

A Possible Design Notation for Aspect Oriented Programming

R.J.A. Buhr

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

buhr@sce.carleton.ca, www.sce.carleton.ca/faculty/buhr

ABSTRACT. *Aspect Oriented Programming (AOP) cuts across (and supplements) conventional programming to focus on emergent entities. In an analogous fashion, a visual notation called UCMs (Use Case Maps) cuts across (and supplements) conventional program-design notations to focus on emergent entities. Where AOP represents emergent entities in textual terms, the UCM notation represents them as visual paths. The notation was developed as an aid to designing and understanding emergent entities in complex systems, without having AOP in mind (or even knowing about it). However, there is so much commonality of ideas that the UCM notation may be ready made to be to AOP what notations like Rational Rose and UML are to conventional programs. In its favour for this purpose are the following facts: the notation is now relatively mature after several years of development, is thoroughly documented in a book and many papers, has a supporting demonstration-of-concept graphical tool which is being used to front-end it to some interesting applications of the kind for which AOP is intended, and has been tested and refined on practical systems of industrial scale and complexity.*

1 Introduction

This position paper is being submitted at the urging of Gregor Kiczales to get the ideas in front of the AOP community. It has been written without any more personal knowledge of AOP than I have gained from listening to Gregor's talk at OOPSLA 97 and having an hour long meeting with him just before submitting this. However, this was enough to reveal a strong commonality of ideas between the UCM (Use Case Map) notation and AOP. The UCM notation was developed without having AOP in mind (or even knowing about it). However, the possibility exists that the UCM notation is ready made for AOP. This paper aims to put that idea in front of the AOP community and to provide enough information to enable a preliminary judgment to be made.

For expediency, this paper is adapted from a position paper on UCMs for a software architecture audience and therefore it introduces UCMs as a new way of looking at software architecture, not as a notation for AOP. Section 1.1 returns to the AOP theme. See [1][2][3] for more explanation of UCMs and references to other UCM publications.

A Missing Element of SW Architecture. Software architecture is concerned in part with *organiza-*

tional structures of systems, often diagrammed as arrangements of boxes that show how system components (the boxes) fit together to form individual members of a family of systems or products. Often, diagrams used to describe organizational structures include additional visual information that I shall lump here under the term *wiring*, meaning annotations and connecting lines that define control/communication paths between boxes, types of quantities flowing between boxes, interface elements of boxes that provide control and access, and so forth. Glance through any work on software architecture, e.g., [4], to see many diagrams of this type.

Behaviour is an important element of architecture, but describing it in terms of wiring bogs us down in details that are not architecturally interesting. A person should be able to understand system behaviour in architectural terms without having to resort to such details. This requires understanding *in the large* how organizational structures and behaviour of whole systems are interrelated. For this purpose, we need a missing concept of macroscopic *behaviour structures*, cutting across organizational structures, and at the same high level of abstraction.

The Concept of Macroscopic Behaviour Structures. On one hand, an uninitiated reader may find it difficult to imagine what a macroscopic "behaviour structure" might be. This is because, in the final analysis, behaviour is whatever *emerges* from details associated with wiring. Diagrams of behaviour in common use (e.g., timing diagrams, message sequence charts) are not suitable "behaviour structures" for a number of reasons: They are too strongly dependent on details that may easily change without changing architecture. The sheer volume of detail clouds the big picture. Much of the detail is local boilerplate that gives little insight into the big picture. The form of the diagrams (temporal sequences related to timelines, separate from diagrams of organizational structure) fails to give much architectural insight.

On the other hand, concepts of macroscopic behaviour exist in the mind of anyone who deals with complex systems. Consider the concept of a *transaction*. A transaction, such as opening a www web page, is understood to involve the joint efforts of many elements of a complex system, in ways that may be very complex in detail. One does not have to know the details to think about a transaction as a something that exists independent of the details. A suitable mental picture is of causal paths through a system. In these terms, a transaction to open a web page is easily visualized as a causal path

terms as plugging in a sub-UCM at some point along the path (called a dynamic *stub*). Slots, dynamic stubs, and the actions that affect them are identified by visual notations that enable large scale dynamic situations to be understood at a glance. With these notations, a person may create understandable *fixed* UCMs that *imply* very complex dynamic situations at a more detailed level. Thus a major problem with more detailed notations is overcome, that of trying to understand large scale dynamic situations in terms of time varying organizational structures and/or detailed message sequences that drive the changes.

UCMs may be used as a front end for developing details. A UCM editor has been developed which we are using as a design front end for dynamic agencies (systems of agents that change their organization and behaviour over time) [3] and for performance analysis of architectures (with UCMs, performance becomes a functional property of architecture, associated with paths, rather than a non-functional property, as it is usually considered to be).

Section 3 provides examples and more information on the UCM editor.

UCMs and OO. How OO classes enter the architectural picture is outside the scope of this paper but is described in the UCM references. Briefly, high level class relationship diagrams (omitting class variables and methods) are regarded as partners of UCMs in a combined representation that can be viewed as a generalized form of system architecture. Classes are not used to describe behaviour in this representation. This high level view complements more detailed views provided by other techniques.

History and Status of UCMs. The notation is now relatively mature after several years of development, is thoroughly documented in a book and many papers (see [1][2][3] for more detailed explanations than this paper can provide, for many examples, and for references to other UCM publications), is supported by a demonstration-of-concept graphical editor, and has been tested in many places in industry on complex software systems of the kind for which AOP is intended.

1.1 UCMs and AOP

The key ideas in common between UCMs and AOP appear to be the following:

- representation of *emergent entities* as first class quantities; and
- cross-cutting of different design representations.

As I see it, UCMs would not be *direct* representations of AOP statements, but would provide a visual front end for describing AOP's cross-cutting effects and a context for machine assistance in 1) producing AOP statements about the cross cutting and 2) associating

conventional code elements that follow from AOP statements with individual components to realize emergent entities. We are already applying UCMs in this way in other contexts (e.g., agent systems [3]) and the approach seems promising. This is not to say the details of how to do it for AOP are obvious at this stage. However, the general idea seems feasible.

Gregor Kiczales speaks of two types of emergent entities described by AOP, data and control. However, data and control are not explicitly represented by UCMs, only causality. Is this a problem? I don't think so, for the following reasons.

Data and control elements are at the level of what I called "wiring" earlier (recall that the term refers to annotations and connecting lines that define control/communication paths between boxes in diagrams of organizational structures, types of quantities flowing between boxes, interface elements of boxes that provide control and access, and so forth). Excluding wiring details from UCMs is a major reason for their usefulness. With this simplification, diagrams of path structures that represent behaviour are at a compatible level of complexity with box structures that show system organization, enabling visual cross-cutting of the two to be shown directly. Otherwise, the visual representation of the behavioural aspect would blow up in complexity, requiring many diagrams to show. Direct visual cross-cutting would be lost. UCMs look useful for AOP precisely because they directly show cross-cutting.

In non-AOP contexts, UCMs may be used to enable humans to associate data or control elements with path segments cutting across components or between them, and there seems no reason why they cannot be used in the same way for AOP. Possibly the demonstration-of-concept UCM editor we have developed could be adapted for this purpose.

1.2 The Rest of This Paper

The rest of this paper summarizes UCM principles and provides examples. It is excerpted from other UCM writings and therefore makes no reference to AOP. However, it will serve to give AOP readers a more solid idea of the nature and application of UCMs.

2 The Essence of Use Case Maps (UCMs)

As shown by Figure 2, UCMs have only three basic elements: *paths*, *components* (shown as boxes), and *responsibilities* (named points along paths that define actions to be performed). Here, the term *components* implies entities that collectively create the behaviour of the system at run time (such as objects or threads); it does not include source code definitional units, such as classes. This use of the term differs from some other

uses; e.g., UML's use of the term includes source code definitional units. Responsibilities are described by active natural language phrases keyed to the labels, e.g "r2: update the data base". Responsibility "dots" (element 1 in Figure 2) visually disappear into the paths, thus avoiding visual clutter by avoiding special symbols for responsibilities. The start and end symbols indicate places—in the environment or internal to the system—where stimuli occur and where their effects stop actively rippling through the system (stimuli may have effects after a path end is reached, due to changes to the system state). As illustrated by the figure, different combinations of the three basic elements are useful at different times for different purposes.

The previously mentioned slots and stubs are simple generalizations of these elements, not new types of elements. Slots are generalizations of boxes that indicate places where dynamic components may appear and disappear. Stubs are generalizations of responsibility "dots".

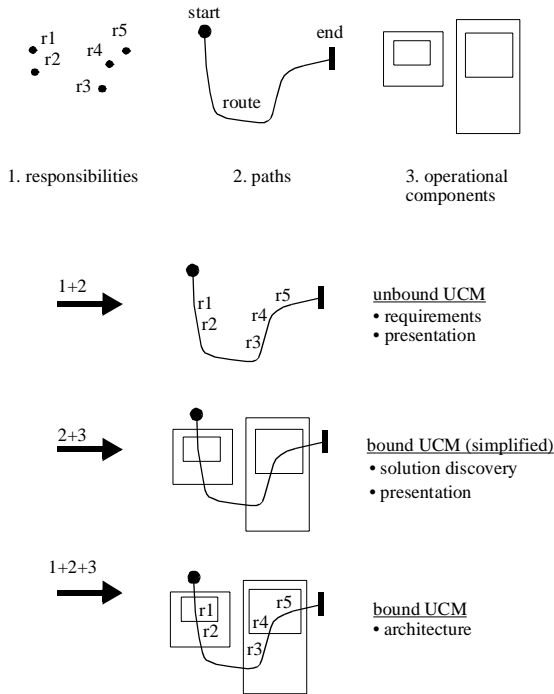


Figure 2 The three independent elements of UCMs.

The UCM term for organizational structures such as the ones in the previous figures is *component substrate*, so we shall use the latter term from now on. The term "substrate" does not imply geographic locality. For example, the component substrate in Figure 1 could as easily represent computers in a nation-wide network as software components in an individual computer (or some combination of both). We say that the paths are *superimposed* on the component substrate. In general,

there is a separate component substrate for each system layer that includes all levels of component decomposition for that layer that are part of the UCM picture (being large scale descriptions, UCMs may defer small scale components to lower levels of abstraction). The component substrate may be shown all at once by showing components as *glass boxes* or revealed in multiple diagrams in which components are black boxes in one diagram and *glass boxes* in another. In the latter case, the substrate is composed—at least conceptually—by overlaying the diagrams (imagine transparency overlays of a diagram of a circuit board).

When UCMs are built up from many paths, different end-to-end routes may end up being partially superimposed, such that they share common segments (Figure 3), giving the appearance of joins and forks. (Only path segments going in the same direction can be superimposed in this way, because superposition implies sharing not just the responsibilities, but also the causal sequences.) Such joins and forks are artifacts of visual superposition. UCMs indicates only the existence of routes and of possible sharing of path segments by routes. If there are only a few routes, the end-to-end identity of each is easy to show visually with different shades or colours of lines (this amounts to labelling the path segments to identify the routes that share the segment, like multiple different highways may be sign-posted on a single city street). A UCM editor we have developed does it by a segment-labelling scheme that may be displayed or hidden, at the user's choice

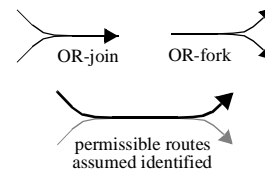


Figure 3 Shared routes.

When reading a UCM, routes are selected according to assumed preconditions and stimuli, and then followed end to end. The components of an actual system must have logic to implement OR-forks and OR-joins, but the logic is at a lower level of abstraction than UCMs.

The helpfulness of UCMs for understanding systems resides largely in people being able to see visual patterns in them that give clues to system organization and behaviour (the term *pattern* is not used here in the specialized technical sense adopted by the object-oriented patterns community). This means that the visual structures must be shaped with considerable care. The arrangements of boxes in the component substrate should not be arbitrary, but should give clues to human

observers about the roles of the boxes, using conventions that will be understood by people who will see the UCMs (e.g., a project team). There is an interdependence between the component substrate and the paths that suggests that the former should be arranged, at least in part, to give the paths helpful shapes. For example, if possible, boxes should be arranged so that paths that go in opposite directions from a functional perspective should go in visually opposite directions (e.g., transmission and reception paths in a computer communication system).

However, much of the usefulness of UCMs comes from the fact that the component substrate and the paths may have a certain degree of independence. Paths may be deformed to accommodate changes to the component substrate and the component substrate may be deformed to accommodate changes to the paths. As long as the deformations preserve the continuity and general relationships of the paths, humans are still able to see the intended meaning.

This way of thinking about UCMs leads naturally to using them for experimenting with different path-structures for the same component substrate (which amounts to designing the behaviour of a given system organization at a high level of abstraction) or different component substrates for the same paths (which amounts to designing a system organization to realize end-to-end behaviour requirements expressed by the paths).

Of course, UCMs must be properly documented, but there is no room here for describing how (see [1][2]).

2.1 Visual Shorthands.

Visual shorthands are useful for some general path constructions (e.g., concurrent forks/joins, interpath coupling, waiting for events along paths, timeouts on waiting, dynamic modifications to organizational or path structures), but these are only shorthands for constructions that can be described with no more notation than Figure 2 and Figure 3. The shorthands will mostly be used without explanation in later examples, but a brief explanation of one of them, asynchronous interpath coupling with waiting and timeout, will give the basic idea. The other shorthands will be identified later without being explained in detail (see [2] for more explanations).

The way to represent interpath coupling using only the notations of Figure 2 and Figure 3 is to introduce a *coupler* component (Figure 4). The nature of the coupling is determined by definitions associated with path elements bound to the coupler component. Because of this binding, coupling implicitly occurs through the internal logic and state of the coupler component (not

specified by UCMs). This component is also implicitly responsible for transferring any quantities that must flow between the coupled paths.

The specific form of coupling shown in Figure 4 is asynchronous with timeout. It is asynchronous because the main path may have to wait but the clearing path never waits. A timeout path originating within the coupler indicates the possibility of spontaneous generation of an activity along a timeout path, after the timer is set along the main path. The timeout path is identified by a zig-zag at the beginning, which is the UCM convention for an error path, because a timeout is considered to be kind of error condition. Timer support would likely be in an underlying layer, so the timeout event would actually arrive from some timer component in an underlying layer, not shown in the component substrate of this UCM. In general, the order of waiting and clearing should not matter. However, this depends on the detailed logic of the coupler component (e.g., on whether or not clearing events are stored). Such matters would be documented as properties of a specific coupler component, assumed as common properties of all coupler components of the same type within a set of UCMs, or deferred as details at the UCM level of abstraction. The visual shorthand replaces the coupler component by symbology that indicates the nature of the coupling at a glance, without having to read textual labels and associated definitions.

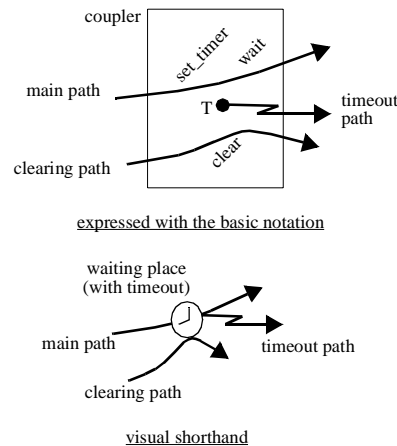


Figure 4 An example of visual shorthand.

3 Examples

Published work on UCMs includes applications to describing object-oriented frameworks, raising the level of abstraction of design patterns, attributing behaviour to system architectures, front ending conventional design models of the UML-ROOM variety, and designing/understanding dynamic agent systems.

However, this is only the tip of the iceberg. The author is personally aware of a large number of unpublished applications by students, collaborators, and readers of UCM publications. Here is a very incomplete list of them: understanding and describing WIN (Wireless Intelligent Network) and ATM (Asynchronous Transfer Mode) standards and their realization; controlling multimedia conferences; designing a distributed banking application; designing control software for a utility company; understanding a complex object-oriented framework for telephony applications; finding race conditions in telephone billing systems; finding race conditions in transparent intelligent network (TIN) applications; 911 call processing scenario analysis; and the list could go on.

This section first summarizes a few published examples to get the ball rolling. Then it presents some previously unpublished examples that illustrate how UCMs may be used for systems of industrial scale and complexity. The latter examples come from former students now working in or with industry and are not fully developed for a variety of reasons: commercial confidentiality is an issue, not enough information was available from the students, or too much explanation would be required for a short paper. In spite of this incompleteness, readers should be able to get enough of a sense of the examples from the UCMs to understand how such UCMs could be useful in practice.

All of the examples are presented in much the same terse style as Figure 1 (the second style of Figure 2). In particular, responsibilities are often left off paths because their existence and nature may be inferred from the components traversed by the paths. Several shorthand notations are used in addition to the one explained in Section 2.1. The additional notations are identified in cross-hatched boxes at the top of the first figure that uses them.

3.1 Transactions

Figure 5 and Figure 6 make Section 1's concept of a "behaviour structure" for a transaction concrete. A simple network transaction may be represented by a path winding its way through a network from some starting point, eventually returning to indicate transaction completion. This UCM depicts the situation when no errors occur along the way.

The meaning of the component stacks is that many concurrent but independent transactions may be in progress at once, involving many client and servers. The notation saves visually replicating paths that are the same for multiple components. The implied concurrency could be implemented in two different ways, and the same UCM covers both (a powerful feature of UCMs). One way is to have Client and Server *processes*

achieve transaction concurrency by explicitly interleaving sequences for different client and server *objects*. The other way is to have client and server *threads* handle the different transactions independently. The difference lies in the nature of the components bound to the paths, not in the paths themselves. UCMs defer the decision while providing a context for thinking about it.

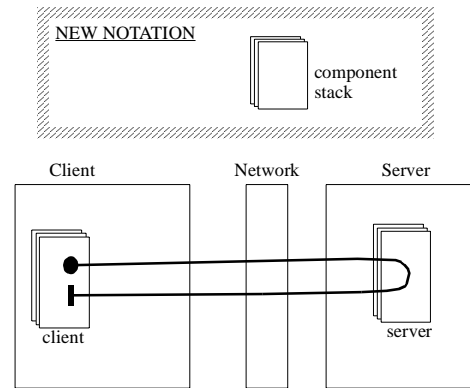


Figure 5 A simple network transaction.

Remote errors may occur that require the "behaviour structure" for the simple transaction to be somewhat more complex than Figure 5. Figure 6 shows a generalized structure, using a path *stub* to identify the place where generalization occurs. The *failure points* shown in this diagram assume that, from the perspective of the client, all remote failures amount to failures between the client and the network in either direction. A stub identifies a place where path details are deferred to a sub-UCM, called a *plug-in*.

Two alternative plug-ins are identified for this particular stub, plug-in 1 with pass-through paths, and plug-in 2 with additional failure handling path structures. The stub is fixed (a dynamic stub would be indicated by a dashed outline), so dynamic selection among the plug-ins is not implied (the plug-ins simply indicate alternative static path decompositions). There is no special notation for identifying plug-ins or the stubs with which they are associated. In practice they would be in separate UCMs, with relationships identified by labeling. Here, all are shown in the same diagram, so relationships are conveniently shown by light lines (not part of the UCM notation).

Replacing the stub with plug-in 1 produces the original UCM of Figure 5. Replacing the stub with plug-in 2 produces a generalized version of this UCM. Plug-in 2 supports forwarding of transactions (*a* to *b*), normal completion of transactions (*c* to *d*, cancelling the timer in passing), and timeout when notice of completion is not received within a reasonable time (completion of the *a-d* path by spontaneous exit from the

waiting place, after timeout). The *AND-fork* indicates that transaction forwarding and waiting for completion are concurrent actions. The normal transaction path and the timeout path join before *d*, so the observed result at *d*

will indicate one or the other (information is assumed to flow to the client from *d* for this purpose). By definition, the proxy object has control of all paths traversing it, and thus isolates local objects from network issues.

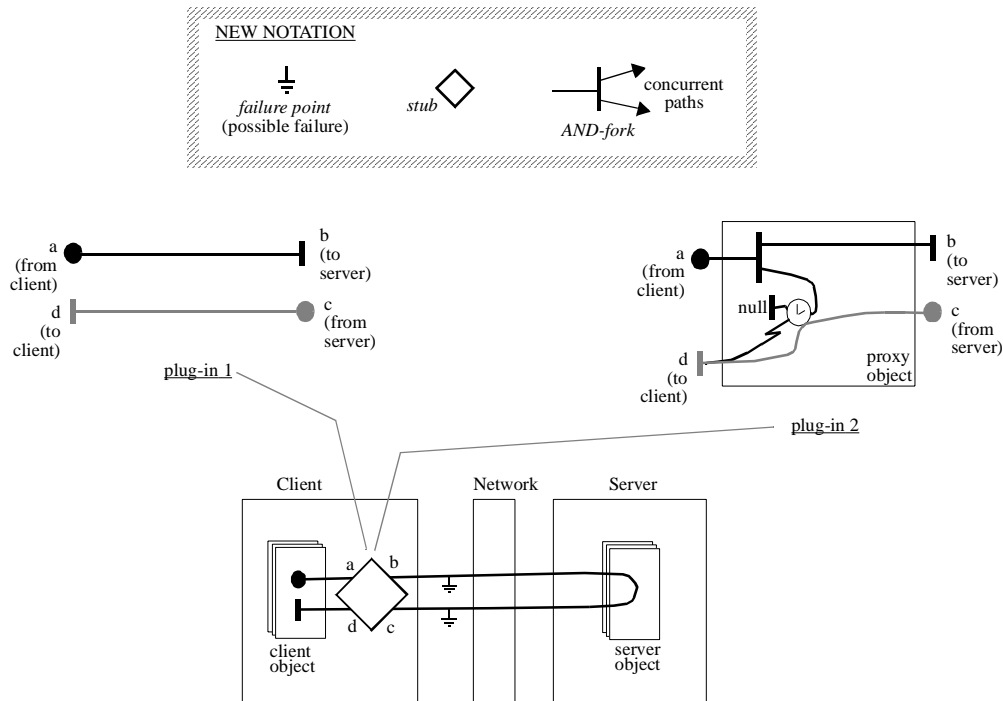


Figure 6 A generalized network transaction.

3.2 A System Dynamically Modifies its Own Components

This example illustrates how UCMs express situations in which a system modifies its own components. The example is downloading device handler objects to customize device driver processes for newly connected devices (Figure 7). The sequence starting from A is triggered by the shipping of new devices to customers. The sequence starting from B is triggered by connecting a new device. The diagram should be relatively self explanatory.

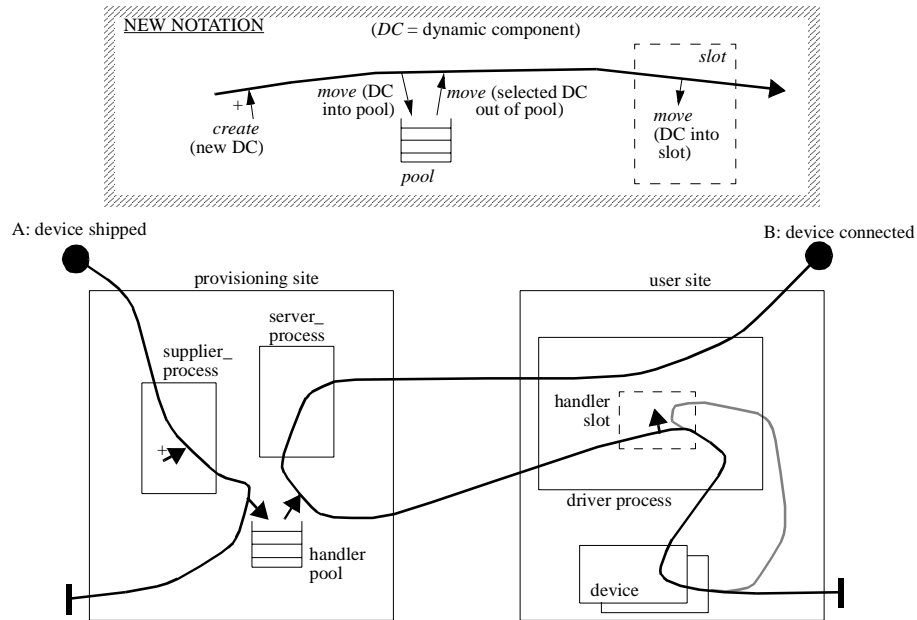


Figure 7 Downloading device handlers.

3.3 A System Dynamically Modifies its Own Path Structures

This example illustrates how UCMs can be used express large scale dynamic situations in which a system dynamically modifies its own path structures. The example is telephony feature interaction in a network environment in which software agents handle telephone calls on behalf of human users. Feature interaction belongs to a very general class of system problems, in which errors at the system level are created by inappropriate combinations of dynamically determined details in different parts of the system.

In Figure 8, telephony features are represented by plug-ins, the dynamic selection of features is represented by the dynamic selection of plug-ins for stubs, and feature interaction appears as incorrect end-to-end routes resulting from particular combinations of plug-ins. Such a diagram clearly expresses the concept that the system must dynamically select from among competing feature plug-ins for each stub and makes it easy to understand the system implications of different selections, without specifying details of either the features or their selection. The figure shows, at its center, how a call request can take us through a set of software agents to a phone ringing at some remote user location. The CSP (Call Side Processing) and ASP (Answer Side Processing) stubs have dynamically selected plug-ins for different features (both stubs are shown in both agents to show that the situation is symmetrical, in principle, although only one direction is shown). At the top

of the figure is shown a set of default plug-ins that cause no problems. At the bottom of the figure is shown a set of plug-ins that, when selected in this particular combination, may cause a feature interaction.

The way to read this diagram is to trace a normal call through the top set of plug-ins and to trace a forwarded call through the bottom set. One of the plug-ins is the same in both sets (the one on the right), but is duplicated so the diagram can be read this way.

The main UCM at the center of this diagram describes a route that winds through a set of Answer-Side agents, due to a diversion via the grey path from point *c* on the ASP stub. The implication is that the diversion causes the route to continue “underneath” for a different agent (a diversion from *c* would not go back into the same agent, by definition, so what follows would have to involve a different agent).

Feature interaction results from selecting the Originating Call Screening (OCS) plug-in for the call side and the Call Forwarding (CF) plug-in for the answer side. With this combination, a caller may call some number not on the OCS list and be forwarded to a number on the OCS list. The design defect that results in the feature interaction can also be immediately spotted: CF does not consult the caller’s OCS list. One way of removing the defect (not shown) is to route the forwarding path back through the calling agent to check if the number is forbidden.

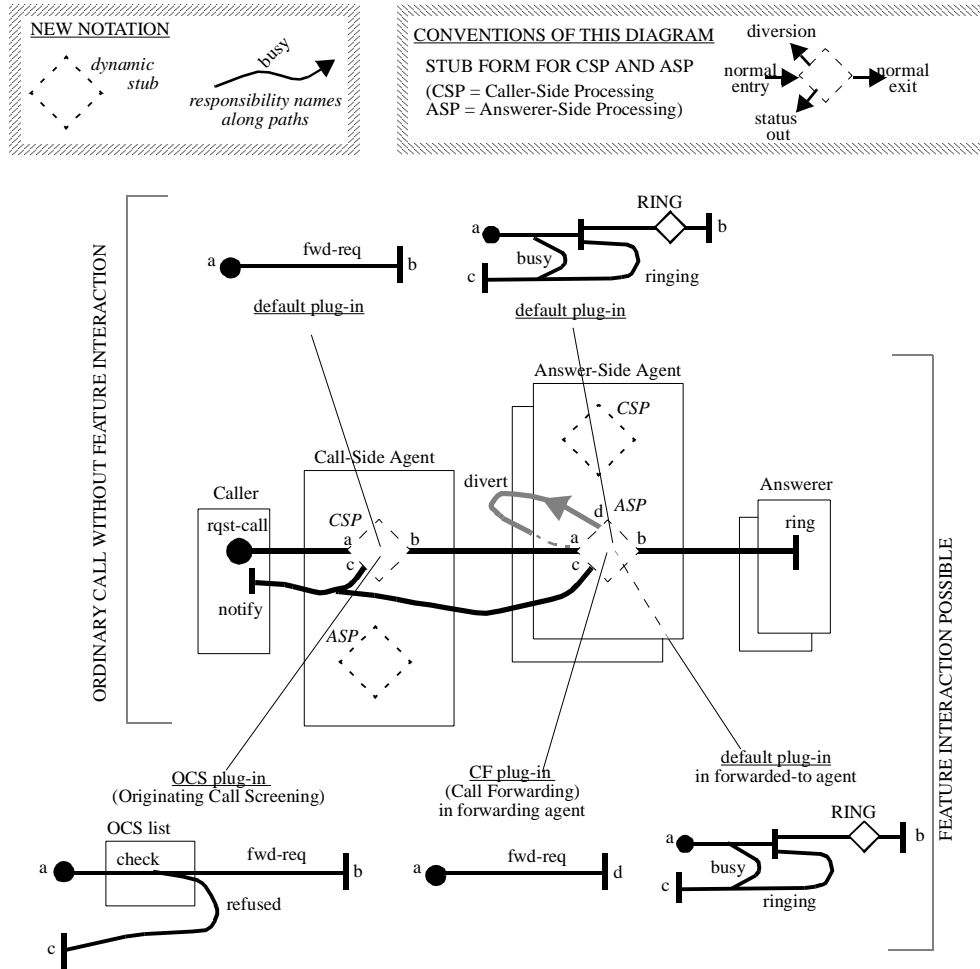


Figure 8 A telephony feature-interaction example.

3.4 ATM Call Control

A student now working in a large telecom company provided me with Figure 9 as an example of a practical industrial application of UCMs to ATM (Asynchronous Transfer Mode) call control. He also included the following observations about why and how such UCMs are useful (the observations refer to UCM1 and UCM2, but I have included only UCM2, which is essentially the same as UCM1 except with more explicit path concurrency):

- "Depict ATM Call Control scenarios as they actually exist in reality."
- "Relevant, of significant complexity, yet still high-level as it is based on the standards (nothing proprietary)."
- "Things are really complicated."
- "You could zoom-in several times and still be at a high-level view. But then they would start to show proprietary ways of how to do things. There are also more scenarios."

- "I use them to study the implications of parallel processing. The Call Router in UCM1 is a bottleneck, so UCM2 identifies the activities that can be done concurrently. There are all kinds of engineering aspects: dimensioning of message queues, memory and CPU power, estimating the worst-case, throttling etc. The dynamic routing system is not detailed. It could be anything, for example PNNI, which is currently a hot routing topic (standard) in ATM."

Note that Figure 9 contains some idiosyncratic notation. The concatenated end-start points along paths between boxes are unnecessary, but were used to provide interface points that could be named. The construction in two places near the bottom of the figure in which a start point touches another path tangentially amounts to an AND fork.

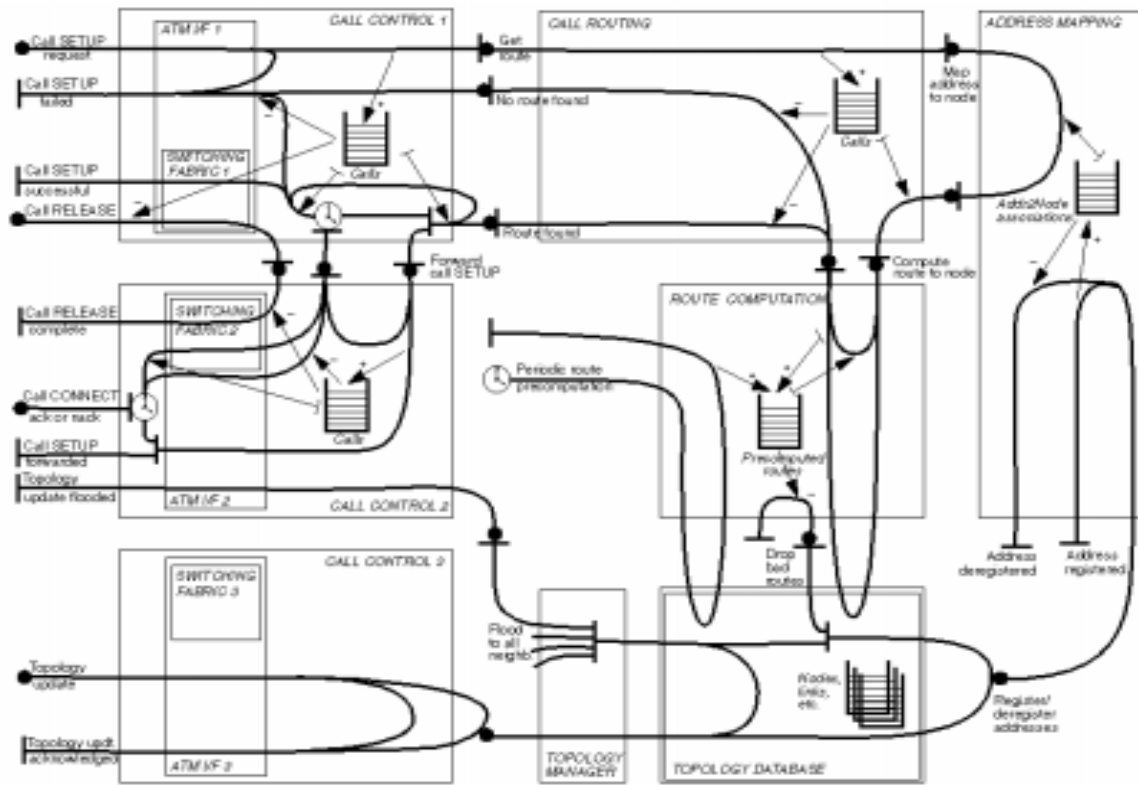


Figure 9 UCM2: ATM call control with dynamic routing.

3.5 WIN Scenarios

A student abstracted Figure 10 and Figure 11 from a large WIN (Wireless Intelligent Network) standardization document. These and a few additional diagrams

condense the essence of the behaviour implied by the WIN standard into a small set of understandable diagrams.

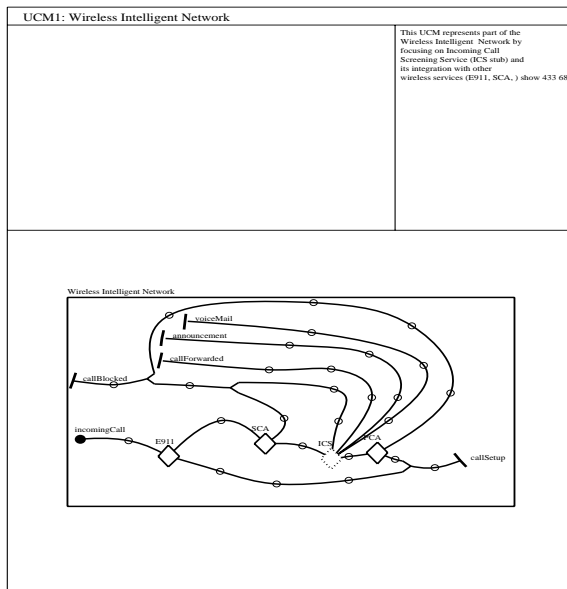


Figure 10 A WIN UCM.

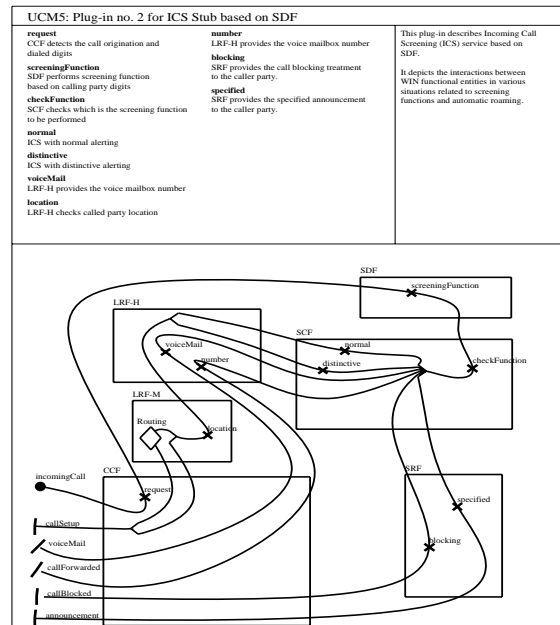


Figure 11 A WIN plug-in.

The purpose here is not to explain WIN or these diagrams, but simply to give a sense of the kind of diagrams that prove to be useful in practice. Unlike all the earlier diagrams of this paper, which were drawn with ordinary drawing tools, these two diagrams were drawn with a UCM editor we have developed. Some display artifacts are visible that may be hidden, if desired: circles identify handles for reshaping paths; crosses identify responsibility points. See Figure 12 for another example of the use of this GUI.

This UCM editor provides a requirements/design front end for research projects on agent system design/prototyping and on performance evaluation of system architectures, among other things. Diagrams like Figure 10 and Figure 11 are currently being used as a starting point for studying the application of WIN standards in industry and as a basis for performance evaluation studies of WIN architectures (see Section 3.6 for a different example of the use of the UCM editor as a performance evaluation front end).

3.6 Performance of Architectures

We have been experimenting with using our UCM editor as a front end for performance analysis, and the approach looks promising. Figure 12 gives a sense of the approach. The GUI is used for such purposes as

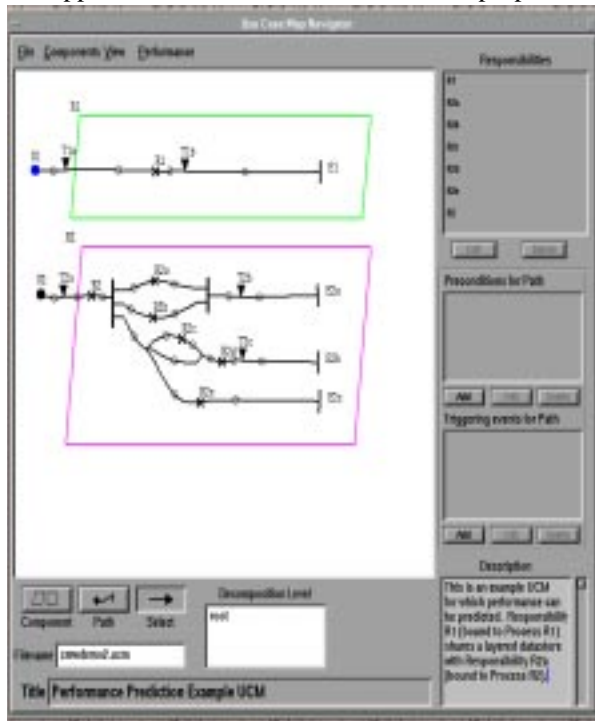


Figure 12 A GUI for performance evaluation.

marking timestamp points (the labelled inverted triangles in the figure), assigning frequencies to path traversals, associating boxes and path segments with physical

elements such as processors, shared data stores, and communication links, and associating performance parameters with physical elements. The result is then fed into a tool that generates performance models and results. This particular diagram only illustrates the nature of the GUI and its support for timestamp points. The idea eventually is to feed the performance results back through the GUI.

In this particular diagram, the boxes do not have rectangular shapes. This illustrates a feature of the UCM editor, which is the ability to distinguish different types of components by different shapes. For example, parallelograms represent concurrent processes.

4 Conclusions

This paper has aimed, not to present UCMs for the first time as a new concept, but to draw the attention of people in the AOP field to a notation that may be ready made as a design notation for AOP. The possible relationships to AOP were summarized in Section 1.1 and that section represents the kernel of the workshop contribution. The rest of the paper describes principles and examples of UCMS, to enable AOP readers to get a sense of their nature and status.

Acknowledgments

Research into UCMs and their applications is currently supported by TRIO (now CITO), NSERC, Mitel, and Nortel. Many students, coworkers, and collaborators have contributed to the development of these ideas over the long term, but are not mentioned here because they are either coauthors of other UCM publications or acknowledged in them. I am grateful for example diagrams contributed by Rossanna Andrade, Andrew Miga, Craig Scratchley, and Adrian Soncodi. Daniel Amyot also provided valuable assistance and input.

References

(see the referenced UCM papers for more complete lists of UCM and related publications)

- [1] R. J. A. Buhr, R. S. Casselman, *Use Case Maps for Object-Oriented Systems*, book, Prentice Hall, 1996
- [2] R.J.A. Buhr, *Scenario-Path Signatures as Architectural Entities for Complex Systems*
www.sce.carleton.ca/ftp/pub/UseCaseMaps/ucmUpdate.ps (or .pdf)
- [3] R.J.A. Buhr, D. Amyot, D. Quesnel, T. Gray, S. Mankovski, *High Level, Multi-Agent Prototypes from a Scenario-Path Notation: A Feature Interaction Example*, Proc. PAAM98, London, April 98
www.sce.carleton.ca/ftp/pub/UseCaseMaps/4paam98.ps (or .pdf)
- [4] Shaw and Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996