services to peers.
- Acceptor and connector patterns support establishment of communication channels, such as UNIX sockets. The acceptor pattern supports listening for connect requests on a particular network address. The connector pattern supports the making of connect requests to particular network addresses.
- A reactor pattern supports event demultiplexing. It supports registration of event handlers so that when an event occurs the correct event handler is notified. Events include communications events, timers, and UNIX signals.

After some period of immersion in ACE details, one can form a mental picture of how the patterns fit together to implement the system of Figure 1. The gateway uses the connector pattern to connect to the peers. The peers use the acceptor pattern to receive connections from the gateway. The service configurator pattern is used in both the gateway and peers to provide a generic initialization mechanism. The reactor pattern is used extensively in both gateway and peers as the underlying communication mechanism driving them.

However, this mental picture, involving as it does intimate knowledge of ACE patterns and code, is neither easy to explain to others nor easy to remember after some time away from the details. This is where use case maps enter the picture.

3.0 Simplest Use Case Map of the Gateway

Figure 2(a) introduces use case maps by showing the simplest possible use case map of the Gateway software at roughly at the same level of operational understanding as the hardware view of Figure 1. It represents normal, failure-free operation of two peers and a gateway, after initialization has been completed. A sense of the effort required to discover the simple use case map of Figure 2(a) is given by Figure 2(b). This section is concerned with explaining Figure 2(a) and Figure 2(b), beginning with Figure 2(a).

In general, a use case is set of related scenarios of system operation. There is no implication that the scenarios are between human users and the system as a whole, although they may be. In general, a use case includes the possibility of scenarios involving many systems or subsystems that are users of each other. Use case maps are a diagramming technique for superimposing paths of such scenarios on diagrams of system structure to help understanding.

The structural part of Figure 2(a) is shown by a set of boxes with different shapes and positional relationships to each other. Here, the boxes with sharp corners represent UNIX processes that provide the operational context for ACE objects. ACE objects are represented by boxes with rounded corners inside the process boxes. The objects gh1, gh2, ic, oc are all service handlers that come from the acceptor and connector patterns, which provide for creation of such objects to handle steady-state operation. The routing table object is not part of any pattern in ACE, but is implemented as an isolated application class.

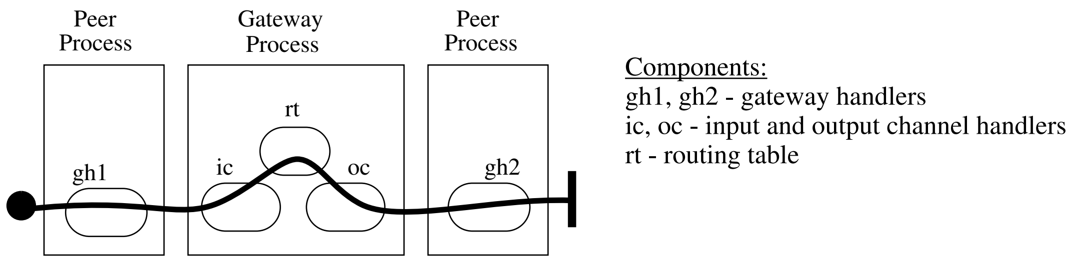
The behavioural part of Figure 2(a) is shown by a path superimposed on the set of boxes (in general there may be many such paths). In general, paths in use case maps trace cause-effect sequences that traverse a system from points of stimuli to points of responses. The paths are at a level of abstraction above the interactions between components that will be required to make the causal sequences happen. In general, there may be many interobject or interprocess interactions required to implement causal path segments between components, so a path provides an abstract view.

In this particular map, superimposing a path on a component indicates that the component has a *responsibility* along the path (the commonly used term "activity" means, in this context, "performing a responsibility").

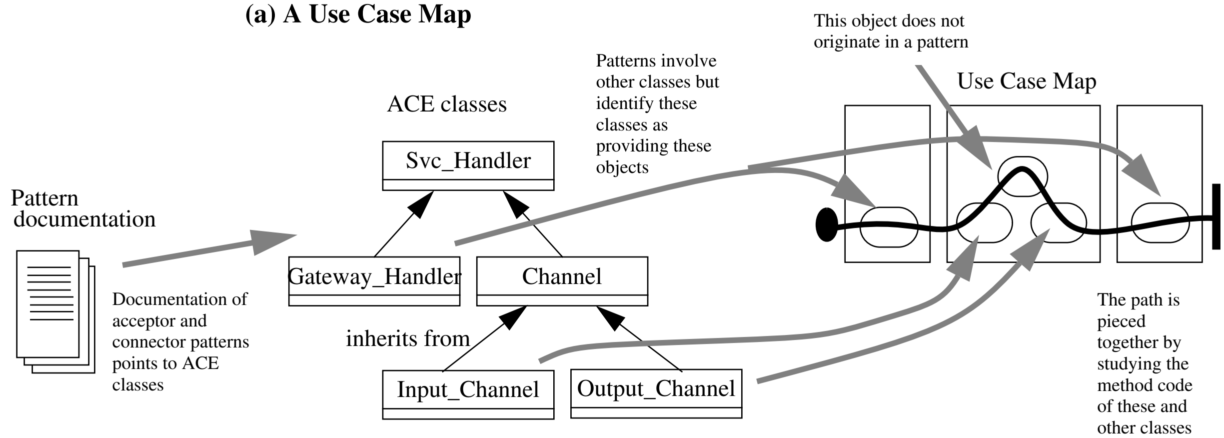
Read the single path in Figure 2(a) as follows: at the starting point on the left, a stimulus occurs in the environment of the first peer (a message becomes available to be routed); the gh1 object in the associated peer process interprets the stimulus and causes the message to be forwarded to the gateway; the ic object in the gateway process interprets the stimulus and causes routing to be performed; the rt object in the gateway process determines the route; the oc object in the gateway process causes the message to be forwarded to the second peer; the gh2 object in the second peer process causes the message to be displayed; the result (the displayed message) is observed at the end point of the path in the environment of the second peer.

The map presents the causal sequence as a requirement or explanation; there is no implication that the causal sequence has a first-class representation in the software. In most implementations, including ACE, it will not have such a representation, but must be inferred from scattered details often buried deep in the code.

The style of thinking in terms of cause-effect sequences instead of intercomponent communication seems to be routinely used by domain experts to understand and explain how complex software systems work. However, it has been largely ignored by writers of



(a) A Use Case Map



(b) Discovering the Use Case Map

Figure 2: Simplest Use Case Map of the Gateway

method textbooks and vendors of CASE tools and, up to now, has lacked a first-class notational representation. Use case maps give it such a representation.

Figure 2(b) gives some indication of a use case map like the one of (a) may be discovered. Discovering the components requires understanding how the configurator, acceptor, and connector patterns perform initialization and how this is realized in code. Discovering the path requires understanding the details of the reactor pattern, its relationship to the other patterns, and its realization in code (although objects of the reactor pattern are regarded as details below the level of this map, the details must be studied to discover the map). All of this requires reading code and studying patterns papers [3][4][5][6][7] in which information is presented with a combination of prose, Booch class relationship diagrams, and object interaction diagrams, none of which give much insight into how individual patterns are connected together to form systems.

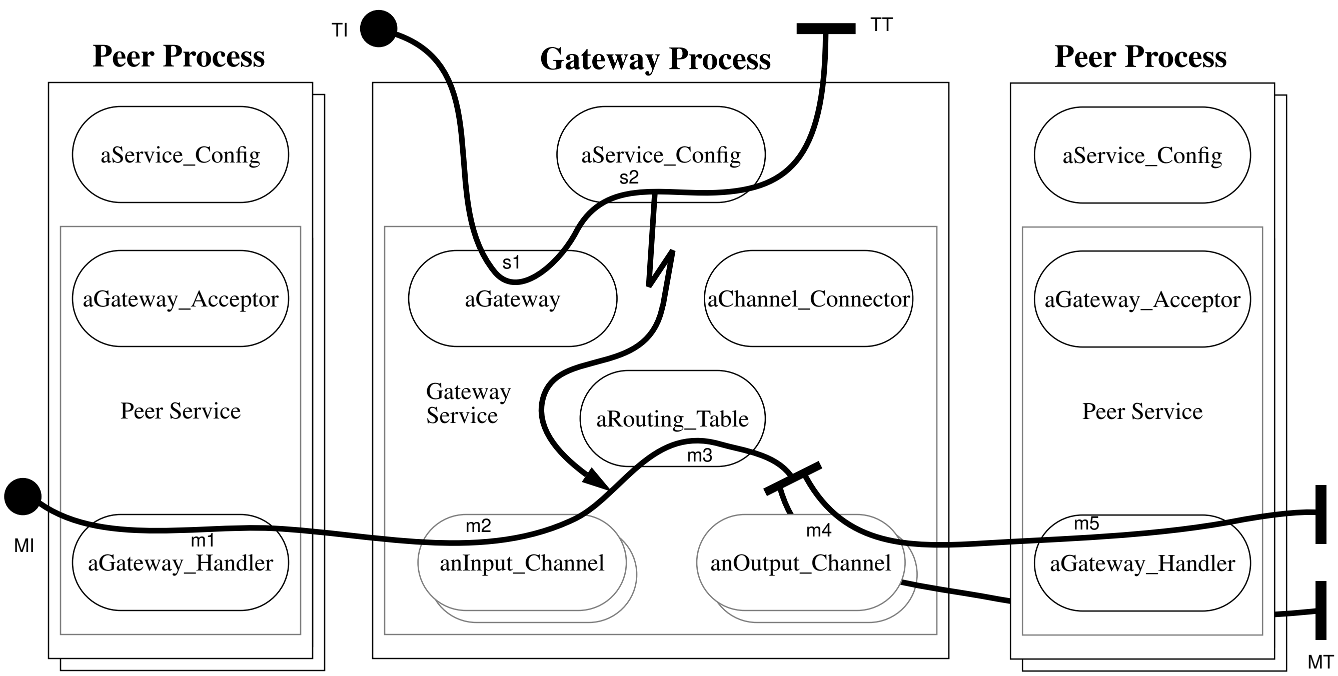
4.0 A Complete Use Case Map Model of the Gateway

We now present a more complete use case map model of the gateway in relation to ACE classes and patterns. In the descriptions, equivalent short-form and long-form component names are used as convenient: see Figure 5 for a cross-reference list.

4.1 Gateway Processes and Classes

Section 4.1 explains in some detail, with the aid of Figures 3, 4 and 5, how the components of the use case map in Figure 3 are derived from ACE patterns and classes, ignoring the use case paths. Not all of this detail is necessarily needed to understand the use case paths. Those who only want the paths explained may prefer to glance at Figures 3, 4, and 5 to get a sense of what they provide and then skip to Section 4.2. Figure 3 gives the structure of the gateway and peer processes. Figure 4 shows the class inheritance hierarchy for a portion of the ACE library and for the gateway and peer programs. Figure 5 cross-references the components in the use case map of Figure 3 to the classes in Figure 4 so the relationships can be seen at a glance.

Referring to Figure 3, we see that a peer process is made up of an object, aService_Config, and a team, Peer Service, which is in turn made up of two more objects, aGateway_Acceptor and aGateway_Handler. Looking at Figures 4 and 5 we see that aService_Config is an instance of the ACE Service_Config class and that aGateway_Acceptor is an instance of the ACE Acceptor



Preconditions:

MI: A message is available on the standard input device
TI: A SIGQUIT or SIGTERM signal has been issued

Responsibilities:

m1: Read a message from a user and send it to the gateway
m2: Read a message from the peer corresponding to the particular input channel
m3: Determine which output channel(s) to use (which peers are to receive the message)
m4: Send the message to the peer corresponding to the output channel
m5: Read the message from the gateway and display it for the user

s1: Interpret the signal received from the system
s2: Shutdown the gateway service and cause the gateway process to exit.

Postconditions:

MT: A message is displayed on the standard output device
TT: The gateway process has exited gracefully

Figure 3: Basic gateway operation including termination

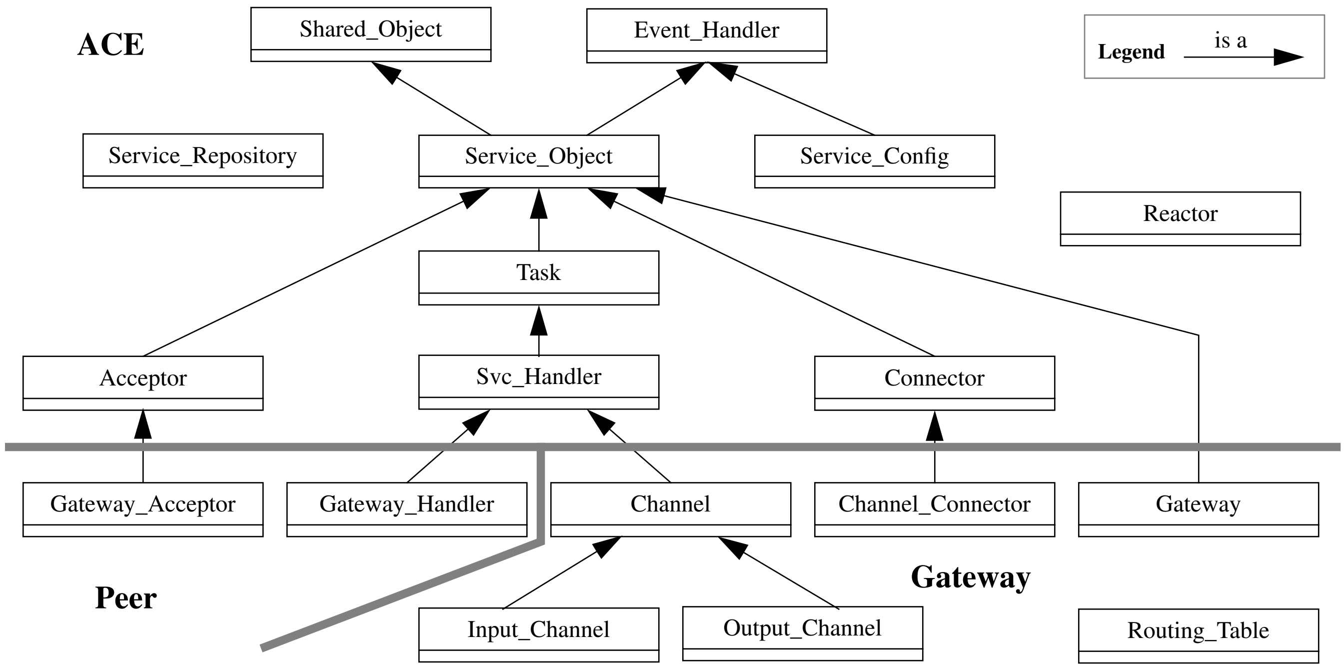
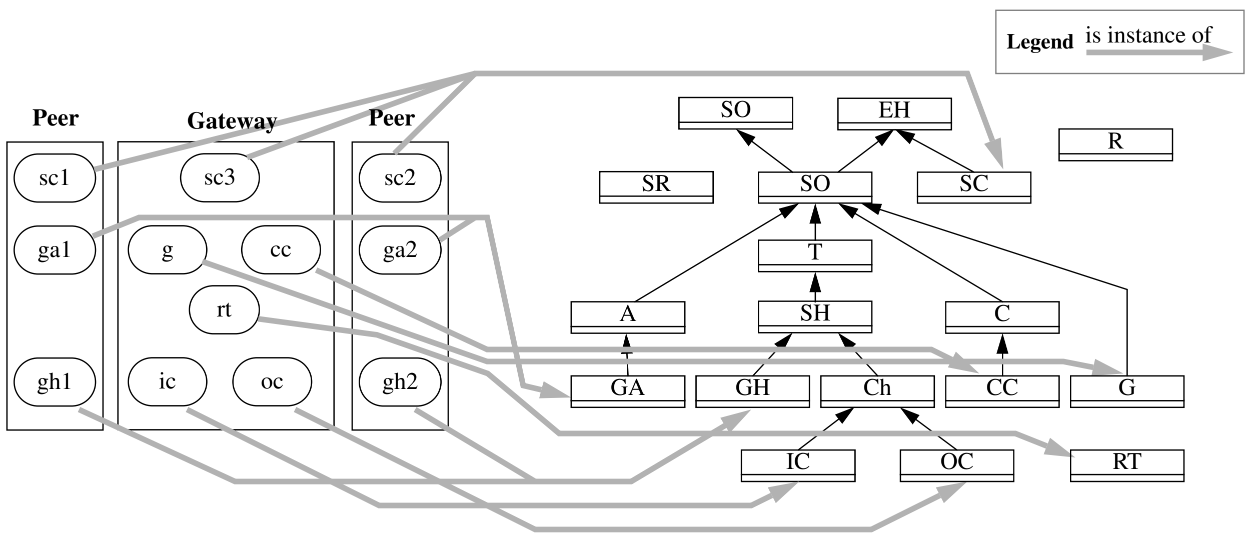


Figure 4: Gateway inheritance hierarchy



Tag	Instance Name
sc1, sc2, sc3	aService_Config
ga1, ga2	aGateway_Acceptor
g	aGateway
cc	aChannel_Connector
rt	aRouting_Table
gh1, gh2	aGateway_Handler
ic	anInput_Channel
oc	anOutput_Channel

Tag	Class Name	Tag	Class Name
SO	Shared_Object	GA	Gateway_Acceptor
EH	Event_Handler	GH	Gateway_Handler
SR	Service_Repository	Ch	Channel
SO	Service_Object	CC	Channel_Connector
SC	Service_Config	G	Gateway
T	Task	IC	Input_Channel
A	Acceptor	OC	Output_Channel
SH	Svc_Handler	RT	Routing_Table
C	Connector	R	Reactor

Figure 5: Component context/Class relationship mapping

class which inherits from Service_Object. Together, aService_Config and aGateway_Acceptor are the components forming the Service Configurator pattern. The object aGateway_Handler is an instance of the Gateway_Handler class which inherits from the ACE Svc_Handler class. The two objects, aGateway_Handler and aGateway_Acceptor, the latter, as indicated, being an instance of Gateway_Acceptor which inherits from Acceptor, are the components of the acceptor pattern. Each of the objects aService_Config, aGateway_Handler, and aGateway_Acceptor are instances of classes which eventually inherit from the ACE Event_Handler class. All of these objects are used as part of a reactor pattern.

The objects aGateway_Acceptor and aGateway_Handler work together to provide the communications functionality which is the main part of the peer process. Although the aService_Config object works with the aGateway_Acceptor object to provide a "service" according to the service configurator pattern, we can think of aGateway_Acceptor and aGateway_Handler forming a team which is created dynamically by aService_Config. This team is called the Peer Service and is represented in Figure 3 by a rectangle within the Peer Process and enclosing aGateway_Acceptor and aGateway_Handler. It is shown using dashed lines to indicate that it is dynamically created.

Figure 3 shows a pair of offset large solid rectangles under the Peer Process label. This is referred to as a stack of peer processes and indicates that there is more than one peer process but that they are all identical.

The structure of the gateway process consists of an instance of Service_Config and a team called Gateway Service which contains the objects aGateway, aChannel_Connector, aRouting_Table, anInput_Channel, and anOutput_Channel. As was discovered in the peer process, the object aService_Config is an instance of the ACE class Service_Config. Figures 4 and 5 show that the object aGateway is an instance of the class Gateway which inherits from the ACE Service_Object class. The objects aService_Config and aGateway form the service configurator pattern. The object aChannel_Connector is an instance of the ACE Connector class. The objects anInput_Channel and anOutput_Channel are instances of the classes Input_Channel and Output_Channel respectively which both inherit from the class Channel which in turn inherits from the ACE Svc_Handler class. The object aChannel_Connector, being a Connector, and the objects anInput_Channel and anOutput_Channel, being Svc_Handlers, together are the components of the connector pattern.

4.2 Generalized Use Case Map

The simple use case map of Figure 2 is generalized in Figure 3 to show a bit more detail in the normal operation of the gateway system and to include termination behaviour. The paths in Figure 3 are annotated according to a convention where the preconditions and postcondition are given upper case labels and the responsibilities are given numerically sequenced lower case labels.

Looking at the normal operation use case path which begins with the precondition MI, follows along responsibilities m1 through m5 and ends at the postcondition MT, we see three features that were not in Figure 2:

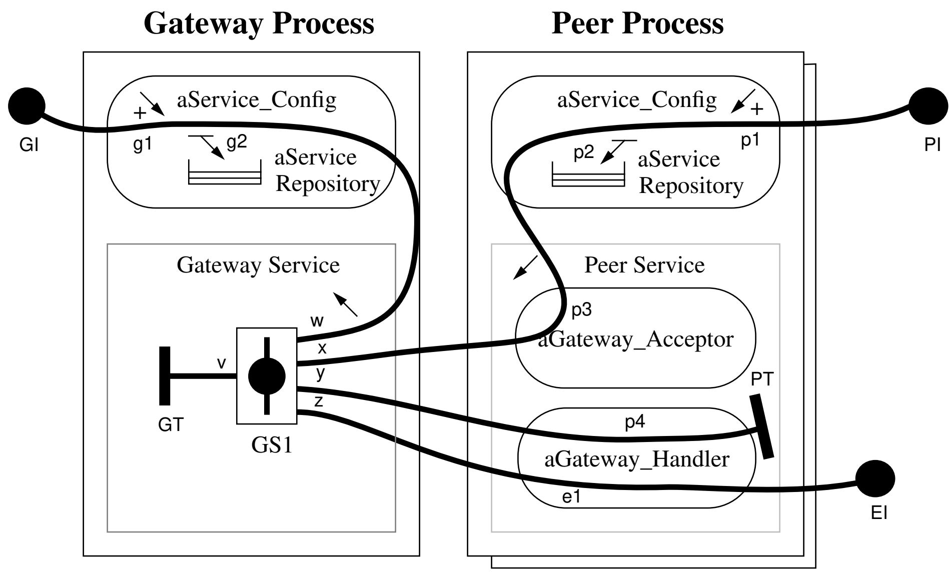
- The path crosses a stack of peer processes in the vicinity of the responsibility m1. When a path crosses a stack like this, it means that the path applies to any one of the components in the stack.
- Between the responsibilities m3 and m4, the path leads to a bar from which emerge two new paths. This one-to-many forking of the path is called an AND fork and it indicates that the emerging paths are logically concurrent.
- At responsibility m4, one logically concurrent path crosses over the top of the object stack Input_Channel while the other crosses between the objects of the stack (hidden beneath the object on top). This notation is used to indicate that each path applies to a different component in the stack.

The AND fork and stack notation make it easy to show how the gateway process forwards to multiple peer processes.

The path which follows from the precondition TI to the postcondition TT by way of responsibilities s1 and s2 shows how the gateway process is terminated. A stimulus, in this case a UNIX signal, occurs in the vicinity of the gateway process and is interpreted by aGateway at responsibility s1. The aGateway object commands aService_Config to shutdown the system which it does at responsibility s2. The zig-zag arrow between the paths means that this termination use case path aborts the normal operation use case path.

5.0 Use Case Maps for Gateway and Peer Initialization and Error Handling

Figure 6 shows a high level view of the Gateway Process and Peer Process initialization and error handling. Ignore for a moment the paths in this diagram and consider just the components. On the left is a view of the Gateway Process where much of the detail that was shown previously in Figure 3 has been abstracted away. We still have the Service_Config object, which has actually been elaborated upon slightly, but the



Preconditions:

GI: UNIX command issued to start gateway process
PI: UNIX command issued to start peer process
EI: Message becomes available as input to output peer

Responsibilities:

g1: Create an instance of the Gateway Service
g2: Store the new Gateway Service in the service repository

p1: Create an instance of the Peer Service
p2: Store the new Peer Service in the service repository
p3: Initialize and start the Gateway_Acceptor instance
p4: Initialize the Gateway_Handler instance

e1: Forward the message to the gateway

Postconditions:

GT: Gateway process initialized and operational
PT: Peer process initialized and operational

Figure 6: High level initialization of Gateway and Peer processes

remainder of the internals are hidden within the Gateway Service team. The new object shown within aService_Config, labeled aService_Repository, is a pool and is used to represent a place where dynamically created objects or teams can be stored and retrieved. To the right of the Gateway Process is the Peer Process, shown as it was given in Figure 3 with the exception that its Service_Config object has been expanded also.

Consider now the use case paths in Figure 6, starting at the upper left hand side with the Gateway Process initialization. The precondition for this path, labeled GI, is that the user types the appropriate UNIX command to start the gateway process. This action causes aService_Config to create an instance of the Gateway Service, shown by a plus sign and small arrow into the path at g1; store a reference to it in aService_Repository for later use, shown by the arrow with a tail at g2; and instantiate the Gateway Service slot with the newly created service, shown by the arrow leaving the path into the Gateway Service slot. The path then leads to a stub the box containing a darkened circle with a bar through it labeled GS1, where it is joined by other paths (indicated by x and z) also involved in the initialization and error handling of the Gateway Process.

A stub represents complex path detail that is not needed or desired for understanding a use case map at the level of abstraction where the stub is used. In this case, the stub is being used to hide the details of the initialization and error handling that occur in the internals of the Gateway Service.

As a result of the initialization of the Gateway Service, the Gateway Process is ready to service message traffic to and from the peers.

The initialization of the Peer Process, beginning at the upper right hand corner of Figure 6, proceeds at first in the same way as the Gateway Process: a user issues the UNIX command to start the Peer Process, indicated by PI, which causes aService_Config to create an instance of the Gateway Service (p1), store it in aService_Repository (p2) and instantiate the Peer Service slot. The path then continues with the initialization of the aGateway_Acceptor (p3) after which some interaction takes place with the Gateway Process (shown by the path entering the stub GS1 at x and leaving at y), which is followed by the initialization of the aGateway_Handler (p4). The result is that Peer Process is ready to handle messages (PT).

In addition to initialization, Figure 6 also shows some error handling. The error handling path shown is that which is followed for one particular kind of error. Figure 6 does not catalog all possible error scenarios but gives an example of how errors are typically handled.

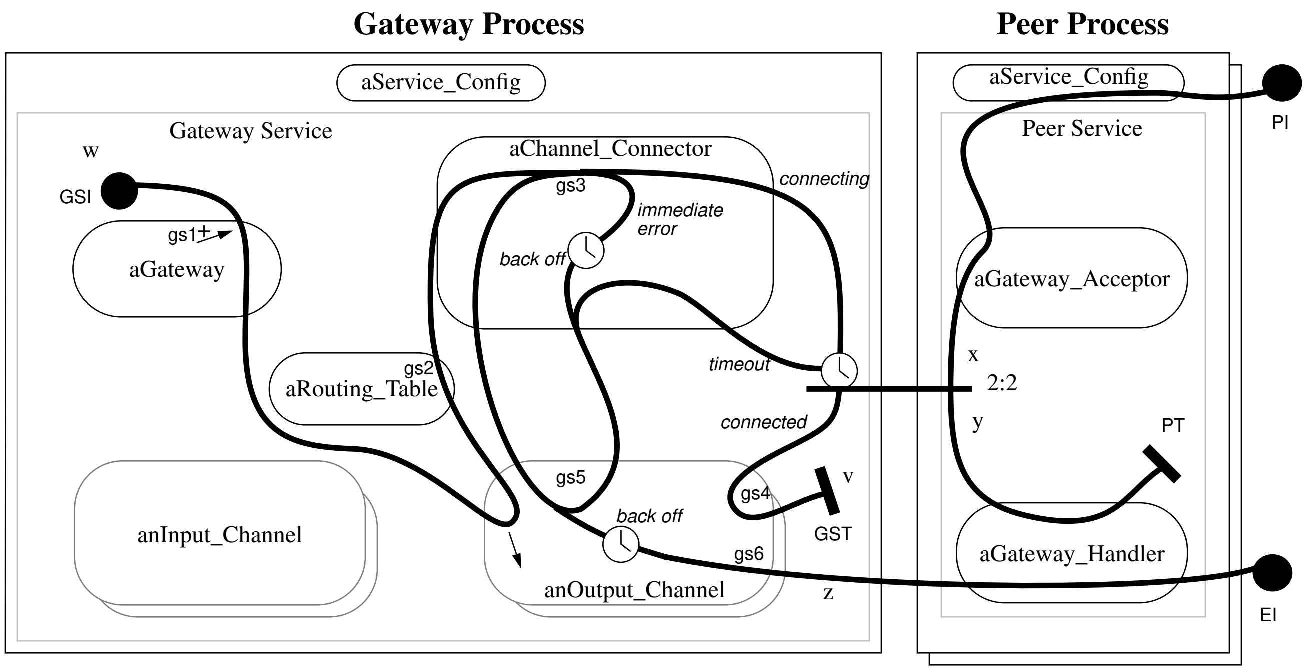
The error handling path begins at the lower right hand side of Figure 6. The precondition for this path is that the Peer Process be an output peer and that the user input a message into it. The message is read from the standard input of the peer and transmitted to the gateway (e1) where the error handling takes place (in the stub, GS1).

5.1 Expanding the Stub

The details of the Gateway Service initialization and error handling will be now considered. The high level view of the initialization and error handling shown in Figure 6 is expanded in Figure 7. Looking first at the components and ignoring the paths, we see that although the sizes of the objects have changed, the structure of Gateway Process looks the same as it did in Figure 3. The Peer Process is unchanged in structure from its representation in Figure 6 (and also Figure 3). The paths in Figure 7 are more detailed versions of the paths shown in Figure 6. In particular, the stub GS1 of Figure 6 has been expanded here to show the complex detail of the Gateway and Peer interaction during initialization and error handling. A path labeling convention helps to maintain continuity between the two diagrams: the paths at the stub in Figure 6 are labeled (v, w, x, y, and z) and these same labels are repeated on the corresponding paths in Figure 7.

The initialization of the Gateway Service begins at the upper left hand side of the diagram with the path starting at w. This path represents the connection of a communications channel from the Gateway Process to one of the output Peer Processes. Input peers are handled identically and, to keep the map simple, the paths for input peer connection are not shown. The scenario represented by this map is performed once for each output Peer Process that is to be connected. The first instance of this scenario is started when the path from aService_Config reaches the stub at the point labeled w in Figure 6. Subsequently, while there are still peers to connect, a new scenario instance begins each time a timer (a circle with hands, suggestive of a clock face) is started or whenever a scenario reaches the end of the path at v.

The scenario starts with aGateway creating a new instance of Output_Channel (gs1) and binding it to the network address of the peer to which it is to be connected. The Output_Channel instance is then dropped into its slot and the routing table is updated (gs2). Then, aChannel_Connector begins the attempt to connect the channel to its peer (gs3). At this point the path forks (towards the right) into two alternative paths. The lower path labeled *immediate error* is followed if the connect attempt (gs3) fails immediately; the upper path, labeled *connecting*, is taken otherwise. Consider first the *connecting* path which turns downwards and leads to a timer, positioned on a horizontal bar. Note that



Preconditions:

GSI: Channel needs initialization and timer started or GST reached
PI: UNIX command issued to start peer process
EI: Message becomes available as input to output peer

Responsibilities:

gs1: Create an instance of Output_Channel
gs2: Register the channel with the routing table
gs3: Initiate a connection on the newly created channel
gs4: Complete initialization of channel instance
gs5: Reset channel
gs6: Detect error condition

Postconditions:

GST: Channel to peer initialized and operational
PT: Peer process initialized and operational

Figure 7: Details of initialization and error handling of Gateway and Peer

the initialization path of the Peer Process also leads to this bar (towards the right hand side). The bar and timer represent a timed synchronization between the gateway and peer initialization paths which functions as follows. If either the gateway or peer initialization scenario “arrives” at the top of the synchronization bar first, it waits there for the other to arrive, after which they both proceed along the paths out the bottom of the bar (the gateway on the left and the peer on the right). The gateway initialization scenario will only wait for a certain period for the peer to arrive (shown by the timer). If this period expires, the path labeled *timeout* is followed by the gateway scenario. The peer, on the other hand, will wait for ever for the gateway (there is no timer for the peer).

On a successful synchronization, the gateway service initialization scenario proceeds along the path labeled *connected* to complete the initialization of the Output_Channel instance (gs4) resulting in a completely connected channel (GSF). If, instead, the timer expires, the path labeled *timeout* is followed from the timer to anOutput_Channel, where the channel is reset (gs4), and back to aChannel_Connector to restart the connection attempt (gs3). Now we return to the path labeled *immediate error*. In this case, the scenario is delayed to prevent successive retries from clogging the network (shown by the timer), after which the Output_Channel is reset (gs5) and a new connection attempt is made (gs3).

Let us now consider the error handling path that was first described in connection with Figure 6. It can be seen in Figure 7 to begin as it did in Figure 6 with the user entering a message into the output peer (EI). The path leads through aGateway_Handler where the message is transmitted to the Gateway Process where the reception of a message is interpreted by the anOutput_Channel as an error (gs7). A delay is introduced, as before, to avoid congestion (shown by the timer), the channel is reset (gs5), and an attempt is made to reconnect it (gs3).

The only new detail about the initialization path of Peer Process shown in Figure 7 (and not in Figure 6) is the synchronization with the Gateway Process which has already been explained. It is not discussed further here.

6.0 Relationship Between Use Case Maps and Message Sequences

A message sequence can be associated with a use case map to give the details of precisely what is happening along a path between components. The activities in the Gateway Service during initialization and error handling are complex and to understand them at a fine grained level the use case map given in Figure 7 can be supplemented by message sequences.

Immediate Connection

In Figure 8, the portion of the use case map of Figure 7 that is responsible for a immediately successful connection from the Gateway Process to an output Peer Process is highlighted in the map fragment on the left. On the right is a message sequence which corresponds to this same behaviour.

The message sequence is as follows: aChannel_Connector receives an initiate_connection message which prompts the Channel_Connector instance to attempt to connect to the peer which is completed without error. The Channel_Connector instance then initializes anOutput_Channel by sending it the open message. It in turn registers with aReactor via register_reactor and completes its initialization with initialize_connection.

Delayed Connection

The left portion of Figure 9 shows the same portion of the map fragment highlighted with the addition that the timer is now highlighted too. The message sequence for this behaviour is given on the right.

The sequence begins as above with the initiate_connection message to aChannel_Connector. Again the connect is attempted and this time, the result indicates that a connection may be possible at some point in the future. The Channel_Connector registers itself with aReactor to be notified of a completed connection by sending it the register_handler message. It also registers itself to be notified of a timeout expiry by sending the schedule_timer message to aReactor.

In this case, aReactor notifies aChannel_Connector by the handle_output message that the connection has been completed. The Channel_Connector instance cancels the timer by sending aReactor the cancel_timer message and then the Output_Channel initialization proceeds as above.

Connection Timeout

The connection timeout scenario is the same as the delayed connection scenario up to the schedule_timer message from aChannel_Connector to aReactor. Then instead of sending the handle_output message to aChannel_Connector, aReactor sends the handle_timeout message which indicates that the connection was not completed in time. The Channel_Connector instance forwards this message to the Output_Channel instance which restarts the connection attempt by sending the initiate_connection message.

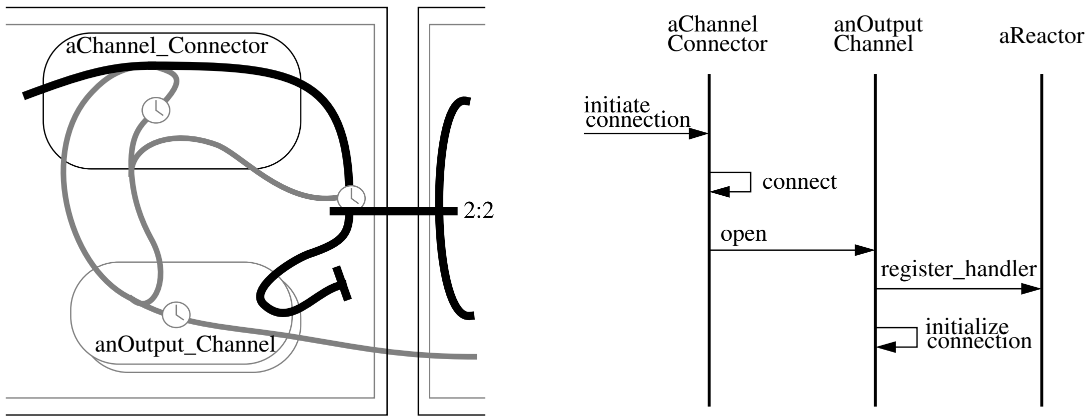


Figure 8: Message sequence for immediate connection case

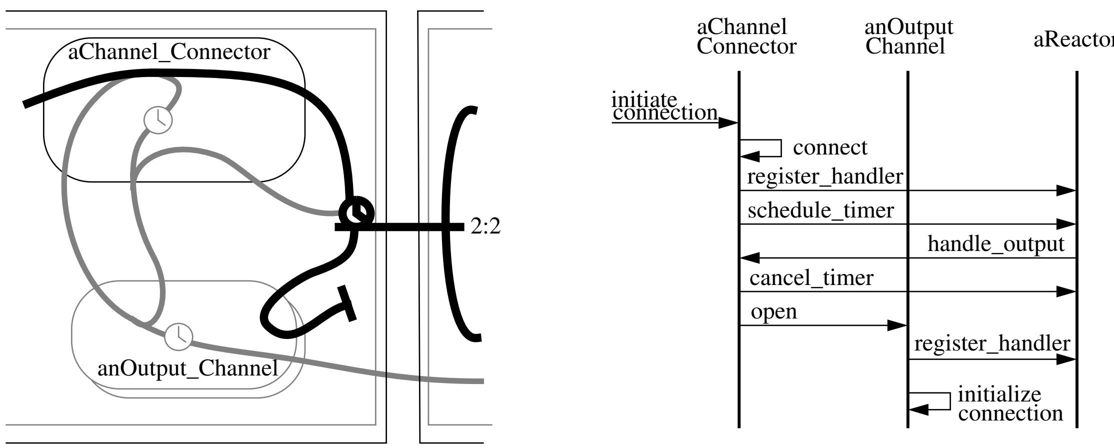


Figure 9: Message sequence for delayed connection case

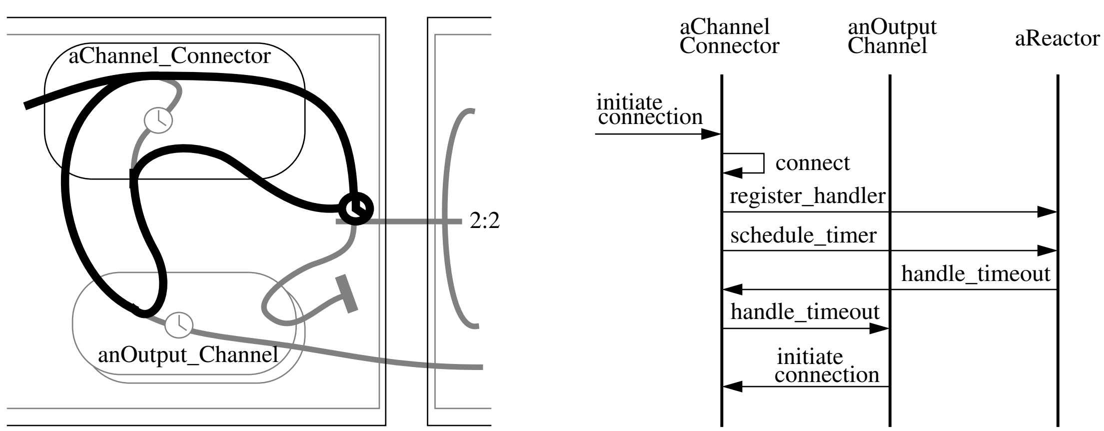


Figure 10: Message sequence for connection timeout case

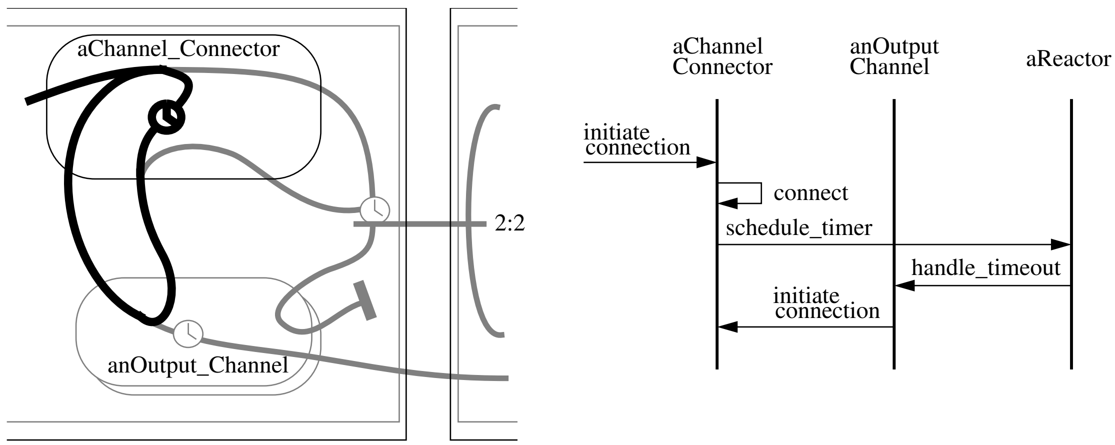


Figure 11: Message sequence for immediate failure case

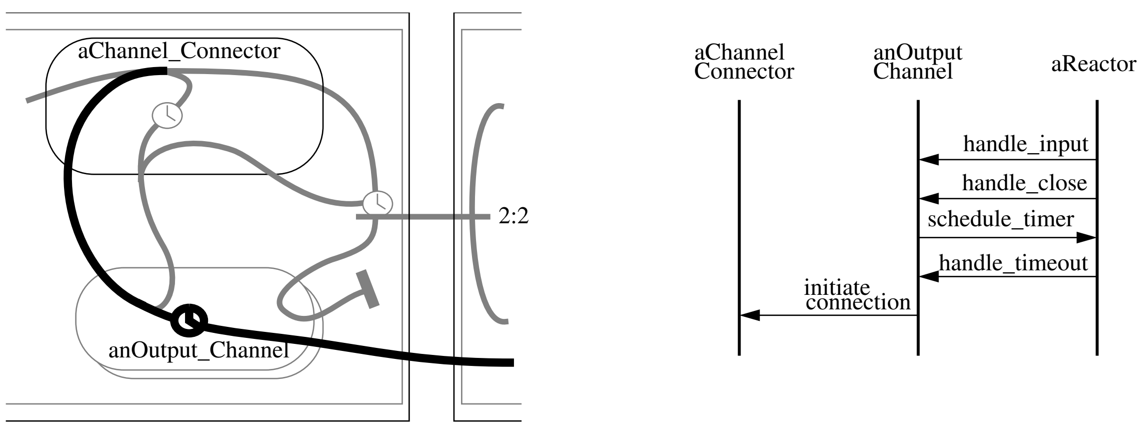


Figure 12: Message sequence for communications error case

Immediate Failure

Figure 11 shows the immediate failure case. Again the sequence begins with the initiate_connection message being received by aChannel_Connector. It attempts a connection which results in an indication that no connection is possible at all. It then schedules a timer with aReactor to delay for a while. The reactor informs anOutput_Channel that the timer has expired by sending it the handle_timeout message. The Output_Channel instance restarts the connection in the usual way.

Communications Error

Figure 12 shows the handling of a communications error. The sequence begins with a handle_input message from aReactor to anOutput_Channel. This is an error which causes anOutput_Channel to return an error condition to the aReactor which the sends the handle_close message. The channel registers a timer to delay for a while, after which time the reactor sends the handle_timeout message. This prompts anOutput_Channel to restart the connection.

7.0 Discussion

We have presented in three use case maps (Figures 3, 6, and 7, not counting Figure 2(a) which is a simplified version of Figure 3) a coherent overview of the operation of the gateway application that otherwise would have to be pieced together from many scattered details and held in the mind's eye and explained to others in terms of these details. The details are complex and keeping them in mind and explaining them is not easy without some higher-level view such as that presented by the use case maps. The maps are connected to the patterns from which the system is constructed through cross-references to classes and message sequences.

The use case map view presented here treats event demultiplexing and interprocess communication details as belonging to an underlying layer of software. This layer implements the reactor pattern, which is, therefore, not explicit in the maps.

The use case map view shows a number of components as objects that the reader might imagine should be threads, for example, the objects that occupy slots anInput_Channel and anOutput_Channel in Figure 3 and Figure 7. ACE supports making such objects either threads or passive objects but the Gateway software does not exercise the thread option. Instead, the UNIX process handles the multiple channels directly. However, the use case map is not sensitive to this choice. The precondition that a new scenario starts when a current one hits an end point or sets a timer covers either possibility. In one case, the waiting for timeouts is done in the lower layer of software mentioned above; in the other case, the threads can wait for timeouts directly.

8.0 Conclusions

Implementations constructed with frameworks like ACE hide the overall structure and behaviour of particular implementations behind a complex screen of disconnected, general pattern concepts and widely scattered customization details. Pattern experts and developers will form their own understanding of overall system structure and behaviour based on experience, but this leaves the understanding locked in their minds.

Use case maps have been presented as a means of enabling all to share this understanding. The important general properties of use case maps are: they are visual, to exploit human ability to recognize visual patterns; they combine structural and behavioural information in an insightful way through visual patterns in composite diagrams, to keep the mental overhead of relating structure and behaviour as low as possible; they represent behaviour in terms of causal paths taken by stimulus-response scenarios that traverse the system as a whole; and they may combine paths for many scenarios into composite maps that show recognizable visual patterns, not just individual scenarios. This is a unique and powerful combination of properties for understanding and explaining complex systems of many kinds, not just software systems constructed from frameworks.

Acknowledgments

This work was funded by NSERC and TRIO. We would like to thank Doug Schmidt for useful discussions.

References

[1] D. C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[2] R. J. A. Buhr, R. S. Casselman, *Use Case Maps for Object-Oriented Systems,* Prentice Hall, 1996

[3] D. C. Schmidt, “A System of Reusable Design Patterns for Application-level Gateways,” in *The Theory and Practice of Object Systems* (Special Issue on Patterns and Pattern Languages) (S. P. Berczuk, ed.) Wiley and Sons, 1995

[4] D. C. Schmidt, “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching,” in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt eds.), Reading, MA: Addison Wesley, 1995

[5] D. C. Schmidt, T. Suda, “The Service Configurator Framework: An Extensible Architecture for Dynamically Configuring Concurrent, Multi-Service Network Daemons,” in *Proceedings of the IEEE Second International Workshop on Configurable Distributed Systems*, Pittsburgh, PA, March 1994.

[6] D. C. Schmidt, “Acceptor: A Design Pattern for Passively Initializing Network Services”, *C++ Report*, SIGS, Vol 7, No. 8, November/December 1995.

[7] D. C. Schmidt, “Connector: A Design Pattern for Actively Initializing Network Services”, *C++ Report*, SIGS, Vol 8, No. 1, January 1996.