Use Case Maps for Engineering Real Time and Distributed Computer Systems: A Case Study of an ACE-Framework Application

R. J. A. Buhr, A. Hubbard

Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada

email: {buhr, alex}@sce.carleton.ca}

Copyright 1996 IEEE. Published in the Proceedings of the 30th Annual Hawaii International Conference on System Science, January 7-10, 1997, Maui, Hawaii.

Abstract

Two major problems in the engineering of software-intensive, real time and distributed computer systems are, without becoming lost in code details, I) understanding how an implemented system works as a whole, and II) specifying, before implementation starts, how the required behaviour of the whole system is to be achieved. These lead to other problems, such as, long iteration cycles during forward engineering while various code changes are tried in attempts to fix erroneous system behaviour, and inadvertently introducing code changes during maintenance or reengineering that will damage correct system behaviour because there is a lack of backwards traceability to it from the code. This paper illustrates the application of a new technique called use case maps to solving these problems, using as an example a system constructed from a public-domain, object-oriented, software framework called ACE.

1. Introduction

Problems I and II identified in the abstract are common to all kinds of software-intensive real time and distributed systems, such as, telephony switching systems, air traffic control systems, radar systems in military aircraft, or intranet-based support systems for enterprises such as travel agencies, to name but a few.

This paper aims to contribute to the exchange of ideas about better techniques for the engineering of such systems by showing, with a case study of a particular class of software-intensive system, how a new technique called use case maps [3][4][5][6] can help to solve Problems I and II. The class of system we have chosen for our case study is, we believe, a representative one from which general lessons may be learned. The example is a distributed application called the Gateway that is constructed from Schmidt's ACE (Adaptive Communication Environment) framework [12]. Frameworks like ACE provide standard software classes and patterns for quickly constructing new applications with good properties (the nature of patterns in this context is described in [8] and specific ACE patterns are described in [13][14][15][16][17]). However, the focus on construction exacerbates Problem I and II. The effect 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 situation is actually worse than for automobiles for a number of reasons: the system view we want of the software will not be a concrete thing like an automobile but an abstraction that may seem quite distant from code; the patterns embodied in frameworks used to construct the code are themselves described abstractly, so getting a system view on top of this 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 the creation, destruction, and changing interrelationships of operational parts such as threads and objects may cause its structural form to be fluid).

To cope with these issues we need a *conceptual model* with the following properties: has the primary objective of aiding human reasoning at a high level of abstraction, as opposed to entering details into a computer tool; is first-class at the macroscopic level, meaning not dependent on details of components or code; combines system behaviour and system structure into a single coherent view; expresses "morphing" compactly, without requiring sequences of snapshot diagrams of changing structural forms; has diagrams that are easily grasped as visual patterns for a system as a whole; provides a macroscopic system view for forward engineering, reverse engineering, maintenance, evolution, and reengineering; can be saved for documentation and maintained without unreasonable effort.

This combination of properties is found, to the author's knowledge, only in use case maps (see Section 7.0 for discussion of this point in relation to other design notations). Use case maps provide a supplementary view at a higher level of abstraction than other design notations.

Because our example is an application developed in the first place without use case maps, the paper necessarily tackles Problem I first. However, we hope that the reader will find it as obvious as we do that any technique that helps with Problem I in the reverse engineering direction is likely also to be helpful for Problem II when applied in the forward engineering direction (see Section 7.0 for more on this point).

2. ACE and the Gateway application

The nature of Gateway application is indicated by Figure 1. The code for this application is provided as an example with ACE. ACE is a framework for constructing applications of this general kind, in which a distributed set of computers achieves some overall purpose. This particular application is a small distributed system in which peer

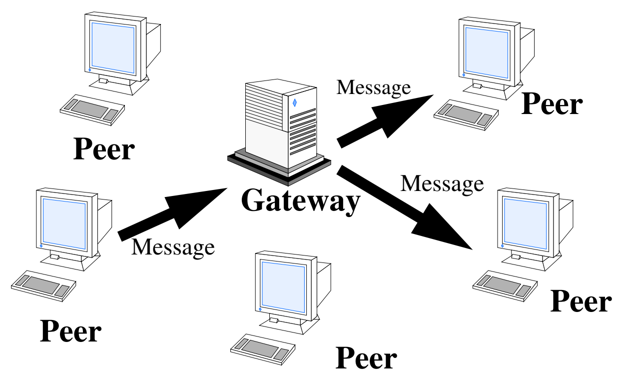


Figure 1: The Gateway system

workstations interact through a gateway computer. From the point of view of the gateway computer, 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 [13][14][15][16][17] (a pattern in this context is a catalogued general solution to a detailed implementation problem). Each ACE pattern is described in an abstract way in isolation from other patterns with a combination of prose and standard object-oriented diagramming techniques that show the classes, objects, and message sequences of the pattern. 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 and must be customized for particular applications, so any particular application is visible only in the customization details. Therefore, piecing together a simple, overall picture of the gateway software analogous to Figure 1 for the hardware requires chasing and mentally integrating a lot of detail in both abstract descriptions and code.

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 by providing a means of registering 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. 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 presents a view of normal, failure-free operation of the software 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).

The term *use case map* has its origins in the term *use case* [9]. A *use case* is a prose description of system behaviour that may embody a set of related scenarios of system operation. Use case maps are a diagramming technique for superimposing paths for scenarios of possibly many use cases on a diagram substrate that shows the structure of a system. The word *use* does not necessarily imply human users or users outside the system. In general, a use case map may embody scenarios involving many systems, subsystems, or components that are users of each other.

The structural substrate of Figure 2(a) is a set of boxes with different shapes and positional relationships to each other. In this diagram, boxes with sharp corners represent

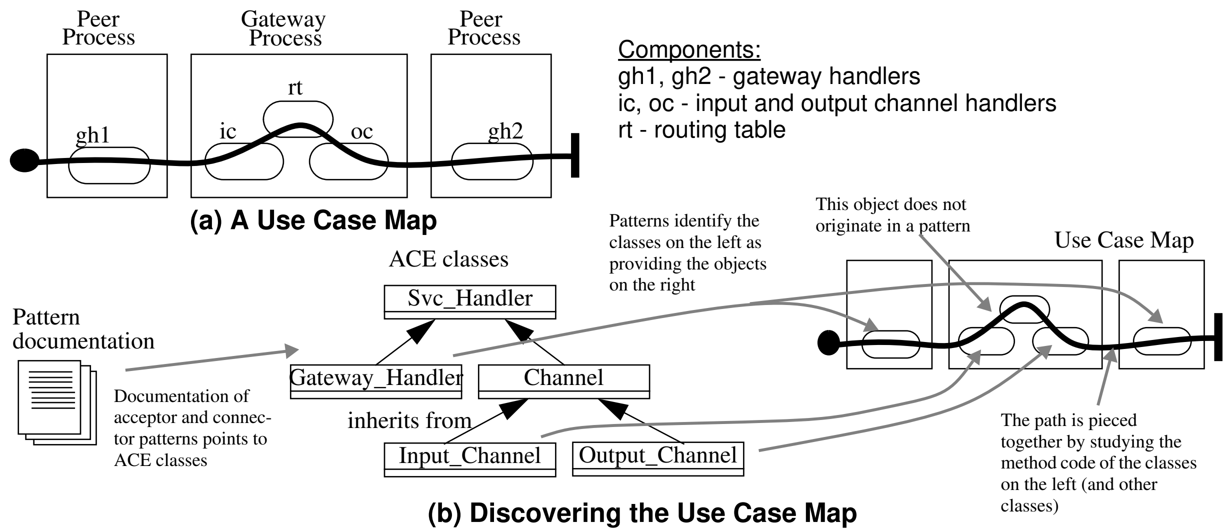


Figure 2: Simplest Use Case Map of the Gateway

UNIX processes that provide the operational context for ACE objects, and boxes with round corners inside the process boxes represent ACE objects that operate in the context of the processes. The objects gh1, gh2, ic, oc are all service handlers that come from ACE's acceptor and connector patterns, which provide classes for such objects. 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). 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"). Responsibilities are identified by names adjacent to points along paths (left off here to convey the uncluttered essence).

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.

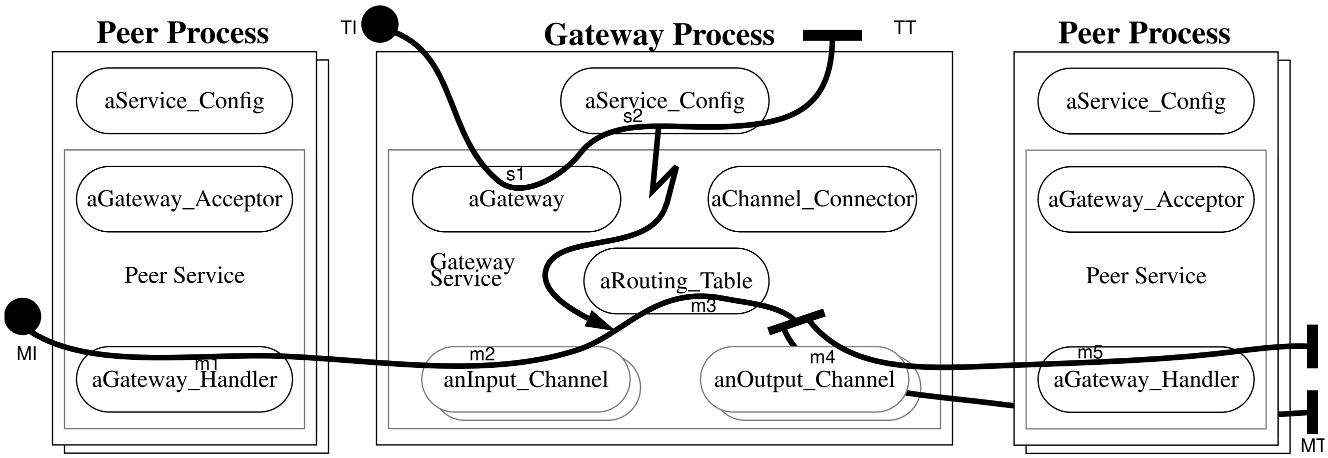
The map presents the causal sequence as a requirement or explanation; there is no implication that the causal sequence has a direct representation in the software. In most implementations, including ACE, it will not have such a representation, but must be inferred from scattered details often buried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

This style of thinking, in terms of cause-effect sequences instead of intercomponent communication, seems to be routinely used by experts to understand and explain how complex software systems work. However, it has been largely ignored by writers of method textbooks and vendors of CASE tools and, up to now, has lacked a first-class notational representation (see Section 7.0 for more on this). Use case maps give it such a representation.

Figure 2(b) gives some indication of how a use case map like the one of (a) may be discovered during reverse engineering. Discovering the components requires understanding how ACE's 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).

4. A complete Use Case Map Model of the Gateway

The following figures present a more complete use case map model of the gateway and its relationship to ACE classes and patterns. In the prose descriptions that follow, equivalent short-form and long-form component names are used as convenient (see Figure 5 for a cross-reference list).



Preconditions:

MI: A message is available on the standard input device

TI: A SIGQUIT or SIGTERM signal has been issued

Postconditions:

MT: A message is displayed on the standard output device

TT: The gateway process has exited gracefully

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.

Figure 3: Basic gateway operation including termination

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 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 a dynamically populated slot.

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

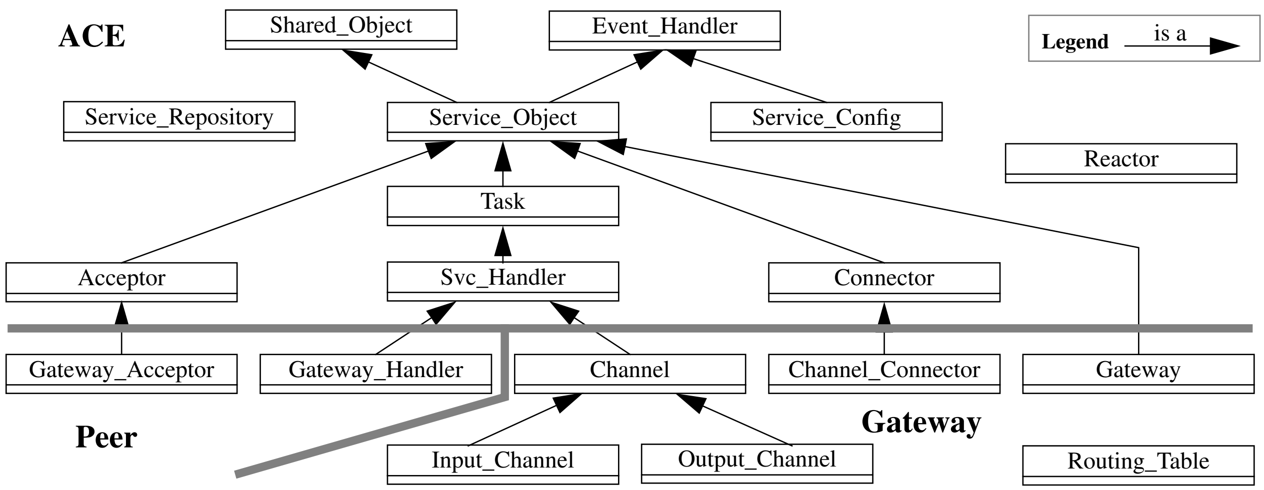
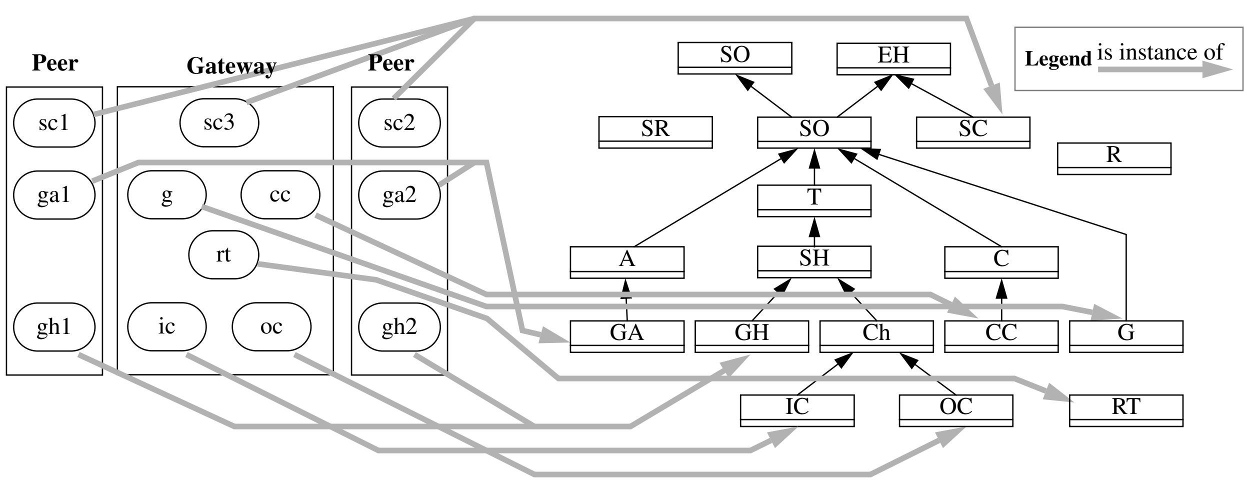


Figure 4: Gateway inheritance hierarchy



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

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 postconditions 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 MI, follows along responsibilities m1 through m5 and ends at MT), we see three notational 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 that the gateway process forwards messages to multiple peer processes.

The path from TI to TT by way of responsibilities s1 and s2 shows how the gateway process is terminated. A terminating stimulus, in this case a UNIX signal, occurs 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 lightning strike symbol between the paths means that the termination use case path aborts all activity anywhere along the normal operation use case path.

5. Use Case Maps for Gateway and Peer initialization and error handling

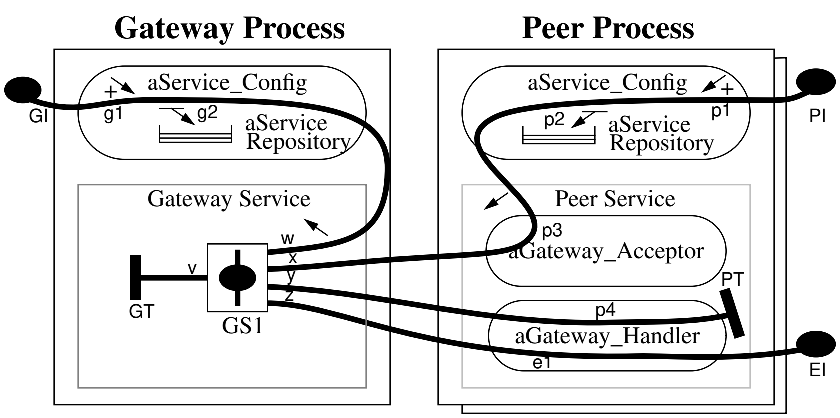
Figure 6 shows a high level view of the Gateway Process and Peer Process initialization and error handling. Ignore for a moment the paths in this diagram and consider just the components. On the left is a view of the Gateway Process where much of the detail that was shown previously in Figure 3 has been abstracted away. We still have the Service_Config object, which has actually been elaborated upon slightly, but the 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 as data 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 GI with the Gateway Process initialization. The precondition for this path 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 populate 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, begins at PI in Figure 6 and proceeds at first in the same way as the Gateway Process: a user issues the UNIX command to start the Peer Process which causes aService_Config to create an instance of the Gateway Service (p1), store it in aService_Repository (p2) and populate 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



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 (erroneous use)

Postconditions:

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

Responsibilities:

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

e1: Forward the message to the gateway

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

Figure 6: High level initialization of Gateway and Peer processes

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).

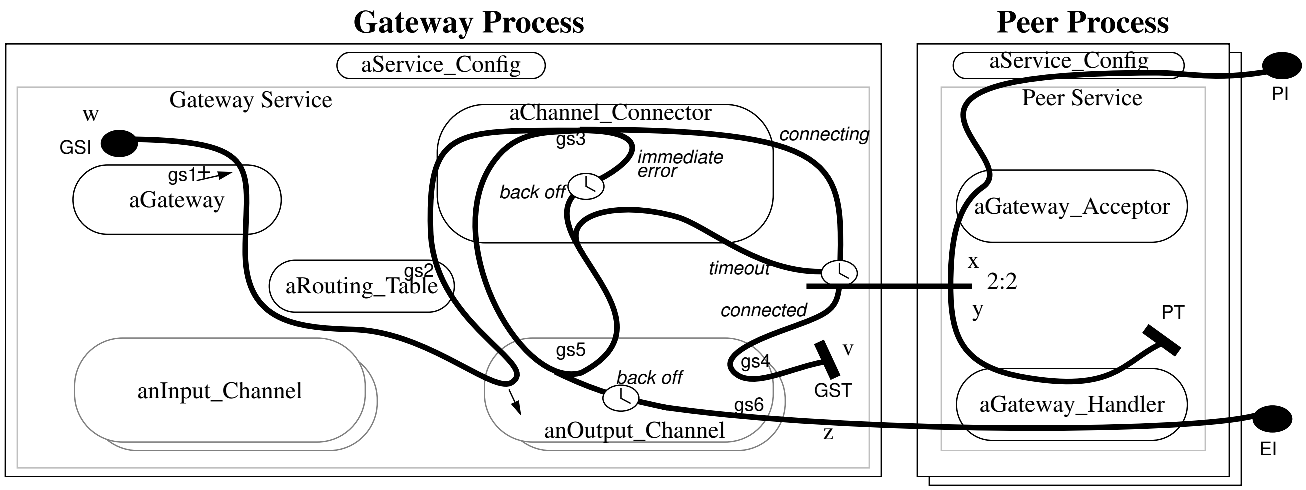
In addition to initialization, Figure 6 also shows an error handling path beginning at EI. The precondition for this path is that the Peer Process be an output peer and that the user erroneously inputs a message to it. The message is read from the standard input of the peer and transmitted to the gateway (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 to show more detail. 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 the stub end point 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 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



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

Postconditions:

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

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

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

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. Relationship between Use Case Maps and Message Sequences

Message sequences can be associated with use case maps to give the details of precisely what is happening between components along paths. The message sequences we shall now describe for this example were derived by reverse engineering, before drawing the use case maps, but it should be obvious that, in principle, use case maps could be developed first from a blank sheet during forward engineering and then used as a starting point for specifying message sequences to realize them (see Section 7.0 for more on this).

Note that the message sequences about to be described may be viewed as combinations of the message sequences of the patterns embodied in ACE. However, the procedure for discovering them was to read the code first. The patterns entered in only as a means of verifying that the sequences had been understood correctly.

6.1 Immediate connection

Figure 8 highlights the portion of the use case map of Figure 7 that is responsible for an immediately successful connection from the Gateway Process to an output Peer

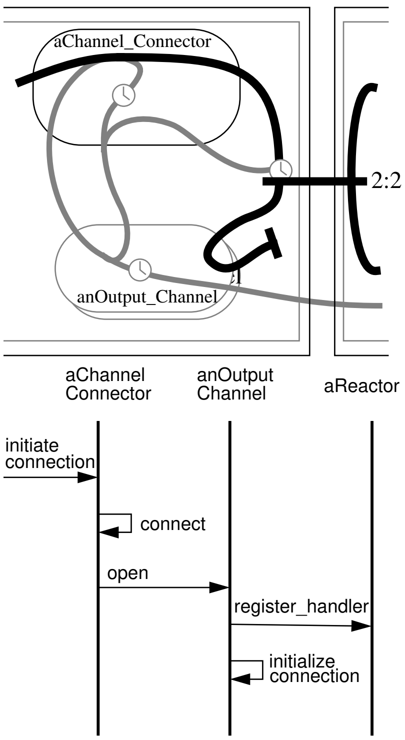


Figure 8: Message sequence for immediate connection

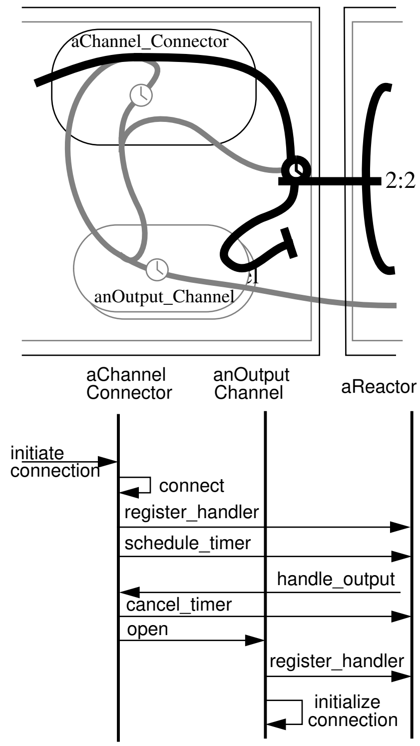


Figure 9: Message sequence for delayed connection

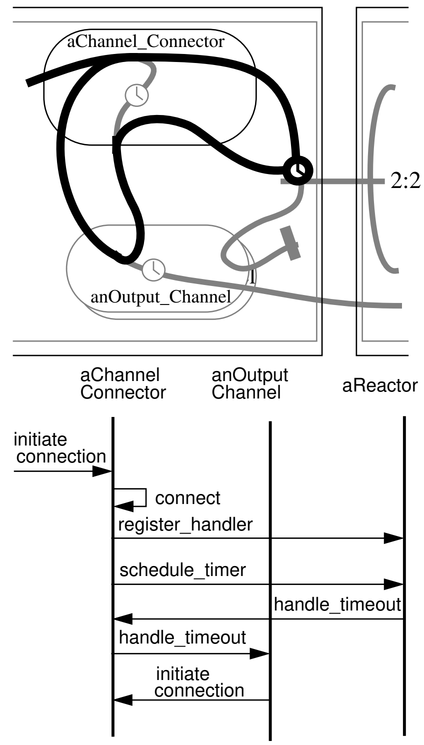


Figure 10: Message sequence for connection timeout

Process: aChannel_Connector receives an initiate_connection message which prompts the Channel_Connector instance to attempt to connect to the peer which is completed successfully. The Channel_Connector instance then initializes anOutput_Channel by sending it an open message. It in turn registers with aReactor via register_reactor and completes its initialization with initialize_connection.

6.2 Delayed connection

Figure 9 highlights the same portion of the map fragment with the addition that the timer is now highlighted too. The message sequence begins as above with the initiate_connection message to aChannel_Connector. Again the connect is attempted but 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.

6.3 Connection Timeout

The connection timeout scenario highlighted in Figure 10 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.

6.4 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. The Channel_Connector then registers itself with aReactor to be notified of a timeout expiry by sending the schedule_timer message to aReactor. 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.

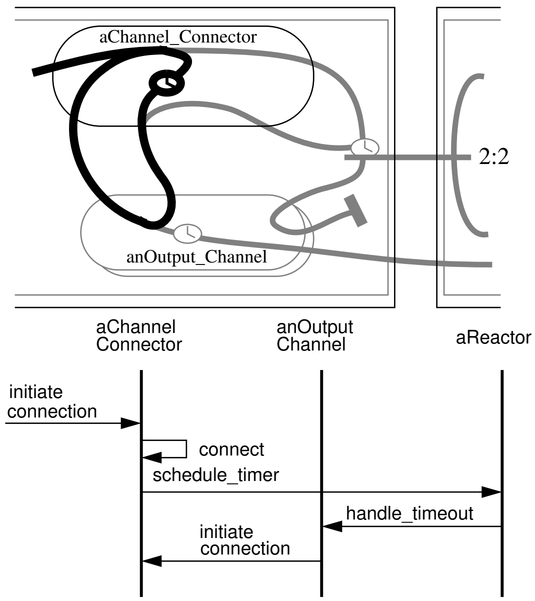


Figure 11: Message sequence for immediate failure

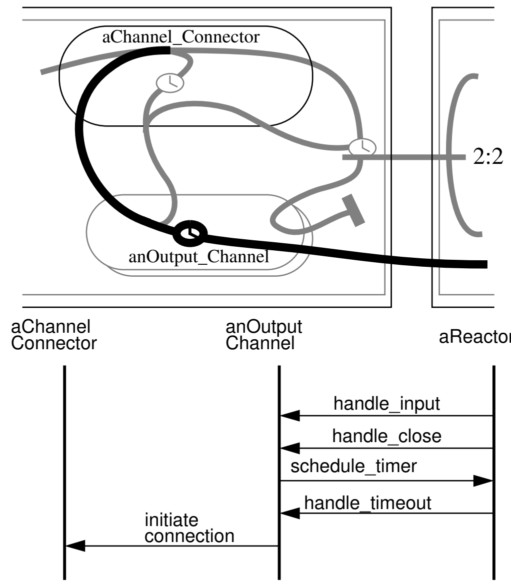


Figure 12: Message sequence for communications error

6.5 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 then registers itself with aReactor to be notified of a timeout expiry by sending the schedule_timer message to aReactor. When the timer expires, the reactor sends the handle_timeout message. This prompts anOutput_Channel to restart the connection.

7. Discussion

7.1 What we have achieved for the Gateway application

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 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 ([3] shows how to indicate such behaviour with use case maps, using additional notation that distinguishes threads—called processes in [3]— from objects).

7.2 Properties and application of Use Case Maps

Here is a brief overview of the relationship of other design diagramming techniques to the properties claimed in Section 1.0 for use case maps:

1. *Has the primary objective of aiding human reasoning at a high level of abstraction, as opposed to entering details into a computer tool.* Use case maps are the only diagramming technique known to the authors that was shaped solely by the need for this property. Others, e.g., [10][9][18] [7][2][1][11], are shaped primarily by the need for machine-executability of design

models and/or machine-translatability into code, thus forcing a commitment to details at the level of methods, functions, messages, interprocess communication, interfaces, internal state machines of components, etc., that get in the way of reasoning at a high level of abstraction.

2. *Is first-class at the macroscopic level, meaning not dependent on details of components or code.* There is only one other notation that has this property, the so-called "high level message sequence charts" under development by the Z120 community [7] (this reference covers only detailed message sequence charts, but examples of proposed high level ones are given in [10]). However, this notation does not possess Property 3 and it clouds the mind's eye with boxes in the separate behaviour diagrams that look like components but are not, exacerbating the problem of mentally superimposing behaviour on structure.

3. *Combines system behaviour and system structure into a single coherent view.* To the author's knowledge, only use case maps possess this property at a high level of abstraction. Other diagramming techniques may attempt to do it by superimposing sequence numbers on connections in structural diagrams to indicate, say, interobject or interprocess message sequences, but this requires many diagrams to present the big picture, thus clouding the mind's eye with details. Approaches that use separate diagrams, such as detailed or high level message sequence charts [7] [10][9][2] cloud the big picture by forcing humans to combine diagrams in the mind's eye.

4. *Expresses "morphing" compactly, without requiring sequences of snapshot diagrams of changing structural forms.* To the authors' knowledge, only use case maps possess this property (through the use of slots and move arrows in ordinary use case maps).

5. *Has diagrams that are easily grasped as visual patterns for a system as a whole.* Use case maps can combine many behaviour patterns in single diagram in a way that enables the mind's eye to sort them out. Recognizing behaviour patterns in superimposed sequence numbers or separate detailed message sequence charts is much more difficult, particularly because many diagrams must be viewed.

6. *Provides a macroscopic system view for forward engineering, reverse engineering, maintenance, evolution, and reengineering.* Only use case maps and high level message sequence charts provide reference views that are independent of details and so can be used to guide decisions about details. Use case maps do it more compactly and simply (Properties 2-5).

7. *Can be saved for documentation and maintained without unreasonable effort.* The avoidance of commitment to details and the compactness of use case maps contributes to this property. However, tool support is desirable (a use case map editor is currently being developed for this purpose).

It is important to understand that use case maps do not replace the other techniques referred to above, but supplement them to give a higher level view.

Although we have not used the term *architecture* up till now [19], we believe that the properties described above make use case maps a new, useful and practical form of architectural description [5].

Steps for reverse engineering an implementation into use case maps are suggested by the arrows in Figure 2(b) and the associated prose explanations (see [4] for a more detailed discussion of steps). Roughly speaking, forward engineering reverses the arrows and the steps in Figure 2(b). Use case maps like the one on the right in this figure are the first step (see [3] for suggestions and examples of how to come up with such maps from a blank sheet). These are used to identify classes/detailed patterns to implement them, including patterns of message sequences. The details are captured by conventional means. There is also the possibility of developing classes/detailed patterns and use case maps in parallel, bringing them together as work proceeds.

Current work by Ph.D. student Francis Bordeleau is producing a systematic method for a forward engineering process that goes from use case maps to detailed message sequence charts to communicating state machines, working with the developers of the ROOM methodology [18] and the associated ObjectTime CASE tool. A thesis and papers will emerge in due course (watch http://www.sce.-carleton.ca/ftp/pub/UseCaseMaps for developments).

8. Conclusions

Two basic problems have been identified in the engineering of software-intensive real time and distributed systems. *Problem I:* Understanding how an implemented system works as a whole may be difficult without becoming lost in code details. *Problem II:* Specifying, before implementation starts, how the required behaviour of the whole system is to be achieved may be difficult without becoming lost in the details of specifying how code is to be implemented. These two problems lead to other problems, such as, long iteration cycles during forward engineering while various code changes are tried in attempts to fix erroneous system behaviour, and inadvertently introducing code changes during maintenance or reengineering that will damage correct system behaviour because there is a lack of backwards traceability to it from code. This paper has illustrated, by means of a case study, an approach to solving Problem I and provided pointers to solving Problem II with the same approach.

The approach centers around use case maps to provide a supplementary high level view of system behaviour and structure that can be used as a starting point for defining details or a context for changing them. 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 defining complex systems of many kinds.

9. Acknowledgments

This work was funded by NSERC and TRIO. We would like to thank Doug Schmidt for useful discussions and the reviewers of this paper for helpful comments.

10. References

[1] G. Booch, Object-Oriented Design, Benjamin/Cummings, 1994.

[2] Grady Booch, James Rumbaugh, *Unified Method for Object-Oriented Development*, Documentation Set, Version 0.8, Rational Software Corporation, 1995.

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

[4] R.J.A. Buhr, R.S. Casselman, T.W. Pearce, *Design Patterns with Use Case Maps: A Case Study in Reengineering an Object-Oriented Framework*, SCE 95-17, http://ftp.sce.carleton.ca/UseCaseMaps/dpwucm.ps.

[5] R.J.A. Buhr, *Use Case Maps for Attributing Behaviour to Architecture*, Fourth International Workshop on Parallel and Distributed Real Time Systems (WPDRTS), April 15-16, 1996, Honolulu, Hawaii, http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/attributing.ps.

[6] R.J.A. Buhr, *Design Patterns at Different Scales*, presented at PLoP96, Allerton Park Illinois, Sep 96. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/plop.ps.

[7] CCITT Recommendation Z120: Message Sequence Charts (MSC), undated document.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissades, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[9] I. Jacobson et. al., *Object-Oriented Software Engineering (A Use Case Driven Approach).* ACM Press, Addison-Wesley, 1992.

[10] B. Regnell, M. Andersson, J. Bergstrand, *A Hierarchical Use Case Model with Graphical Representation*, Proc. ECBS96, IEEE Second International Symposium and Workshop on Engineering of Computer Based Systems, March 1996.

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson, *Object-Oriented Modelling and Design*, Prentice Hall 1991.

[12] 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.

[13] 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

[14] 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

[15] 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.

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

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

[18] B. Selic, G. Gullickson and P.T. Ward, *Real-time Object-Oriented Modeling, Wiley*, 1994.

[19] Shaw and Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.