Making Behaviour a Concrete Architectural Concept

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. *A practical approach is presented to making behaviour a concrete, first-class architectural concept. The approach overcomes the forest-tree problem that results when the only way of understanding behaviour in relation to the organizational aspect of architecture is in terms of sequences of inter-component interactions that emerge at run time (calls, messages, etc). The approach centers around diagrams called Use Case Maps (UCMs) that superimpose sets of continuous wiggly lines (representing signatures of causal sequences) onto arrangements of boxes (representing organizational structure). A powerful feature of the approach is its ability to express large scale dynamic situations clearly. This paper does not present UCMs for the first time, but provides new insight into their essence in relation to architectural issues, alerts workers in the field of software architecture who have not encountered them before to their possibilities, and introduces for the first time a demonstration-of-concept tool to support them.*

1 Introduction

Architecture is concerned in part with *organizational 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., [6], to see many diagrams of this type.

Behaviour is an important element of architecture, but is difficult to pin down as a whole-system property if the only understanding we have of behaviour is in terms of the kinds of details associated with wiring. Then, behavioural properties of architectures, such as performance, robustness, and the ability to satisfy use case requirements, can have meaning only in terms of implicit or explicit assumptions about such details. However, we should be able to associate behavioural properties with architectures without having to resort to assumptions about details. This requires understanding *in the large* how organizational structures and behaviour of whole systems are interrelated. For this purpose, we need a concept of macroscopic *behaviour structures*, complementary to that of organizational structures, and not dependent on details associated with wiring.

On one hand, it can be difficult for the uninitiated to imagine what a macroscopic "behaviour structure" might be. 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 the desired "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 in a complex system, such as opening a WWW page, may involve the joint efforts of many elements of the 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 WWW page is easily visualized as a causal path cutting across the system from local entry of a URL, through a network of computers and software to some server, to a data base at the server, and back to the local computer.

The visual notation about to be introduced gives concrete form to this kind of mental picture by imposing sets of wiggly lines representing causal paths onto diagrams of organizational structure (Figure 1). The notation is called "Use Case Maps" (UCMs) because its wiggly lines may be viewed as mapping scenarios of use cases onto system organizations to show how the use cases are realized. However, the term *scenario* is intended to mean something more abstract here than UML's *sequence of interactions*, which express sequences in terms of wiring-level details. With UCMs, scenarios are seen in terms of continuous paths directly superimposed on organizational structures and their sequences are of responsibilities performed by components of the system, not of interactions between components. The paths are *signatures* of scenarios that may be used by a human observer to visualize scenarios pro-

gressing along them. Compositions of paths (sets of wiggly lines) are macroscopic behaviour structures that are useful for architectural purposes.

UCMs are powerful for expressing and understanding important macroscopic aspects of behaviour in relation to architecture that are very difficult to deal with at the level of wiring details (Section 3 provides examples):

- Path interactions in concurrent systems are visible at a glance.
- Performance becomes a property of paths, rather than a non-functional property of a whole system, as it is usually considered to be.
- Large scale dynamic situations can be made visible at a glance.
- Paths directly indicate how the architecture satisfies use case requirements.

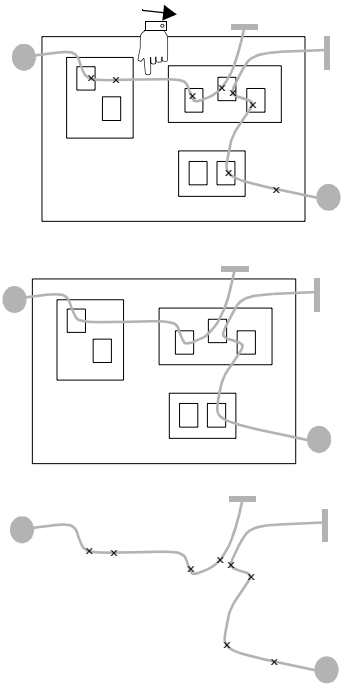
The approach has been developed over many years through many publications but many people who might benefit from it have still not encountered it. It has application in many different areas whose working communities do not necessarily overlap and so a recent series of publications with partially overlapping content has attempted to present the ideas to different communities (e.g., [1][2][3][5]). This paper is part of that series. It aims to alert workers in the field of software architecture to the possibility that UCMs can resolve some difficulties in that field and to point to where they can find more detailed information.

For those who have encountered the notation before, this paper still offers something new: 1) A clear identification of the core of the notation is provided. As the notation has been developed and applied to new problems, we have come to realize that earlier publications, such as [5], may have made the notation appear too complex by presenting some notational forms as fundamental that are either visual shorthands for constructs that can be expressed with a much simpler notational core (for example, path coupling with timeout), or embellishments that are distinct from the core. Identifying the core of the notation is important in for the purposes of this paper because it is the core that provides architectural expressiveness (a more comprehensive archival paper [1], which aims to be the last word on the UCM notation, contains more in this direction but is not yet in print as this is being written). 2) A glimpse is provided for the first time of a demonstration-of-concept UCM tool we have developed [4] that is already being evaluated in several places in industry and that we expect can be made more widely available by the time this paper is presented.

2 The Essence of UCMs

To place UCMs in context, [1] identifies three dimensions of description of large, complex systems: *abstraction*, *decomposition*, and *layering*. In this view, abstraction is not just deferring detail, but making a paradigm shift in the method of description (e.g., from code to UML, or from UML behavior descriptions to UCMs); and decomposition and layering are not abstraction techniques, but organizing techniques at a particular level of abstraction (i.e., within a particular paradigm). In general, a large, complex system may have many descriptions identified with many points in the associated 3-dimensional space. The reason for mentioning all this here is to emphasize that UCMs do not aim to cover this space, but only to provide some points in it, complementing the descriptions provided by other techniques such as UML. The UCM descriptions are at a higher level of abstraction but do not eliminate the need for descriptions at lower levels of abstraction.

As shown by Figure 1, the core notation, is very simple. The notation has only three fundamental ele-



(a) Imagine tracing a path on top of a diagram of the structure of a system to explain a scenario as a causal sequence of events and actions, leaving behind a wiggly line to be preserved as a visual record of the path.

(b) Paths touch components at points called responsibilities, shown as x's in (a), but UCMs are usually displayed with responsibility-points shrunk to invisible dots, as here.

(c) Unbound maps (without components shown explicitly) may be useful as requirements entities or as stand-alone "behavior structures."

Figure 1 Core UCM Notation.

ments: *Scenario paths* are represented by wiggly lines; rectangular boxes represent runtime *components*; *responsibility-points* along paths touch components to indicate that components have responsibilities along paths (the order of touching along a path expresses a causal sequence). The start (filled circle) and end (bar) symbols for paths indicate places—in the environment or internal to the system—where stimuli occur and the

effects of stimuli stop actively rippling through the system. Paths, components, and responsibilities all have labels, not shown. To make diagrams as uncluttered as possible, responsibility points are usually identified in UCMs *only* by their labels.

Paths, component boxes, and responsibility points are all formal elements of the notation. Additional information may be associated with these elements for human use, but this information is not formal. For example, a component may be identified as storing data items, and responsibilities may be identified along paths to operate on these data items, but there is no formal representation of the data items and no formal way of specifying how responsibilities use or change them. This lack of formality in the details is what makes the notation lightweight.

Useful incomplete UCMs may be created by combining any pair of the three fundamental elements. Figure 1(b) combines paths and components, without responsibilities. This kind of diagram is useful for back-of-the-envelope-style sketching of ideas and for presenting an overview of the big picture. Figure 1(c) combines paths and responsibilities, without components. Diagrams of this kind (called *unbound* maps) can be useful as requirements entities that provide a transition from prose use cases to UCMs. They may also be useful as stand-alone behavior structures that may be reasoned about or saved for reuse. Not shown because it is not particularly useful as a diagram (although the information may be useful) is the combination of components and responsibilities, without paths.

The UCM term for arrangements of boxes, such as in Figure 1, is *component substrate*, so we shall use this term from now on instead of *organizational structure*. 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). For large scale systems, UCMs may be both decomposed and layered. A single component substrate identifies a system layer. Separate system layers have separate component substrates. Decompositions of a single substrate may be shown either all at once, by showing components as *glass boxes* (as in Figure 1), 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, but do not confuse this concept with system layering, which requires separate substrates that should not be visualized as transparency overlays).

Composite UCMs may be built up from many paths. When this is done, different paths may end up being partially superimposed on each other, such that they share common segments, as shown in Figure 2. This creates joins and forks in the map (called *OR-joins* and *OR-forks*). OR-joins/forks are artifacts of visual superposition; they have no implication for synchronizing scenarios along them. Scenarios proceed independently through OR-joins/forks. OR-joins/forks neither increase nor decrease whatever level of concurrency is intended to be present in a map.

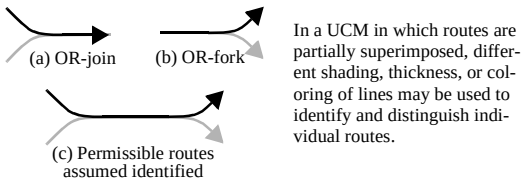


Figure 2 Shared routes and OR forks/joins.

OR-forks/joins may introduce apparent visual ambiguity that may be eliminated by highlighting through-routes, as suggested by Figure 2. This figure shows one of the many ways in which highlighting of lines in UCMs may be useful (highlighting may be with different line shadings, as here, or with different line thicknesses or colors). The meaning of line highlighting is not standardized, but is diagram-dependent. The particular highlighting in Figure 2 is intended to make the diagram unambiguous by indicating that only two routes are possible through the shared path segment, a route highlighted in black and another one highlighted in gray (visualize the black route as overlaying the gray one along the shared segment). This means there is no crossover from the black route to the gray one, or vice versa.

The concept of routes is very important for composite UCMs, whether the routes are identified by visual line highlighting or by other means. A composite UCM in which some path segments are shared must always be understood as an overlay of different routes. Tracing a scenario through a UCM requires picking a route and following it. *The forks and joins along the way have no decision logic associated with them at the UCM level of abstraction to determine which way to go.*

In some practical situations, attempts to identify all possible routes may be defeated by combinatorial explosion (e.g., if routes double back on themselves to retry a segment after an error). In such cases, one must be satisfied with identifying main route segments, leaving it to the UCM observer to visualize combinations. This is a consequence of the deliberate lack of commitment to detail in UCMs and should be regarded as a positive feature of UCMs, rather than a defect.

Some labeling conventions for start/end points, responsibilities, and route segments are indicated by Figure 3. Routes may be identified for reference purposes by paired labels of start and end points (e.g., route

AB, route CD). In more complicated cases (such as were identified in the previous paragraph), variations on a route may exist between the start and end points, due to the possibility of different combinations of route segments between these points. In such cases, it may not be practical to identify all route variations by labels.

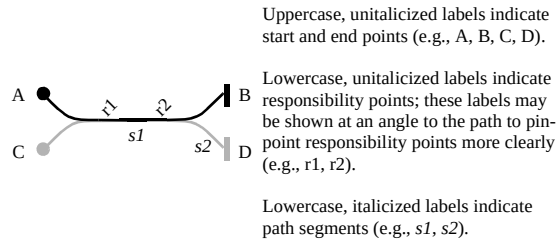


Figure 3 Labeling conventions.

Informal information may be associated with any label (e.g., for Figure 3: *a precondition of CD is that AB has been traversed at least once*; *r1 changes the system state; r2 reads the system state*). The purpose is only to provide information to the person reading the UCM. There is no implied underlying relationship between such descriptions, other than that provided by the words themselves.

Paths may be concatenated as shown by Figure 4 (and also broken into parts by the reverse process).

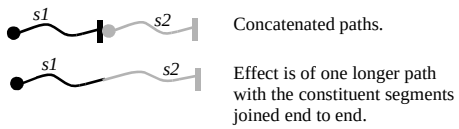


Figure 4 Concatenated paths.

There is another type of UCM fork called an AND-fork (Figure 5). The notation for AND forks is a generalization of the concatenation notation (the outgoing paths are jointly concatenated to the end of the incoming path). AND-forks split a single scenario into parts that may proceed independently (and, if concurrency is allowed, concurrently). AND-forks are complemented by AND-joins (Figure 5); these indicate a strong form of interscenario synchronization in which scenarios along different paths are mutually synchronized.

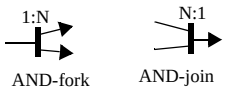


Figure 5 AND-forks/joins.

For concurrent situations, AND-forks/joins change the level of concurrency (AND-forks increase it, AND-joins decrease it). However, the notation is concurrency-neutral because the forks and joins have meaning even when there is no concurrency, namely that sequences are independent.

Annotations on the fork/join bars of the form N:M indicate the number of independent scenarios leaving a fork or being synchronized at a join (in general, this could be different from the number of paths entering or leaving because of the possibility of many scenarios proceeding along a single path).

Some variations on the basic concept of AND-forks and joins are useful. Figure 6 summarizes the important variations, which will now be explained. The fork-join shorthand is just a concatenation of a fork followed by a join, to indicate a temporary split of a scenario into independent parts that are then resynchronized. The rendezvous shorthand performs the concatenation the other way round (a join followed by a fork), to indicate scenarios coming temporarily together. The synchronize shorthand indicates a point rendezvous (the shared rendezvous path segment has shrunk to zero length). The other shorthands indicate situations where multiple independent scenarios may proceed along the same path.

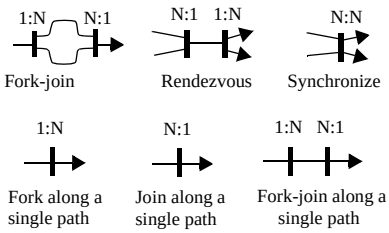


Figure 6 Variations on AND-forks/joins

In summary, the core notation contains only wiggly lines, responsibility points along lines, component boxes touched by the wiggly lines at responsibility points, and some special notations for start points, end points, forks, and joins. Elements of UCMs may be highlighted for special purposes by shading, coloring, or thickening lines, but there is no standard meaning for such highlighting. Elements are labeled and informal prose descriptions may be associated with the labels but there is nothing underlying the visual notation to link such descriptions other than the words themselves. Everything that follows is built up from these elements. Additional notations will be introduced as we go along but these are mostly no more than visual shorthands for constructions that can be shown with the core elements.

How can such a lightweight notation say anything important about system behavior? The answer is: Much of it is in the eye and mind of the beholder, guided by heuristics which are illustrated by the examples of Section 3 and explained in more depth in [1].

3 Examples and Additional Notation

Two examples adapted from [1] will now be presented to illustrate some of the issues and notational principles

covered up to this point. The examples are necessarily small scale. There is never sufficient space in a paper or book chapter to present complete, large-scale examples and, even if there was, very few readers would be interested in seeing the details. Examples such as the ones about to be presented are abstracted from actual systems but the full picture of the systems is not presented. Therefore some imagination must be exercised when studying the examples. Imagine scaling up the system and the coverage of it, such that inch-thick stacks of conventional diagrams would be required, because this is representative of large, complex systems in practice. Also remember that many details are deferred by UCMs, so they do not necessarily directly reflect the scale of a system.

Although, in principle, no new notation is required to present these examples, visual shorthands are useful for some path constructions that amount to UCM boilerplate. These shorthands make diagrams less cluttered and more understandable at a glance (once you are used to them). Readers who are encountering UCMs for the first time may feel that they complicate the notation, but experienced users find them helpful. The shorthands will be introduced as the examples are developed (boxes at the tops of figures identify any new notation in the figures).

The examples presented here focus on the use of UCMs to show behavior structures of system-wide transactions and the self modification of such behavior structures while a system is running. See [1] for more complete treatments and for other examples, including self modification of component organizations while a system is running.

3.1 Behavior Structures for Network Transactions

Figure 7 makes Section 1's concept of a behavior structure for a network 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 (the assumption that no errors occur will be relaxed as we develop this example).

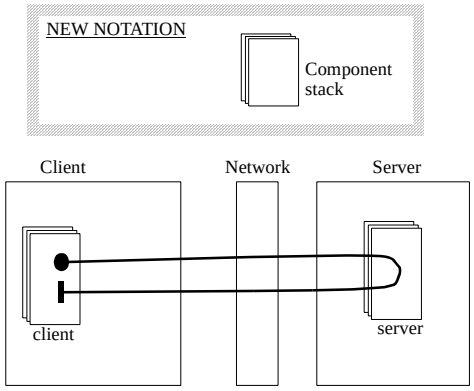


Figure 7 A simple network transaction.

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 avoids visually replicating paths that are the same for multiple components. The nature of this particular application problem makes clear, without additional notation, that scenarios along independent transaction paths may be concurrent.

Transaction-path concurrency could be implemented in two different ways, and the same UCM covers both. 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 in the context of a single process that manages all transactions in a network node. The difference lies in the nature of the components bound to the paths, not in the paths themselves. This diagram has not yet made a commitment about the nature of the components.

Figure 8 adds more "behavior structure" to provide for transaction completion in case of network failure.

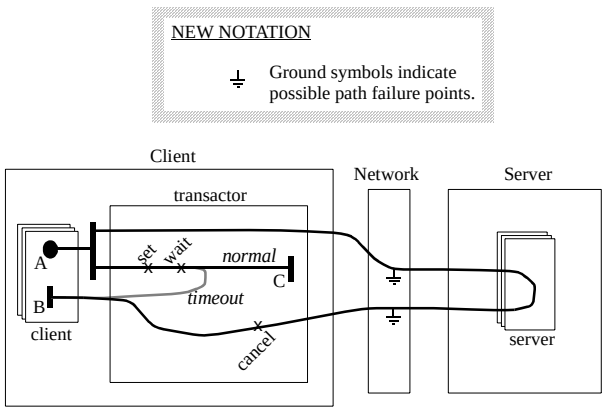


Figure 8 A generalized network transaction.

Possible failure points are shown by ground symbols borrowed from electrical engineering. To make the diagram easier to read, normal paths are shown in black and abnormal ones in gray (this is just a convention for this example, not a general one for the notation). There are two alternate routes from A to B, one as a result of a timeout and the other as a result of normal transaction completion. The AND-fork provides a local path to `set` a timer and to `wait` for cancellation of the waiting condition through either normal transaction completion or timeout. Normal transaction completion triggers a `cancel` responsibility that cancels the waiting condition and the timer. The A to C path is intended to have a null effect (it just terminates the AND-fork in the case of no timeout). The `transactor` component groups the `set`, `wait`, and `cancel` responsibilities, to make clear that they are related; prose documentation of the responsibilities would be sufficient to do this, but visual grouping helps.

While such a diagram can be made clear by appropriate labeling and prose documentation, the timeout-recovery mechanism within the `transactor` may be usefully regarded as UCM boilerplate that could have the same meaning in many contexts. A simple visual shorthand is supplied in Figure 9 to indicate the nature of this particular mechanism (asynchronous interpath coupling with timeout) at a glance, without textual labels and associated responsibility definitions; the notation also reduces textual clutter.

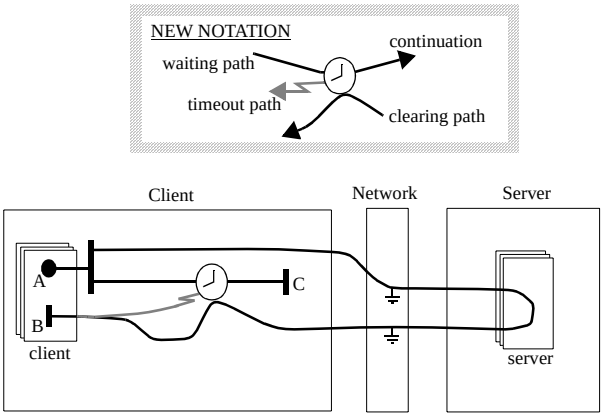


Figure 9 A generalized network transaction expressed with a visual shorthand.

The clock symbolizes a timed waiting place along a path. The implication is that the entering path may wait here for a canceling event and then either continue normally or follow a timeout path. In this example, the normal path simply ends without further action (no timeout was needed so there is nothing more to be done). The event that cancels the waiting condition comes asynchronously from either a timer (implicitly set by the path that enters the timed waiting place) or a separate clearing path. The clearing path touches the timed waiting place tangentially, symbolizing asynchronous coupling. The zig-zag visually distinguishes the timeout path from the continuation path (in general, a zig-zag after the start of UCM path indicates a path triggered by some exceptional condition, of which a timeout is a special case). The notation assumes that any quantities that must flow along or between paths through the timed waiting place will flow, without indicating how. The notation is not intended to cover all nuances of asynchronous coupling with timeout (for example, whether multiple `clear` events are queued or not remains undefined and must be covered by prose documentation).

Figure 10 gives additional insight into how to read Figure 9 by displaying examples of token traces that should be imagined while reading it (actual diagrams like these are not needed because the original UCM implicitly contains all the possibilities in a way that a person can separate in the mind's eye).

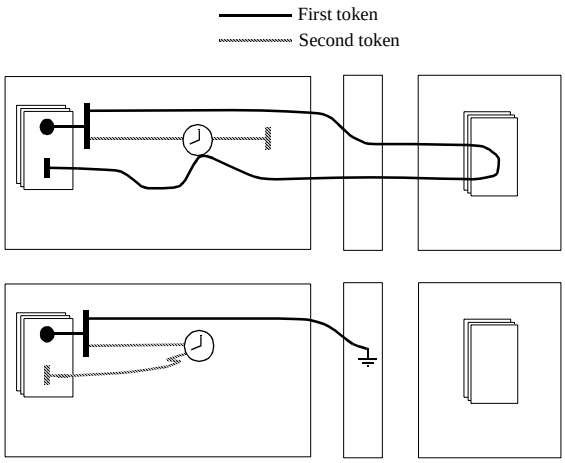


Figure 10 Different token traces that should be imagined while reading the previous UCM.

Figure 11 continues this example by showing how a *stub* notation may be used to defer details. 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 the particular stub in Figure 11, plug-in 1 with pass-through paths (which reduces Figure 11 to Figure 10), and plug-in 2 with additional failure handling capability (which reduces Figure 11 to Figure 9). The stub is static (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 diagrams, 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). By definition, the proxy object has control of all paths traversing it, and thus isolates local objects from network issues.

There is no free lunch. The end-to-end sweep of paths through the system as a whole that is a major strength of UCMs is partially lost with stubs. Experience has indicated that judicious use of stubs and plug-ins can simplify understanding of complex situations but that great care must be taken to keep the big picture as intact as possible. To be avoided is forcing the viewer to study and understand all the details of many plug-ins for many stubs before any sense can be made of the big picture.

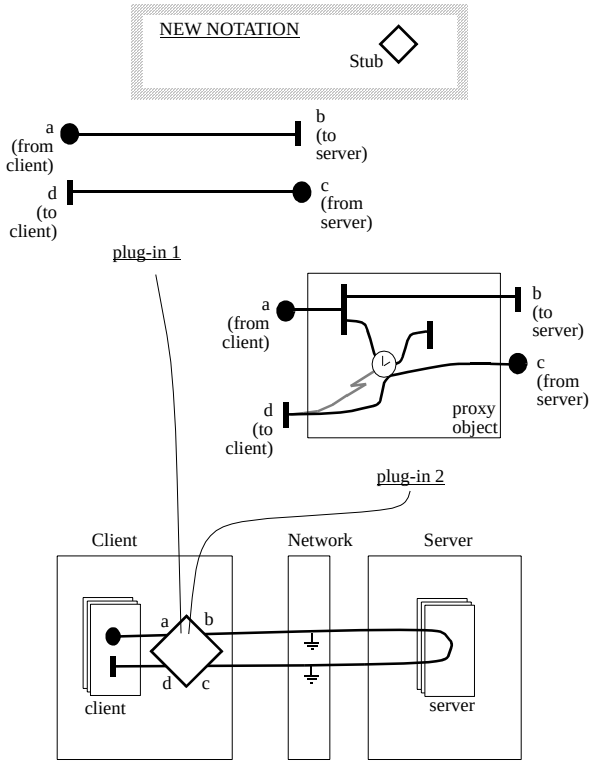


Figure 11 Deferring details with stubs and plug-ins.

3.2 A System Modifies its Own Transaction Structures

This example (Figure 12) is telephony feature interaction in a network environment in which software agents handle telephone calls on behalf of human users. It illustrates the use of UCMs to represent the kind of system self modification in which the makeup of transactions (telephone calls) changes dynamically according to system circumstances (different users subscribing to different features). In this example, 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 that may be traced through particular combinations of plug-ins. An important property of this way of representing dynamic situations is that the big picture and the modifying details are all represented in the same terms (paths). This makes it easier for a person to grasp the big picture than if different terms were used, as they often would be with other notations.

The only new notation relative to notations presented so far in this paper is dashed outlines for *dynamic stubs*. The stub/plug-in ideas are the same as in Section 3.1 except that the new stub notation implies that plug-ins are selected dynamically when a scenario arrives at the stub 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).

There are basically two forward routes through the central UCM, depending on which features are selected. One of the plug-ins is duplicated at the top and the bottom of the figure (the right-hand one) so that all the plug-ins for each route are grouped together at either the top or bottom of the figure.

A default route is followed when the default features at the top are selected. This route proceeds directly from a caller through a pair of software agents to an answerer and is free of feature interaction.

A forwarding route is followed when the features at the bottom are selected. This route is the same as the default route initially but then follows a diversion path to a forwarded-to agent and on to its associated answerer. The implication of the diversion path segment is that the route continues for a different agent in the stack (a diversion from *c* would not go back to the same agent, by definition). The forwarding route may exhibit feature interaction.

The feature interaction along the forwarding route is as follows: A caller may call some number not on the OCS (Originating Call Screening) 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 (Call Forwarding) does not consult the caller's OCS list. One way of removing the defect (not shown) is to route the diversion path back through the calling agent to check if the number is forbidden. This provides an example of how UCMs may be used to discover and correct problems at a very high level of abstraction.

Figure 13 displays token traces for scenarios with and without feature interaction, extracted from the previous UCM (to make these traces clearer, the two

Answer-Side agents involved are shown separately instead of stacked). Again, these traces are implicit in the original UCM so this diagram only provides examples of what the mind's eye sees and is not necessary for UCM documentation.

As we continue to apply UCMs we learn more about how to make clean UCM models. The UCMs here follow a paper presented to the agent community [2] and are cleaner and more understandable than the ones in a treatment of the same problem in a previous HICSS paper [3].

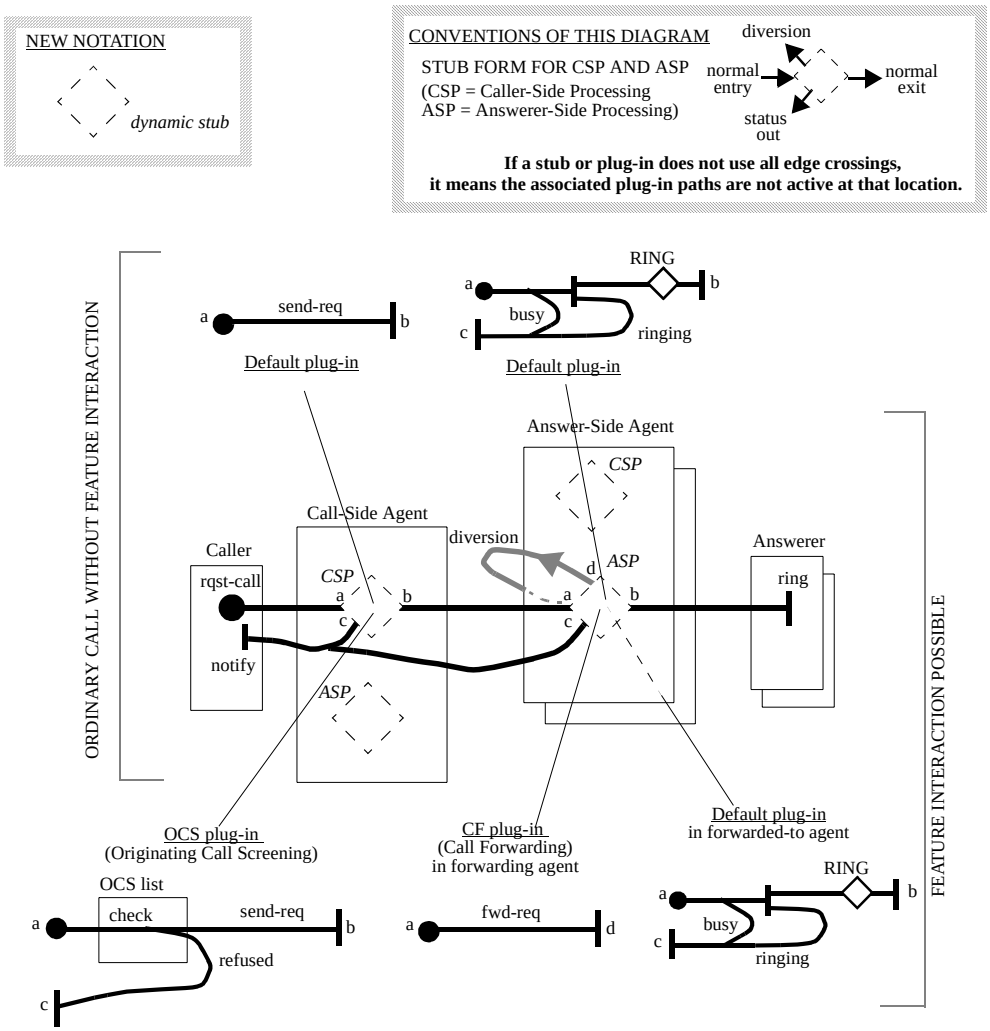
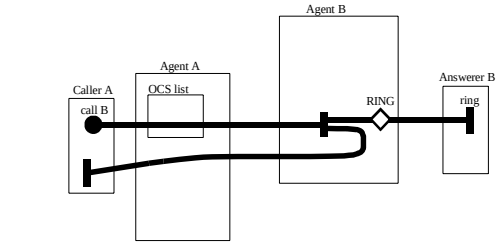
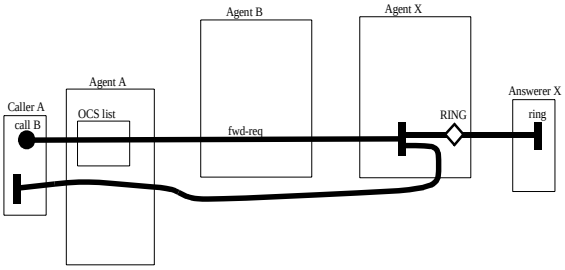


Figure 12 A telephony feature-interaction example.



(a) Without feature interaction.



(b) With feature interaction.

Figure 13 Token traces from the previous UCM for scenarios with and without feature interaction.

4 A Demonstration-of-Concept UCM Tool

A demonstration-of-concept UCM tool has been constructed [4]. This tool is providing a requirements/design front end for research projects on agent system design/prototyping and on performance evaluation of system architectures, among other things. Figure 14 shows a screen dump taken from a current study of the performance evaluation problem.

In this study, the tool is used for such purposes as 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 another tool that generates performance models and runs simulations to produce results. The idea, not yet realized, is to feed the performance results back through the GUI.

Figure 14 illustrates the general nature of the GUI, its support for timestamp points, and its support for identifying different types of components (the parallelograms represent concurrent processes).

The UCM tool is currently undergoing evaluation/refinement in collaboration with a few companies. We expect it will be available in the public domain by the time this paper is presented.

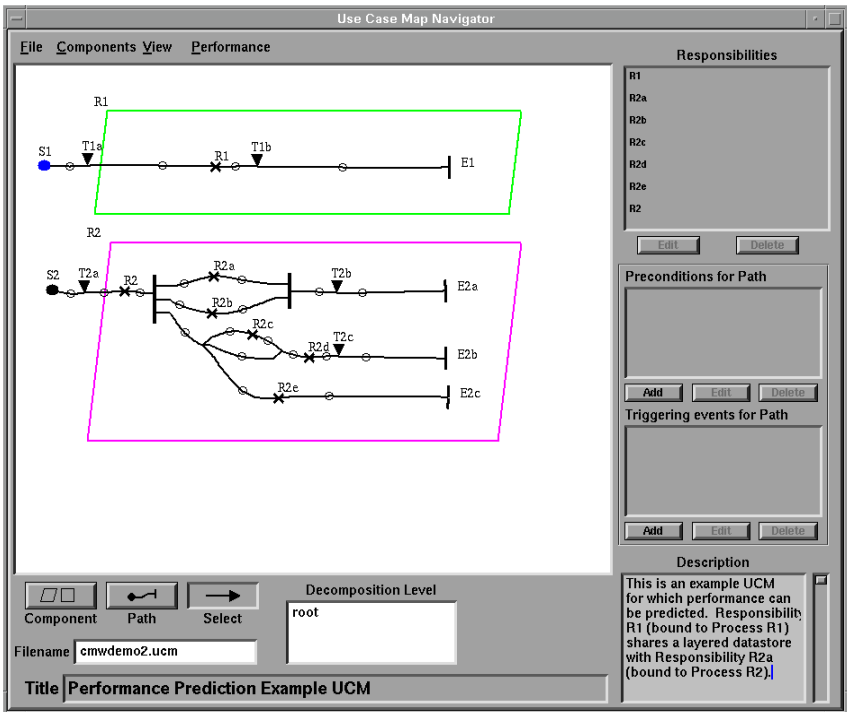


Figure 14 A UCM tool applied to performance evaluation.

5 Conclusions

This paper has aimed, not to present UCMs for the first time as a new concept, but to draw the attention of workers in the field of software architecture to the possibility that UCMs can resolve some difficulties in that field. An explanation has been provided of the core of the UCM notation that provides architectural expressiveness, illustrated by selected examples. A glimpse has been provided of a demonstration-of-concept UCM tool that is expected to be available in the public domain by the time this paper is presented.

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 the UCM tool example diagram contributed by Andrew Miga and Craig Scratchley. 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, *Use Case Maps as Architectural Entities for Complex Systems*, to appear in a forthcoming IEEE/TSE issue on scenario management
www.sce.carleton.ca/ftp/pub/UseCaseMaps/ucmUpdate.ps (or .pdf)

[2] R.J.A. Buhr, D. Amyot, D. Quesnel, T. Gray, S. Mankovski, *High Level, Mult-Agent Prototypes from a Scenario-Path Notation: A Feature Interaction Example*, Proc. Third Conference on Practical Applications of Intelligent Agents and Multi-Agents (PAAM98), London, April 1998
www.sce.carleton.ca/ftp/pub/UseCaseMaps/4paam98.ps (or .pdf)

[3] R.J.A. Buhr, M. Elammari, T. Gray, S. Mankovski, two papers in the Information System Design mini-track, HICSS98, *A High Level Visual Notation for Understanding and Designing Collaborative, Adaptive Behavior in Multiagent Systems*, and *Applying Use Case Maps to Multi-Agent Systems: A Feature Interaction Example*.

[4] A. Miga, *Application of Use Case Maps to System Design, with Tool Support*, MEng thesis, Carleton University, Ottawa, Canada, Sep 1998

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

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