Use Case Maps as Architectural Entities for Complex Systems

R.J.A. Buhr
Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada.
buhr@sce.carleton.ca

ABSTRACT. *This paper presents a novel, scenario-based notation called Use Case Maps (UCMs) for describing, in a high-level way, how the organizational structure of a complex system and the emergent behavior of the system are intertwined. The notation is not a behavior specification technique in the ordinary sense, but a notation for helping a person to visualize, think about, and explain the big picture. UCMs are presented as "architectural entities" that help a person stand back from the details during all phases of system development. The notation has been thoroughly exercised on systems of industrial scale and complexity and the distilled essence of what has been found to work in practice is summarized in this paper. Examples are presented that confront difficult complex-system issues directly: decentralized control, concurrency, failure, diversity, elusiveness and fluidity of runtime views of software, self modification of system makeup, difficulty of seeing large-scale units of emergent behavior cutting across systems as coherent entities (and of seeing how such entities arise from the collective efforts of components), and large scale.*

1 Introduction

This paper presents a scenario-based notation for describing, in a high-level way, how the organizational structure of a complex system and the emergent behavior of the system are intertwined. The notation is not a behavior *specification* technique in the ordinary sense, but a notation for helping a person to *visualize*, *think* about, and *explain* the overall behavior of a complex system. The focus is on the big picture, not on details. To be practically useful for this purpose, the notation must be capable of dealing with the following complexity factors in an integrated and manageable way.

The behavior of the kind of system that interests us *emerges* from the self-coordinated individual efforts of semi-independent components, which operate (often concurrently) without the explicit help of any central algorithm or plan. The components must cope with the possibility of failures of both other components and intercomponent communication. The components may be software or hardware, may be distributed in a network, and may be quite diverse, such as objects, processes, threads, interrupt service routines, monitors, modules, packages, communication chips, workstations, network gateways, and web servers, to name but a few possibilities. To avoid confusion, note that the use of the term *component* here is both more and less restrictive than the use of this term in an important software notational standard, UML [31]: It is more restrictive because it refers only to runtime entities of systems; it is less restrictive because it does not apply only to software. A different term would avoid confusion but the term is so appropriate that it seems artificial to look for another.

Systems have large scale units of emergent behavior cutting across them, such as network transactions, that are part of the way we think about systems but that tend to be hard to see as coherent entities in the detailed terms in which they emerge at runtime. Another side of this coin is difficulty seeing how such entities arise from the collective efforts of components.

The focus on behavior implies, for the software parts of systems, that we shall be concerned with the runtime picture not the organization of source code. The runtime picture of software is often characterized by a certain elusiveness and fluidity, viewed from the perspective of a programming language. Elusiveness results from the fact that many types of runtime components may not be explicitly identified as such in the programming language (e.g., processes supported by separate operating systems, components defined by grouping source code elements into a file, container objects identified as such by the fact that they store identifiers for objects of other classes). Fluidity results from the fact that practical software systems may be self-modifying as they run, meaning they dynamically change their makeup as a routine part of normal operation (e.g., objects, processes, and threads are created and destroyed; visibility relations among them are changed; intercomponent protocols are changed; applets are moved from one network node to another; network transactions take different forms under different system conditions).

This mix of complexity factors presents a strong challenge to notations, with respect to showing the kind of big picture we seek. Notations may have two problems in this respect: 1) System-representation notations that require commitment to a lot of detail will produce descriptions that hide the big picture we seek behind clouds of details as the scale of the system increases. The current generation of software design notations, exemplified by UML, tends to have this problem.

2) Stand-alone scenario specification techniques that focus more on formal specification of sequences than on system organization will not give a self-contained architectural perspective. Section 7 provides references and more discussion with respect to these points.

Use Case Maps (UCMs) [5][6][7][8][9][4][10] are presented here as a solution to these problems. The notation was invented by the author in direct response to the complexities of practical systems, not as an extension of techniques appearing in the literature or used in the workplace, and it approaches these complexities from a unique angle. The basic idea is very simple and is captured by the phrase *causal paths cutting across organizational structures*. The realization of this idea produces a lightweight notation that scales up, while at the same time covering all of the foregoing complexity factors in an integrated and manageable fashion. The notation represents causal paths as sets of wiggly lines that enable a person to visualize scenarios threading through a system *without the scenarios actually being specified in any detailed way.* (The term *wiggly lines* may seem too informal for a technical paper but nicely captures the look of UCMs—e.g., Figure 1—while also underlining the fact that there is no more to the notation: Wiggly lines are what you see, and what you see is what you get. I am grateful to Ian Graham for this term from a review [13] of a book on UCMs [5].) Compositions of wiggly lines (which may be called *behavior structures*) represent large-scale units of emergent behavior cutting across systems, such as network transactions, as first-class architectural entities that are above the level of details and independent of them (because they can be realized in different detailed ways).

The notation is intended to be useful for requirements specification, design, testing, maintenance, adaptation, and evolution. By no means does this paper attempt to cover all of these topics. However, it does attempt to show that the notation is so lightweight and expressive that its usefulness for all of these purposes is at least plausible.

1.1 Outline of the Paper

Section 2 summarizes basic UCM principles (Section 2.1 presents the look of the core visual notation and Section 2.2 provides guidelines for interpreting the notation in behavioral terms, and using it). Section 3 provides some realistic examples that illustrate the foregoing complexity factors, and also introduces additional notation that helps a person to visualize the impact of these factors on a system. Section 4 and Section 5 enlarge on techniques for large-scale systems and self-modifying systems, respectively, that were introduced in Section 3. Section 6 completes the notation picture by presenting topics that were skipped over or only mentioned briefly in earlier sections. Section 7 summarizes UCM properties, compares UCMs with other techniques, and describes related work. Section 8 draws conclusions.

The paper is relatively long, for several reasons: It aims to demonstrate the unusual breadth of coverage of the notation and needs the length to cover important topics and examples that demonstrate this. It aims to convey the message that the path (wiggly-line) notation *by itself* leverages thinking; this idea is novel, takes some getting use to, and space is needed to make a convincing presentation of it. The paper aims to provide an up-to-date, compact reference for all important features of the visual notation, including interpretations and extensions that have emerged from recent experience.

In spite of its length, some topics that a reader might expect should be included are deliberately omitted as side issues that would distract from the main message (to say nothing of adding to the length). These include notations for different component types, formats for textual documentation, and a formal description of visual syntax (a syntax has been developed for a tool, but is somewhat tool-dependent in its present state and would be out of place in this paper).

The notation has been thoroughly exercised on systems of industrial scale and complexity and the distilled essence of what has been found to work in practice is summarized in this paper. Everything in this paper is explained only in words and pictures, because this gives the best idea of how UCMs are intended to be used.

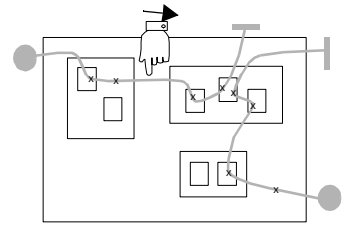
2 Basic Principles of UCMs

2.1 The Core Notation

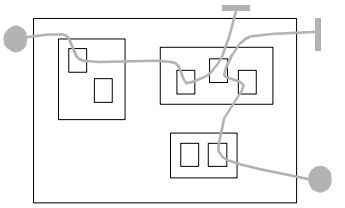
As shown by Figure 1, the core notation, is very simple. The notation has only three fundamental elements: *Scenario paths* are represented by wiggly lines; rectangular boxes represent runtime *components*; *responsibility-points* along paths touch components to indicate that components have responsibilities 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

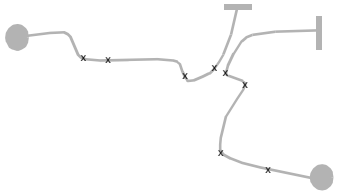
items, and responsibilities may be identified along paths to operate on these data items, but there is no formal representation of the data items and no formal way of specifying how responsibilities use or change them. This lack of formality in the details is what makes the notation lightweight.



(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 The Essence of UCMs.

This notation deals with the complexity factors identified in Section 1 in the following ways:

- *Lack of any central algorithm or plan.* Paths are, in effect, high-level visual representations of implicit plans, as seen by a person from a perspective outside the system.
- *Concurrency.* The path notation by itself is concurrency-neutral, meaning any set of paths may be interpreted as having concurrent scenarios progressing along them or not. The proper interpretation is provided by the application context and/or the types of components to which the paths are bound.
- *Failures.* Paths may represent normal operation or recovery after a failure. Additional notations introduced in Section 3 may be used to highlight failure points and identify failure-handling paths.
- *Elusiveness of software runtime components.* The component notation has no programming-language baggage associated with it. A box may represent any type of runtime component, whether or not it is explicitly identified in a programming language.
- *Diversity.* The simple notational elements of Figure 1 stand back from the kinds of details that make practical systems so diverse, to focus on the high-level aspects that make them similar.
- *Difficulty of seeing large-scale units of behavior cutting across systems as coherent entities (and of seeing how such entities arise from the collective efforts of components).* This is the central purpose of UCMs.
- *System self modification.* Self modification is represented in a lightweight fashion by simple additions to the notation (Section 3, Section 5) that identify placeholders for dynamic elements (components and submaps); the picture is completed by adding responsibilities that treat these dynamic elements as data.
- *Large scale.* The notation has no inherent scale and minor notational enhancements (Section 3, Section 4) provide specific mechanisms for creating related sets of diagrams that provide a coherent view of large-scale systems.

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. Section 4 covers these topics, but a brief indication of how they relate to the component substrate will be helpful here. 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.

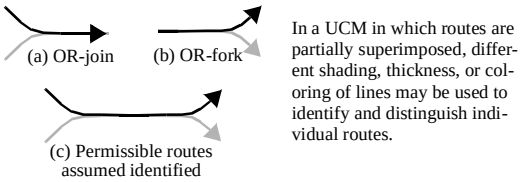


Figure 2 Shared routes and OR forks/joins.

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.

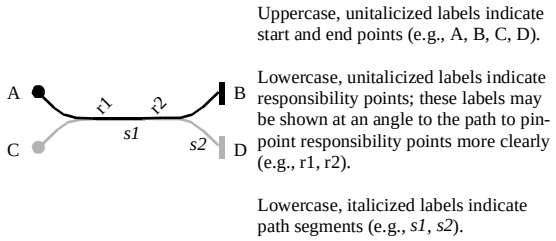


Figure 3 Labeling conventions.

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).

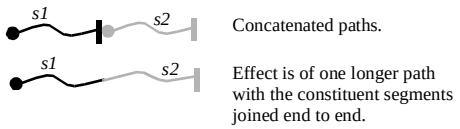


Figure 4 Concatenated paths.

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.

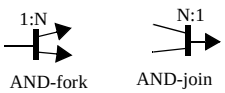


Figure 5 AND-forks/joins.

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.

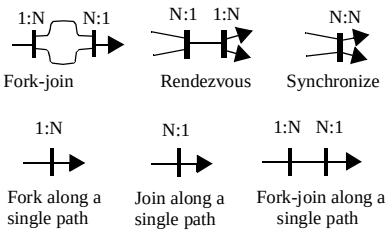


Figure 6 Variations on AND-forks/joins

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.

2.2 UCMs and System Behavior

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, as follows (guided by heuristics which will be explained beginning in Section 2.2.1):

- Paths enable a person to *infer* scenarios by mentally moving tokens along them (think of the pointing finger in Figure 1 as moving a token), without requiring complete specification of all the details that govern the actual scenarios.
- Responsibilities along paths enable a person to *visualize* how implied data changes and transfers will be performed and observed along and between paths, without the presence of any explicit representation of data, and thereby to gain an informal understanding of how scenarios affect the system state. Thus scenario preconditions and postconditions that make statements about the system state may be given meaning even though UCMs have no explicit way of representing the system state.
- Composite maps with many paths enable a single diagram to represent many related scenarios, allowing a person to *visualize* interscenario relationships.
- Paths crossing components enable a person to *visualize* how scenarios affect components and how different scenarios interact through components.
- Paths crossing intercomponent gaps enable a person to *infer* intercomponent communication requirements.
- The component substrate enables a person to *think* about the behavior picture in light of assumptions about the nature of components, intercomponent communication, and implementation technology.
- After working with a set of UCMs for a while, people start to see them as helpful visual patterns that *stimulate thinking and discussion* about design issues at both the level of UCMs and at more detailed levels.

2.2.1 Inferring Scenarios

The UCM notation is path-specification technique (in the sense of wiggly lines not formal path expressions). The paths do not specify how scenarios that follow them will emerge. This is because UCMs do not specify anything about details such as data transfers along paths, local data values at points along paths, and local decisions based on local data values. As formal specifications, UCMs provide only the information that the identified paths, components, and responsibilities exist and are related as shown visually.

Reading UCMs is like playing a board game with multiple players and a token for each player. Just as the board layout and the game rules determine possible play scenarios, so does a UCM and some rules determine

possible scenarios. The rules are provided by the meanings of the notations, the nature of the system and its components, and knowledge of the nature of scenarios that the paths are supposed to represent (which includes having an informal understanding of the nature of their preconditions and postconditions). In the examples in Section 3, the rules needed to understand particular UCMs will be explained informally in unstructured prose. Structured prose documentation is a housekeeping issue not covered in this paper.

Here is how to read a UCM: Put a token (mentally or literally) on each start point from which a scenario may start concurrently. Then move each token along its path. If the rules allow it, start other tokens from the same start points before previous tokens have reached their end points. Start tokens at different times relative to each other and move them along paths at different rates to experiment with possible race conditions in concurrent situations. Generate additional tokens along the way at AND-forks (or take them away at AND-joins). If tokens have to wait at certain responsibility points, continue to move other tokens, and begin moving the waiting tokens again when events have cleared the waiting condition (see Section 3 for examples of waiting situations). When a token reaches an end bar, remove it from the UCM.

If alternative routes are identified by OR-forks and OR-joins, follow different routes in different sessions. Although it may seem that some sort of logic is being invoked when choosing which path of an OR-fork to follow with a token, the concept is that a route has been mentally selected in advance, according to assumed preconditions and stimuli, and then followed end to end. 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.

Understanding how scenarios affect the system state and, through it each other, requires paying attention to responsibilities. The interaction of scenarios through the system state is indicated by responsibility names and descriptions. Understanding the effects of responsibilities on the system state leads to understanding scenario preconditions and postconditions and vice versa. As said earlier, all of this is informal. A quick overview of the nature of scenarios and their possible interactions may often be obtained without paying much attention to responsibility points except as markers to indicate which components are involved.

Figure 7 reminds the reader that consumption of real time along paths, although not literally specified by the notation of this paper, is always implied.

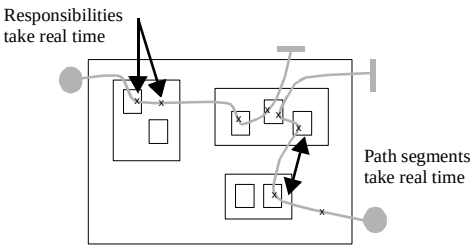


Figure 7 Real time along paths is implied by the assumed existence of components, intercomponent communication, and implementation technology.

2.2.2 Paths Crossing Components

Paths crossing intercomponent gaps enable a person to *infer* intercomponent communication requirements. Paths crossing components enable a person to *visualize* how components control local parts of scenarios and how scenarios interact within components.

A segment of a path joining responsibilities of two different components, as shown in Figure 8(a), implies that the *components must communicate after the first responsibility has completed and before the second one has started*, but not how they will communicate. In general, there are many ways of arranging communication at a more detailed level (see Section 6.1). The simplest would be for the first component along the path to make a call or send a message to the second. Other possibilities include exchanging a series of calls or messages, starting with either component, or communicating through some underlying layer.

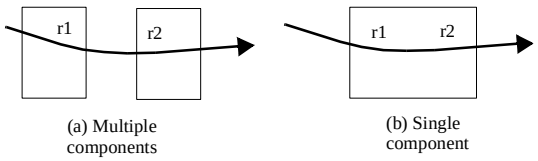


Figure 8 Paths crossing components.

A segment of a path that crosses a component, as shown in Figure 8(b), tells us that component *controls the sequence of responsibilities within its edges*. In general, there are many ways to arrange matters at a more detailed level (see Section 6.1). For example, responsibilities may be programmed as functions, or sets of functions, and their execution arranged through calls chained together in various ways, triggered by intercomponent calls or messages. An important property of UCMs is their ability to provide a framework for reasoning about such details without requiring commitment to the details.

Interscenario relationships are important aspects of UCMs. Beyond the specialized relationships of Figure 6, interscenario relationships are indicated by

paths sharing a responsibility or traversing a shared component, as shown in Figure 9. For example, a responsibility along one path may be identified as changing the state of a shared component, and responsibilities along other paths as examining the state to make decisions or extract data. Such relationships may exist between concurrent or sequential scenarios. In the former case, an effect of a responsibility may be felt by another scenario before the causing scenario is finished. In the latter case, the effect may occur after the causing scenario is finished. An earlier statement said that end points of paths are where stimuli stop actively rippling through the system. However a stimulus may have an indirect effect after this, due to changes made to the system state in the manner just described.

Interference may occur when multiple concurrent scenarios traverse a single component, and mutual exclusion may be desirable to prevent it. In the example of Figure 9, if mutual exclusion is required the descriptions of r1 and r2 would say that further scenarios along any of the paths must wait while a single scenario on any path is performing either r1 or r2 (the component notation of [5] includes symbology for components that enforce such mutual exclusion, often called monitors).

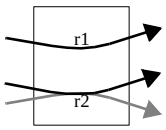


Figure 9 Interscenario relationships.

The same paths may be interpreted quite differently if components along them are passive objects or active processes (or threads). Concurrent scenarios may interfere with each other as they traverse a passive object, because the object has no independent thread of control to prevent it. For an active system component (e.g., process, thread), the opposite is true.

A book on UCMs [5] distinguishes visually between different types of components (e.g., objects, processes, teams). Although this can be useful, it adds nothing to the path notation, which is the focus of this paper, so we do not present this component notation here.

Observe that the paths of UCMs are *not* control paths. In the types of systems for which UCMs are most useful, control is often decentralized among many components along a path, with no single component having control of the whole path. UCMs do not indicate how control of paths is to be achieved. All they show about control is that all components along a path must somehow, collectively, exercise control to make the implied sequences emerge and that all paths traversing a single component are somehow controlled by that component. Control details are at a lower level of abstraction.

2.2.3 Using UCMs

UCM elements may be developed in different orders: Paths and responsibilities may be developed before components during requirements analysis. Paths and components may be developed before responsibilities for back-of-the-envelope style design exploration. Components and responsibilities may be developed before paths for cases where the design starting point is component organization and functionality rather than scenarios. Thus the notation is intended to be very flexible.

UCMs are particularly useful for design exploration. A designer may explore different component substrates for the same paths (which amounts to designing a system organization to realize end-to-end behavior requirements expressed by the paths) or explore different path-structures for the same component substrate (which amounts to designing the behavior of a given system organization at a high level of abstraction).

UCMs provide a high level framework for reasoning about the implications of low-level details. A qualitative real-time picture, useful for exploring design issues and alternatives, may be formed based on assumptions about the nature of components, intercomponent communication, and implementation technology, without these things being explicit in a UCM. For example, referring to Figure 7, there will be progressively more communication overhead for path segments implemented with inter-object function calls, inter-process communication, or inter-computer messaging. We understand these things, not because of anything explicit in UCMs, but because we can mentally project onto UCMs an understanding of how objects, processes, and communications are implemented in a particular system.

As was said in the introduction, the notation is intended to be useful for requirements specification, design, testing, maintenance, adaptation, and evolution. From a requirements-design perspective, the idea is to design the pieces of localized logic that give rise to emergent behavior not as pieces developed separately in an ad hoc manner, but as related pieces that flow from UCMs identified during requirements analysis. From a testing perspective, the idea is to choose tests that will give suitable coverage of UCMs. From a maintenance perspective, the idea is to use UCMs as references for understanding how to change details. From an adaptation-evolution perspective, the idea is to use existing UCMs as a starting point for defining new ones.

2.2.4 Visual Patterns

After working with a set of UCMs for a while, people start to see them as helpful visual patterns that *stimulate thinking and discussion* about design issues (the term *pattern* is not used here in the specialized technical sense adopted by the object-oriented patterns community). To make this possible, visual structures in UCMs

must be shaped with considerable care and artistry. 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, seeing useful visual patterns is not dependent on absolute shapes, but relative ones. 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.

3 Examples and Additional Notation

Published work on UCMs includes applications to object-oriented frameworks and design patterns [6][8][9], attributing behavior to system architectures [7], front ending conventional design models of the UML-ROOM variety [4] and formal description techniques [1], and designing/understanding dynamic agent systems [10]. 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.

A few realistic examples will now be presented to illustrate 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).

3.1 Behavior Structures for Network Transactions

Figure 10 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).

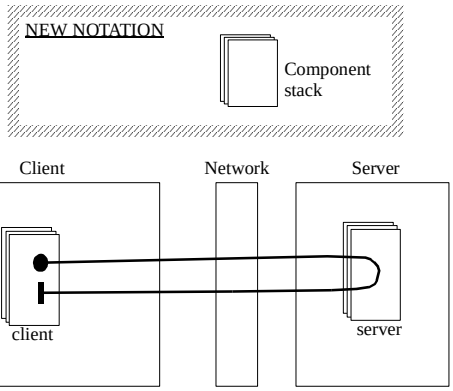


Figure 10 A simple network transaction.

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.

Figure 10 illustrates a powerful property of UCMs, namely the ability to defer decisions while providing a context for thinking about them. The decision that is deferred here is how transaction-path concurrency will be implemented. It 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 (such a commitment may be made by appropriate labeling, or by using the component notation from the UCM book [5] that provides different shapes for different types of components).

Figure 11 adds more "behavior structure" to provide for transaction completion in case of network failure. 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.

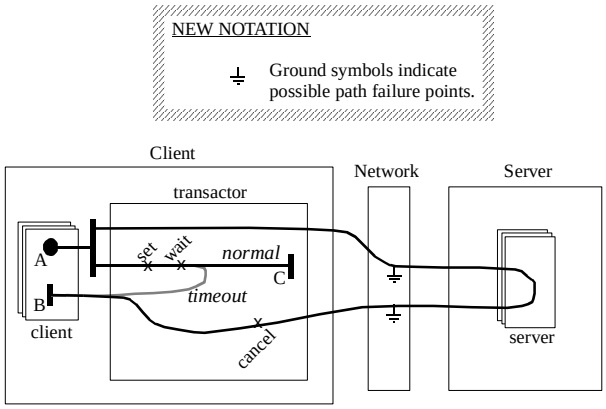


Figure 11 A generalized network transaction.

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 12 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.

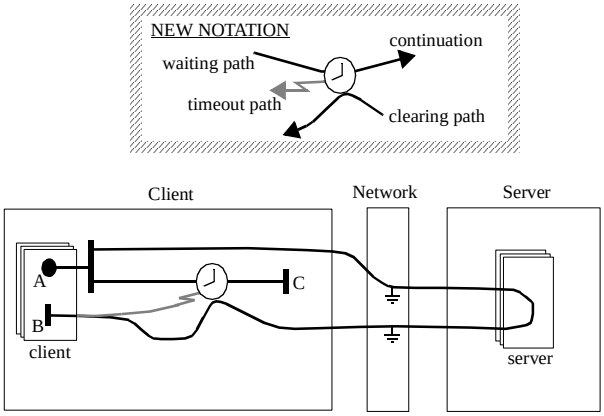


Figure 12 A generalized network transaction expressed with a visual shorthand.

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 13 gives additional insight into how to read Figure 12 by displaying the token traces that should be imagined while reading it. In either case, the original UCM embodies all the possibilities; there is no need to draw separate diagrams (except for presentation purposes, as here).

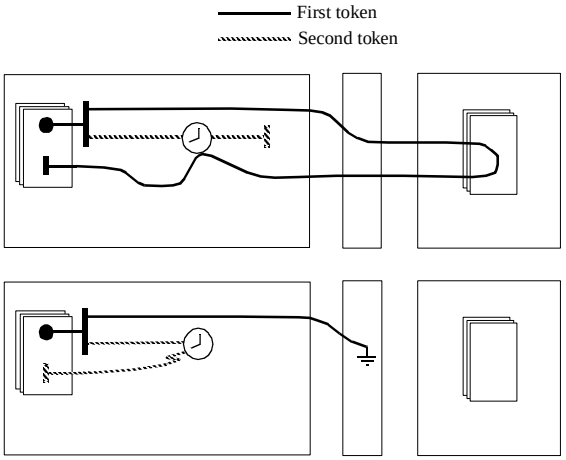


Figure 13 Different token traces that should be imagined while reading the previous UCM.

Figure 14 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 14, plug-in 1 with pass-through paths (which reduces Figure 14 to Figure 10), and plug-in 2 with additional failure handling capability (which reduces Figure 14 to Figure 12). 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 UCM reader to piece together paths cutting across a whole system from many plug-ins described in many different diagrams. How to stub is not always obvious because UCMs may have different purposes. For example, the UCM of Figure 14 focuses on transactions, leaving recovery mechanisms as details. However, what if the requirement is to focus on recovery mechanisms in the context of a system as a whole? Then leaving them to stubs may be undesirable.

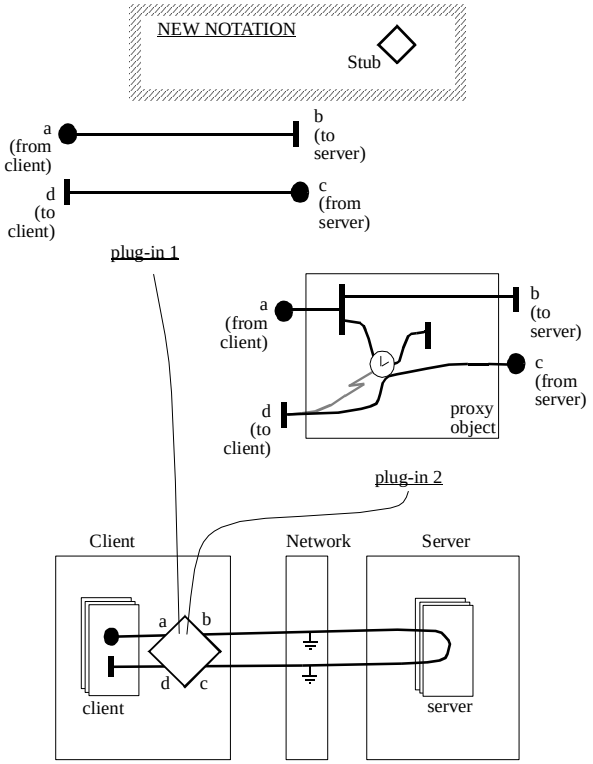


Figure 14 Deferring details with stubs and plug-ins.

3.2 A System Modifies it Own Components

This example illustrates how UCMs express situations in which a system dynamically modifies its own makeup, in terms of components, as it runs. The example is provisioning device handler objects from a central site to customize device driver processes for newly connected devices (Figure 15).

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 only new notation is a dashed-outline box, called a *slot*, which represents a potential location for a dynamic component (DC); the concept is that a slot is empty and therefore inoperative until a DC is installed in it. In this case, the DC would be a handler object. This diagram clearly shows that device operation at b4 follows local installation of a handler object. The diagram should be relatively self explanatory.

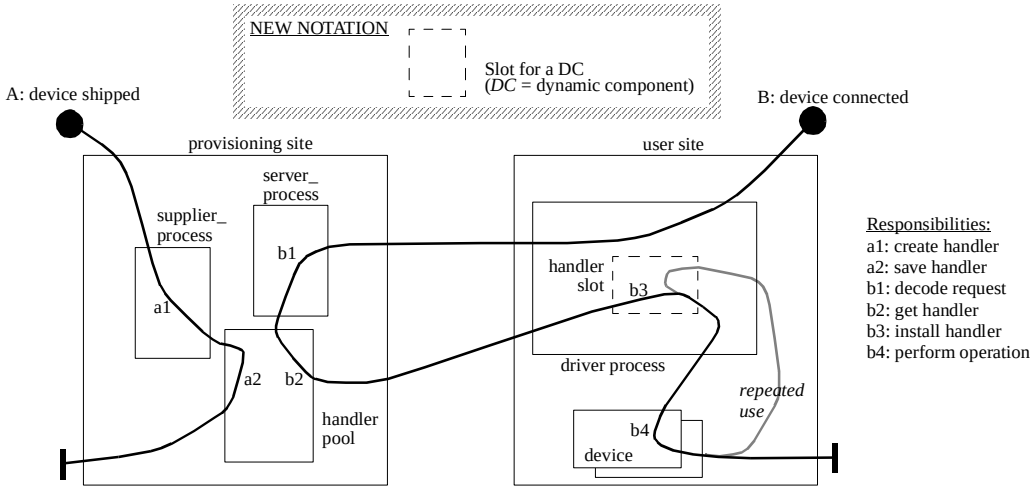


Figure 15 Provisioning device handlers.

Figure 16 shows how a bit of visual shorthand to express boilerplate operations associated with creating, moving, saving, installing, and destroying DCs can help to make diagrams more readable at a glance (to those used to the notation) and to reduce textual clutter.

In Figure 16, *pools* identify places where DCs may be temporarily stored as data. Small arrows identify responsibilities for creating and moving DCs as data. The only explicit responsibilities from Figure 15 that remain along the paths of Figure 16 are the two that do not directly relate to the foregoing boilerplate operations (b1, b4).

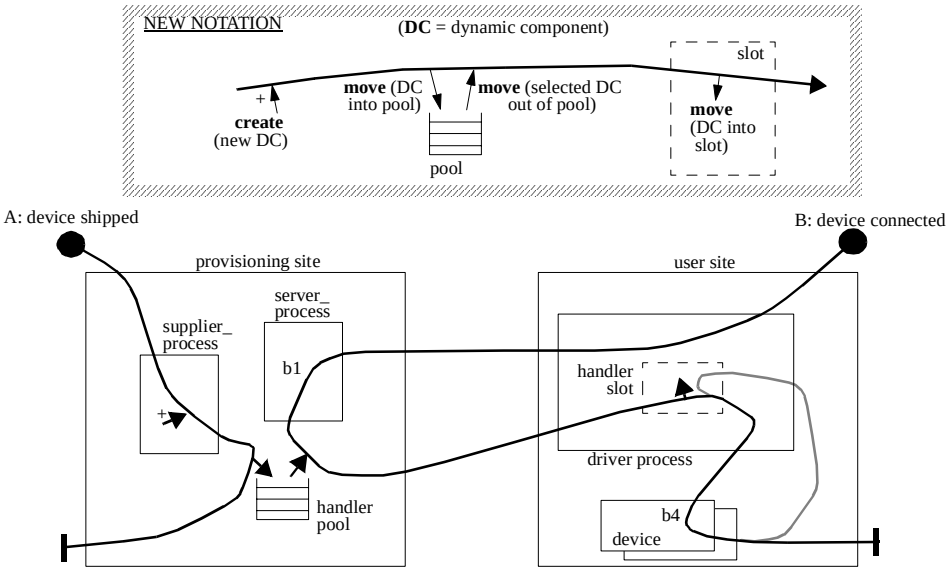


Figure 16 The same example using some notational shorthand.

3.3 Many Scenarios in One Diagram

This example (Figure 17, Figure 18) is drawn from ACE (Adaptive Communication Environment), a well known, public-domain, object-oriented framework for communications applications (for those interested in ACE, the sources are [20][21][22][23][24][25]). For those not specifically interested in ACE, the references should only serve to indicate that there is a lot behind this

example; they are not required reading to understand what is presented here. The example explains part of a gateway application that is included in the ACE distribution software. Although the gateway application is not by itself very complex as communications systems go, understanding it from reading ACE code and documentation is remarkably difficult (this is typical of frameworks).

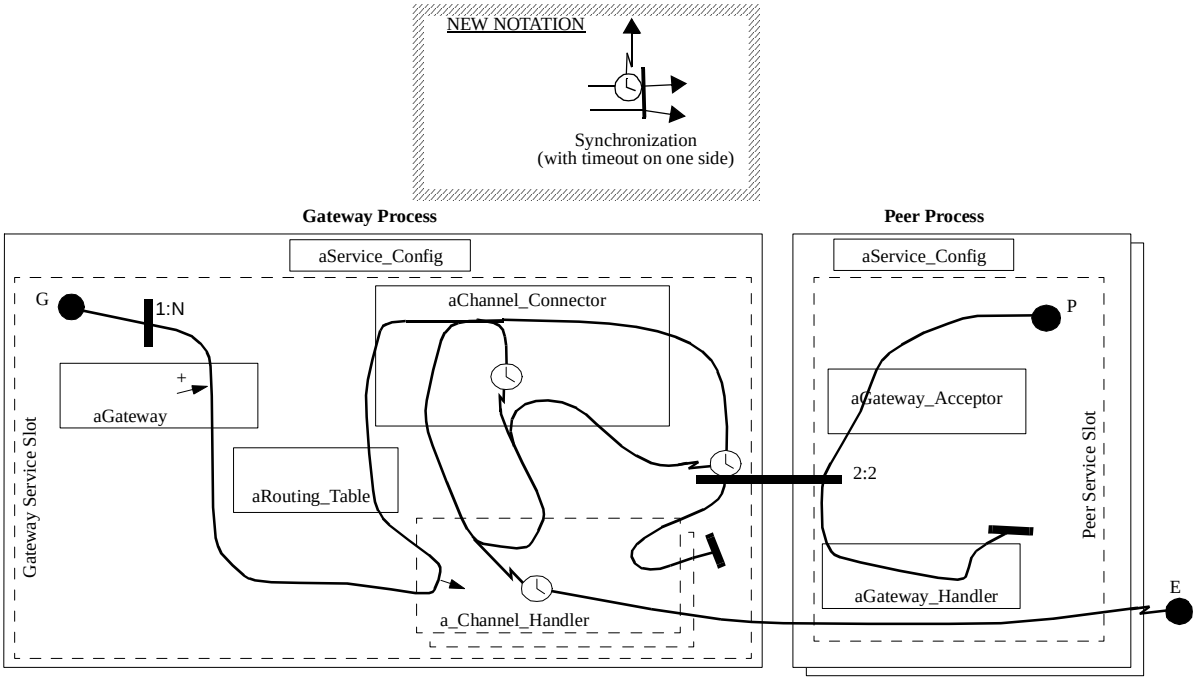


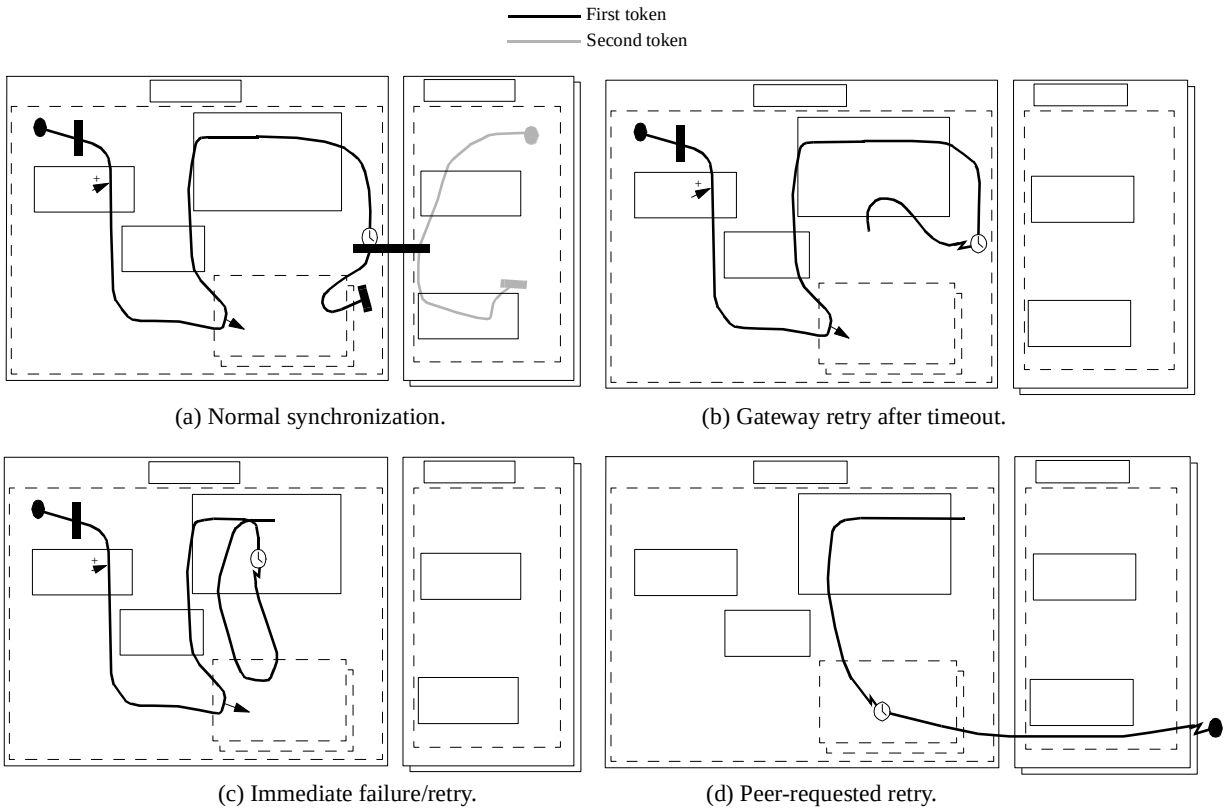
Figure 17 Startup synchronization in the ACE gateway application.

Figure 17 uses a shorthand notation that combines the synchronize notation of Figure 6 with the timeout notation of Figure 12 to indicate mutual synchronization of scenarios with timeout on one side. Given that the elements of this shorthand were explained in some depth previously, we shall rely on the explanation of Figure 17 that follows to make its meaning clear.

Figure 17 shows how processes in different network nodes of the gateway application synchronize on start-up. One does not need an explanation of the whole gateway application to follow this diagram, only to know that there is one gateway node and many peer nodes and that the precondition for the paths is that an initialization mechanism, not shown, has populated the service slots with appropriate dynamic components (DCs) containing the internal fixed components and slots shown in the diagram. The initialization leaves a set of handler slots in the gateway process unoccupied (the initialization specifies the number of peers, thus identifying the number of slots needed for them). The purpose of this UCM is to show how handlers to fill these slots are dynamically created and initialized, in cooperation with a handler in each peer. The handlers could be objects or threads (ACE supports either); the UCM paths are the same for either case, just interpreted differently for implementation purposes.

The AND fork near the beginning of the G path indicates that many concurrent scenarios progress along this path concurrently (as many scenarios as there are peers to be initialized). This means that explaining the G path for one peer serves to explain it for all. The G path shows the following sequence: create a handler for a peer and install it in the appropriate slot; either succeed or fail to synchronize with the peer (different routes are taken for the different cases); then initialize the handler. Different routes express different ways of giving up on the synchronization attempt (immediately when some system condition prevents communication, or after timeout) and for retrying after a back-off period. The path P is the same for each peer and simply synchronizes with the gateway to initialize its own (single, fixed) handler. An error path E from the peers indicates that a retry may come from the peer (details of the error are not important here).

For those familiar with ACE, this UCM directly incorporates two ACE patterns (connector and acceptor) in UCM terms and treats another ACE pattern (reactor) as part of an invisible underlying layer.



(a) Normal synchronization. (b) Gateway retry after timeout.

(c) Immediate failure/retry. (d) Peer-requested retry.

Figure 18 Different token traces that may imagined while reading Figure 17.

Figure 18 gives additional insight into how to read the foregoing UCM by showing some possible traces and partial traces of token movements that may be imagined while reading it. The partial traces are for retries that could result in either normal synchronization or more retries. As before, the original UCM embodies the different possibilities; there is no need to draw a separate diagram for each.

3.4 A System Modifies it Own Path Structures

This example (Figure 19) 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 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 20 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).

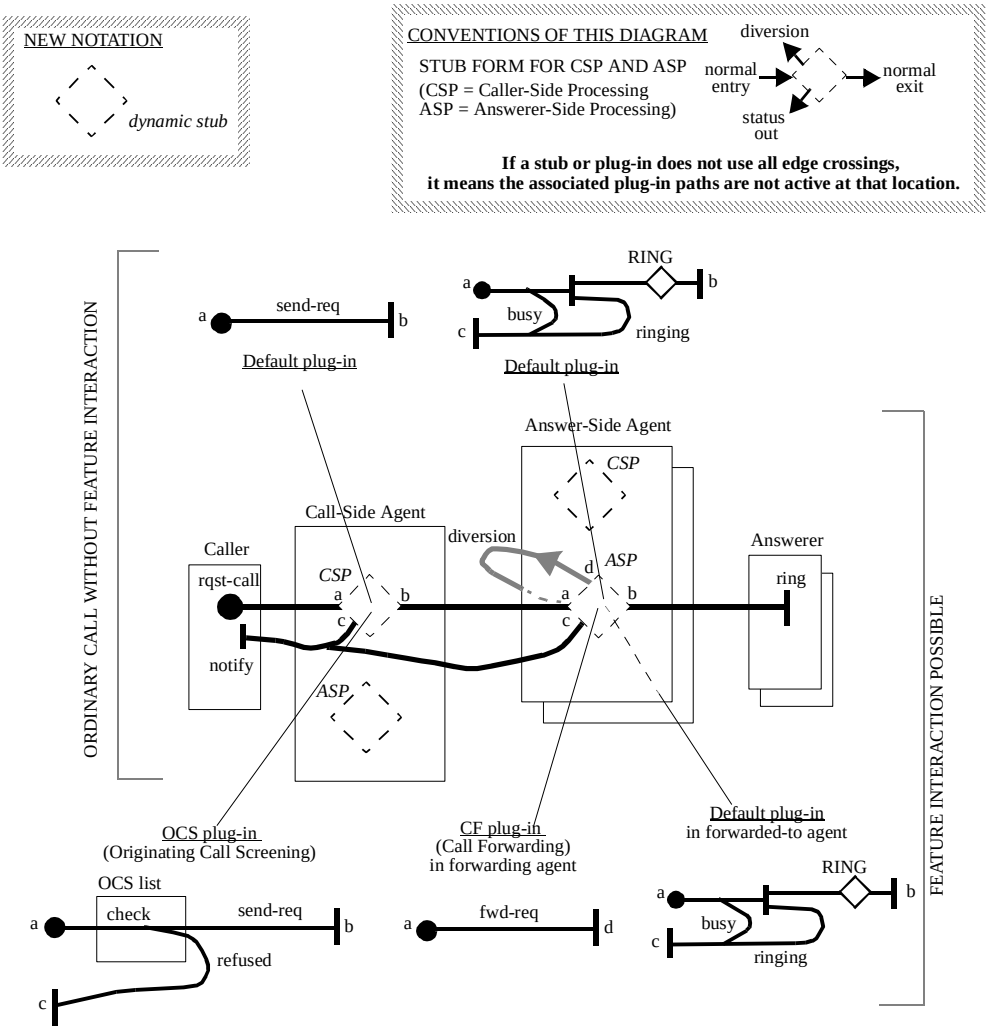
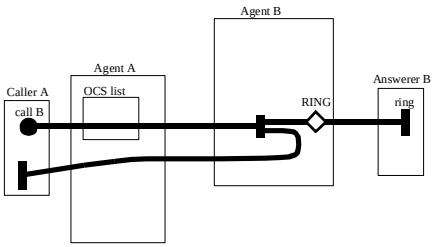
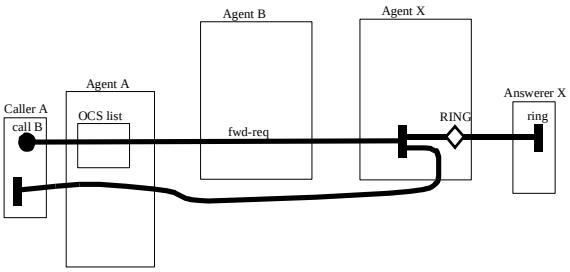


Figure 19 A telephony feature-interaction example.



(a) Without feature interaction.



(b) With feature interaction.

Figure 20 Token traces from the previous UCM for scenarios with and without feature interaction.

4 Large Scale Systems

Figure 21 positions architectural descriptions of large scale systems in a space defined by three orthogonal coordinate axes—abstraction, decomposition, and layering—and gives some examples of points along each axis. These points are only intended to indicate the nature of the axes, and are not intended to be complete. One of the points introduces a new term, *wiring*, used here as an umbrella category for any technique that expresses system behavior in terms of intercomponent calls and messages that flow over literal or conceptual intercomponent connections (i.e., wiring), such as [3][31][16][26]. The three axes are orthogonal to indicate that, in general, levels of abstraction, decomposition and layering may be chosen independently.

Figure 21 views abstraction as a paradigm shift not just deferral of detail. Both decomposition and layering defer detail. However, unless deferral of detail is performed by making a paradigm shift, it is regarded in the scheme of Figure 21 as simply an organizing technique within a single level of abstraction. Thus decomposition and layering are regarded as organizing techniques. Granularity of description is a separate issue. At any level of abstraction, different degrees of granularity may be employed for both decomposition and layering; and, at higher levels of abstraction, fine-grained details may be left to lower levels.

The nature of the view of Figure 21 is illustrated by the following examples of decomposition and layering at the implementation level of abstraction: decomposition may be performed by packaging source code into compilation units; layering may be performed by making a layer a separate runtime unit to which other code must be dynamically linked after loading (e.g., an operating system). Although deferral of detail takes place in both these examples, in the terms of Figure 21 this is accomplished by code organization not abstraction. Abstraction requires making a paradigm shift away from code.

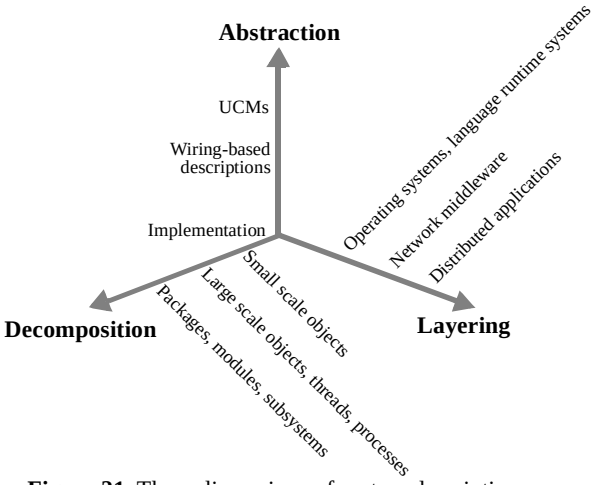


Figure 21 Three dimensions of system description

Here is how to interpret Figure 21: A point along the abstraction axis identifies the level of abstraction of the description technique (the farther away from the origin, the more distant it is from implementation details). A point along the decomposition axis identifies the existence of an actual system description at a particular level of decomposition (the farther away from the origin, the more levels of decomposition exist underneath). A point along the layering axis locates this description in a set of layers (the farther away from the origin, the more layers exist underneath).

These are not the only dimensions of system description (e.g., they do not include class hierarchies in object-oriented programs), but they are the only ones that UCMs directly express. Other dimensions may be related to UCMs (e.g., [6]), but these matters are outside the scope of this paper.

Whether a component in a UCM is regarded as a peer component of others or as underlying infrastructure in a separate layer is largely a matter of how we choose to regard it. There may be little or no difference at the code level, apart from how the code is managed and maintained (again, this provides reinforcement for saying that layering is not by itself an abstraction technique, but is an organizing technique for a description at any level of abstraction). Some CASE tools at the wiring diagram level of abstraction (e.g., ObjecTime) support layering explicitly.

In general, many complementary system descriptions may exist for a practical system, characterized by many points in the space defined by these axes. For example, a top layer of a system might be described first by UCMs (perhaps with many levels of UCM decomposition) and also, as a step towards implementation, by descriptions at the level of wiring diagrams. Lower layers of a system might not merit being described with UCMs, but might be described only at the level of wiring diagrams, or even left to code.

Paradigm shifts along the abstraction axis do not necessarily imply replacement of lower level paradigms. Just as descriptions at the level of wiring diagrams do not necessarily replace code (although tool vendors try to go as far in this direction as possible), so UCMs do not replace descriptions at the level of wiring diagrams. They only supplement them to give a better view of the big picture.

Earlier UCMs in this paper show a single layer of a system, with at most one level of component or stub decomposition (component black boxes were opened up as glass boxes to show the second level in the same diagram, stubs were shown separately). These diagrams showed selected aspects of systems that might or might not be large scale. UCMs by themselves have no inherent scale. Their scale is determined by their relationships to other descriptions. The key to using UCMs for large scale systems is splitting them into individually understandable diagrams that form a coherent picture of the whole in UCM terms, and that are also clearly related to descriptions at lower levels of abstraction. UCMs do not have to (and should not try to) do it all.

A number of useful UCM techniques exist for dealing with large-scale systems. Two of them, *separation* and *factoring*, are not identified in Figure 21 because they amount to little more than diagram management. They are described in Section 4.1 and Section 4.2. Decomposition and layering are described in Section 4.3 and Section 4.4.

4.1 UCM Separation

When UCM diagrams become too complicated for a person easily to separate out the possible routes by eye, a useful technique is to separate different routes through them into different *presentation diagrams*, against a background of the same component substrate, possibly with all the other paths shown lightly, to provide context (e.g., Figure 18). When UCMs become too complicated to show all paths on a single diagram, different aspects of operation—for example, initialization/finalization, error discovery/recovery, normal operation in different modes—may also be separated into different presentation diagrams, but still against a background of the same substrate. These techniques are analogous to the management of transparency overlays.

4.2 UCM Factoring

Factoring (Figure 22) means cutting a UCM into fragments by cutting its paths at judicious points, adding start and end symbols appropriately to the cut ends, and then separating the fragments into separate UCM diagrams. Conceptually, the UCMs in these diagrams are connected peer elements of a single UCM (the conceptual relationship is intended to be preserved through documentation). The start and end symbols added after cutting amount to off-diagram connectors, which must be appropriately labelled to make the peer relationships clear (to minimize the number of symbols in the notation, no special notation is provided for off-diagram connectors). Compatible peer UCMs may be composed into a single UCM by the reverse process of path concatenation.

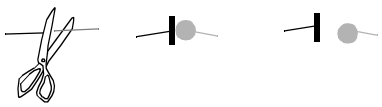


Figure 22 Factoring.

Factoring is useful for splitting a large diagram into smaller diagrams at the same level of decomposition (several levels of component decomposition may also be collapsed into a single diagram that may then be factored). However, factoring leaves dangling path ends that have to be traced to other diagrams to get an overview of paths.

4.3 UCM Decomposition

Decomposition refines UCM elements recursively. *Component* decomposition makes black boxes into glass boxes with their own internal components, with the paths simply following along. *Path* decomposition enables paths to be refined in very general ways, using *stubs* to indicate where path details are deferred and *plug-ins* to describe the details. Plug-ins may jointly refine both paths and components.

Decomposition has several nice features: it keeps the big picture intact (no dangling path ends); it provides for reusable pieces; and it has a useful runtime interpretation—choosing an appropriate decomposition at the last minute to match current system conditions. However, care must be taken not to decompose UCMs so drastically that the big picture is no longer clear, because this defeats the whole purpose. Particular care must be taken with path stubbing, which can easily be used to hide the end-to-end connectivity of paths, which is exactly the opposite of what we want UCMs to do. Decomposition and factoring may be performed as alternative steps of a recursive design process.

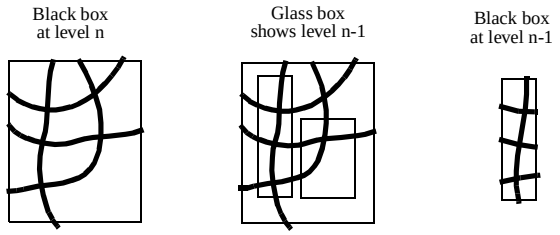


Figure 23 Component decomposition with black boxes and glass boxes.

Component decomposition (Figure 23) is the simplest and least general form of UCM decomposition, but is nevertheless useful. A black box with paths traversing it at one level becomes a system of black boxes with the same paths winding through them at the next level down. Path segments at all levels show the same responsibility sequences, the only difference being that the same responsibilities are bound to progressively finer grained subcomponents (this may require distorting the paths in successive lower-level diagrams). Containment of components within other components in a diagram like Figure 23 only has an operational meaning and does not commit to whether or not enclosing components are code-level containers. Therefore UCMs may be used to explore many levels of component decomposition without committing to details at the level of component interfaces or intercomponent interactions. The absence of commitment to such details makes relatively lightweight work of exploring design alternatives, reducing the tendency that arises at lower levels of abstraction to give up exploring alternatives too soon. If the lowest level components in a UCM such as Figure 23 are primitive, meaning they contain no internal components, but only logic to implement the path segments crossing them, then we may proceed directly to detailed design of the logic (Section 6.1). Otherwise, decomposition may continue in terms of either paths or components.

Path decomposition is more general than component decomposition. In the sequence of decompositions in Figure 23, there is no provision for decomposition of responsibilities or path segments. However, Figure 23 hints at an approach to such decomposition, namely regard path segments traversing the black boxes as stubs for subpaths. This is the germ of the idea of path decomposition in UCMs. However, instead of path segments themselves being treated as stubs, a more general approach is to superimpose stub symbols (diamond shapes) on the path segments (Figure 24). (For a number of reasons, a diamond shape has been adopted for stubs instead of the superimposed-bar-and-filled-circle icon used in [5] and some other publications, but readers who like the other icon can think of the diamond shape as an abstract outline of it).

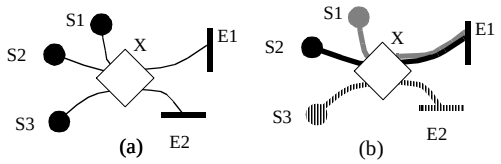


Figure 24 Stubs and path continuity.

Figure 24(a) gives an example of the most general type of statically decomposable stub, namely one with multiple entry and exit paths. The distinct edge crossings identify requirements for distinct start and end points in any associated plug-in. In accordance with the nature of UCMs, the continuity of end-to-end routes through stubs must be identified at the stub level. For example, Figure 24(b) indicates, with different line shadings, three permissible routes through the stub (S1-E1, S2-E1, and S3-E2).

The fact that a stub is represented by a shape does *not* indicate that a stub is a component in the component substrate—it is not. Stubbing is a path concept, not a component concept. Think of a stub as a kind of generalized responsibility that may be decomposed. Accordingly, the principles of stubbing are explained here with unbound maps. However stubs of bound maps may include components, providing a way of jointly decomposing path segments and components, and these components do become part of the component substrate (Section 4.3.1).

Plug-ins are described in separate diagrams and are bound to stubs by associating elements in the plug-in diagram (e.g., start points and end points) with elements in the main UCM diagram (e.g., path-stub edge crossings). Binding a plug-in to a stub is performed by the creator of the UCMs as part of UCM documentation. Binding identifies *operational substitutions* that may occur when the system is initialized (or, for self modifying systems, while it is running). In general, any plug-in may be bound to multiple stubs in different places. Multiple alternative plug-ins may be identified for the same stub, but, for static decomposition, only one would be bound (all could be bound to a dynamic stub, in which case, only one at a time would be operationally substituted).

The relationship between stubs and plug-ins for the case of static decomposition is summarized by Figure 25. For simplicity, no responsibilities are shown in the plug-in, but you should imagine that all paths have responsibilities along them. This is an arbitrary example in which a plug-in, bound to a single stub, provides synchronization for some, but not all, paths through the stub. The meaning of binding is indicated by visually superimposing labels of start and end points of the plug-in on the main UCM next to the stub edge crossings. Operational substitution means that start and end

points of the plug-in disappear, in effect, leaving the associated path segments of the plug-in directly joined to the path segments of the main map. Observe how the continuity of Figure 24 is preserved by Figure 25.

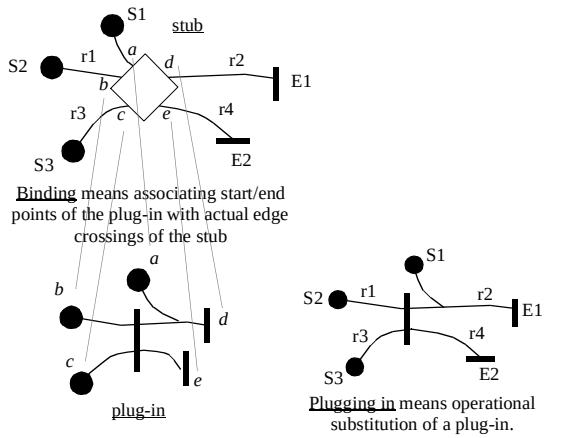


Figure 25 Path decomposition with stubs and plug-ins.

Any plug-in must have the requisite number of start and end points for binding to a stub and these are bound to a stub in a one-to-one fashion.

4.3.1 Components in Path Decomposition

The essence of the relationship between stubs, plug-ins and components is shown in Figure 26.

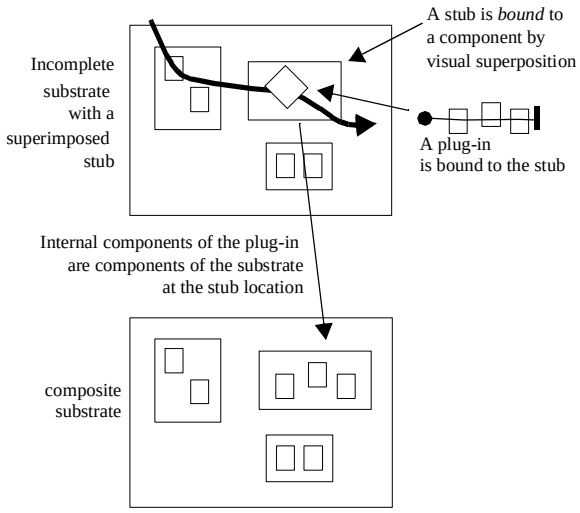


Figure 26 The effect of plug-ins on the component substrate.

Stubs are not components, meaning they are not part of the component substrate of a UCM. However, stubs are *bound* to components in the same way that responsibilities are, by visual superposition. The plug-ins of a bound stub may, in turn, contain components bound to its paths. These components *are* part of the component substrate.

4.3.2 Control of Stubs

Although UCMs do not define control arrangements among components (e.g., who calls whom), they give hints about control. For example, in Figure 27, binding

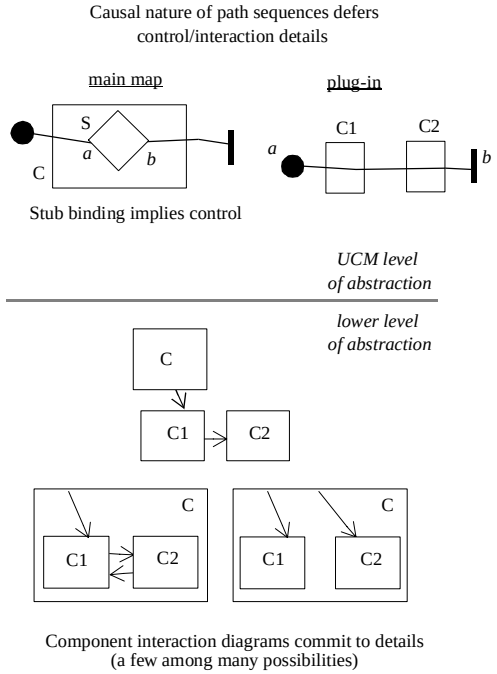


Figure 27 Implied control.

S to C implies that C controls the components of S in this location (let us call them C.C1 and C.C2). How this control will be exercised is undefined by the UCM and left to detailed design as described in Section 6.1 (C might send a message to C.C1 to start things off, leaving C.C1 and C.C2 to implement the rest of the path between themselves, but many other possibilities also exist, including C individually controlling C.C1 and C.C2).

4.4 UCM Layering

Layering enables a large scale system to be represented by a hierarchy of layers that are described independently, *without explicit reference to each other.* The concept is that lower layers provide infrastructure for higher ones. Layering provides a way of describing large scale systems with many levels of infrastructure without becoming bogged down in details of how infrastructure is used. If the infrastructure is not itself described with UCMs, and if its existence is implied clearly enough by the nature of components and relationships in UCMs, then relating UCM elements explicitly to the infrastructure may be deferred to detailed design (Section 6.1). Here we shall only be concerned with cases for which this is not so.

There are two cases of interest. **Case 1:** (Figure 28) An infrastructure layer is described by

UCMs and relationships must be made clear at the UCM level. **Case 2:** An infrastructure layer is not described by UCMs, but understanding UCMs requires the relationship between UCM elements and the infrastructure to be made clear. In either case, the relationships have the meaning of cross-references between otherwise-independent descriptions, not of "opening up" UCM elements to show layer details underneath (this would be decomposition, not layering). For Case 2, the most that can be done is to cross-reference UCM elements and layer elements (e.g., responsibility r is performed by layer component C). See Section 6.6 for ways of showing such things visually. Case 1 will now be explained.

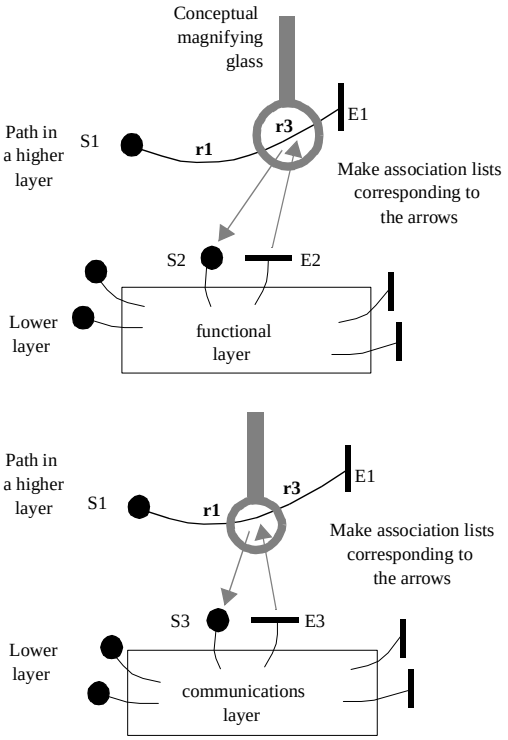


Figure 28 Documenting interlayer relationships.

Figure 28 positions a conceptual magnifying glass over two places where layering might occur, underneath a responsibility and underneath an apparently empty path segment. The conceptual magnifying glass symbolizes the existence of knowledge about layering that is not documented in the UCM itself.

Layering under responsibilities means that the responsibilities require functional infrastructure provided by the layer. Layering under apparently empty path segments means that the segments require communication infrastructure to advance causal sequences and move associated data.

A UCM for an underlying layer will have start and end points of paths for different purposes. Paths from higher layers detour through these paths. Detouring means diverting the main path to a specific layer start point and directing it back to the detour point from the corresponding layer end point. Documentation of this requires no more than listing associated points in the different maps (e.g., r3 is associated with S2 and E2, and the midpoint of the r1-r3 segment is associated with S3 and E3). The operational effect of detouring is a lot like positioning a stub at the place indicated by the conceptual magnifying glass, except that layer UCM is an actual separate part of the system, not a plug-in that is inserted at the magnifying glass location.

Detouring may have more subtle implications than Figure 28 suggests. Figure 29 provides three examples.

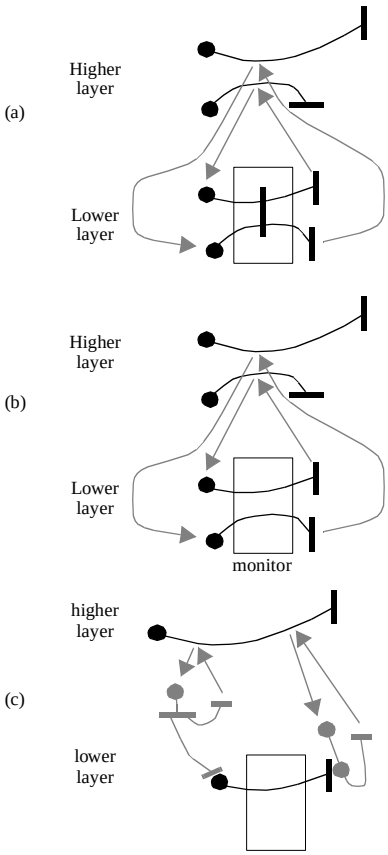


Figure 29 More complex interlayer situations.

Figure 29(a) shows that paths that appear to be uncoupled may be directly coupled in an underlying layer. Figure 29(b) shows that paths may be coupled through congestion at a shared layer component that has some restrictions on concurrent use (e.g., a monitor). Figure 29(c) shows that an underlying layer may be triggered to do its work in parallel with a higher layer and left to return results later, requiring some interlayer coupling machinery. The machinery of this example consists of an AND-fork to start the path through the layer and a path with a waiting place along it to pick up the result later. The AND-fork would be layered under a

responsibility that implicitly requests a service of the layer and the waiting path would be layered under a responsibility that implicitly gets the result. Other examples may be imagined, but these examples give the gist.

5 Self Modifying Systems

Section 1 introduced the concept of self-modifying systems and Section 3.2 and Section 3.4 provided examples. The UCM approach to self modifying systems centers around two types of dynamically pluggable elements: *Dynamic components (DCs)* are system components that may be created, stored, moved around, and destroyed as data; *dynamic plug-ins* are dynamically selectable sub-UCMs. These two types of dynamic elements have receptacles in UCMs called, respectively, *slots*, and *dynamic stubs*. The concept of pluggability for slots and dynamic stubs is a simple generalization of the concept of static decomposition developed in Section 4 for components and static stubs. The only new factor is that the receptacles (slots and dynamic stubs) are filled by system actions during normal system operation, making necessary additional UCM notations to indicate the system actions. Terminology is not uniform between somewhat analogous concepts (slots versus dynamic stubs, dynamic components versus plug-ins) because slots and stubs are different kinds of system entities (slots are components and stubs are generalized responsibilities).

5.1 Dynamically Pluggable Components

Slots and pools (Figure 30) identify *places* in the component substrate where dynamic components (DCs) display different *personalities*. DCs have dual personalities, as data (in *pools* and while moving from place to place), or as operational units (in *slots*). Only one of the personalities is normally manifest at one time. A DC may be stored for a while as data in a pool and then move across the system as data to become an operational unit in a slot. When a DC is in a pool, it is not operational (this is a manifestation of its data personality); it must be in a slot to be operational.

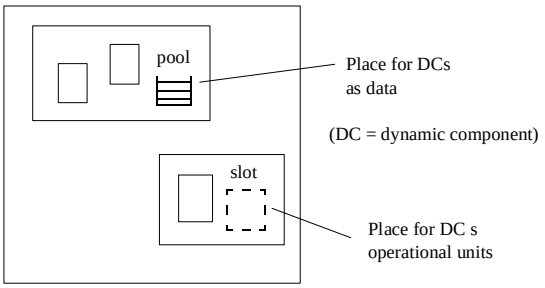


Figure 30 Notations for dynamically pluggable components.

Care must be taken to distinguish this usage of the term *pool* from other usages in software practice that imply sets of components that are already in an operational state, ready to provide service (the UCM representation for this would be a set of fixed components or slots).

At any particular time, a slot is either occupied or empty. When a slot is occupied, it is like a fixed component. When it is not occupied, it is no more than an empty hole in the substrate that cannot do anything (including cause itself to be occupied).

UCM paths are used to *imply* the creation, movement, and destruction of DCs in relation to pools and slots, without explicitly representing these things.

The UCM notation for manipulating DCs as data (Figure 31) will now be described. The concepts are as follows: paths are conduits (in an abstract, causal sense) for moving DCs, as data, from one place to another, and DCs move into or out of paths at pools or slots. The characterization of paths as conduits for DCs-as-data is not meant to be taken literally; paths are never more than traces of causal sequences, and characterizing them as conduits means only that we assume needed quantities will somehow be transported to where they are needed to enable causal sequences to be realized. This is no more than has been assumed before for ordinary data quantities.

The movement of DCs-as-data into or out of paths is indicated by small arrows perpendicular to the paths. The direction of these arrows indicates the local source or destination at those points (one is the path and the other is a pool or slot). No arrows *along* paths are needed, because paths are implicit conduits and the movement direction along the path is implied by the direction of the path. The small arrows are actually visual notations for special types of responsibilities that could be shown as ordinary responsibilities, except that this would not convey the meaning as graphically. Movement of DCs-as-data may be implied by positioning two or more of these responsibilities at different points along a path.

The responsibilities, which these arrows symbolize, are described next:

- **move:** Used for unaliased moves from a path or pool to a slot, or vice versa. *Unaliased* means the source forgets the component.
- **move-stay:** Used for aliased moves. *Aliased* means the source does not forget the component, so that the component ends up in more than one place at once (meaning it is visible in more than one place at once, in other words, is aliased). The *stay* part of the name refers to what happens at the tail of the arrow after the scenario moves on. For

example: after a *move-stay* of a DC into a slot, the DC stays in the path (in a conceptual sense); after a *move* of a component into a slot, the DC does not stay in the path; however, in both cases, the DC moves into and remains in the slot.

- **create** and **destroy:** The component is created *before* the move, or destroyed *after* the move. Initialization is assumed to be part of the create responsibility. These operations imply use of an underlying factory layer.
- **copy:** This is like move-stay, except that instead of moving the same component, a copy of it moves (so there is no aliasing). A copy is a new, distinct component that starts out with the same internal state as the original, but that may diverge from the original over time. Note that this notation does *not* mean the same thing as create followed by move-stay (the notation looks like it might mean that); create followed by move-stay requires use of both symbols, one after the other.

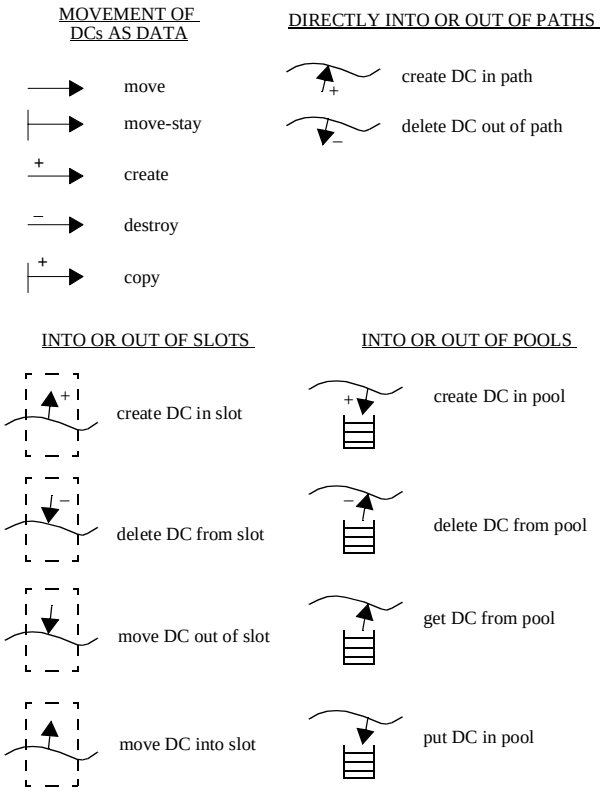


Figure 31 Movement notation (a visual notation for some special responsibilities).

The convention is to show pools to one side of paths because this makes for clearer diagrams, but the meaning is that the pool is a fixed component with responsibilities to get a component (as data) from storage or put it (as data) into storage (recall Figure 15 and Figure 16).

There is one new aspect to interpreting the small arrows as responsibilities: superimposing one of them on a slot does not indicate that the responsibility is performed by the slot. The implicit assumption (Figure 32) is that the responsibility belongs to whatever component forms the operational context for the slot—every slot has some component as an operational context, even if only implicitly in a particular UCM. Otherwise, binding of responsibilities to slots is the same as for any other component.

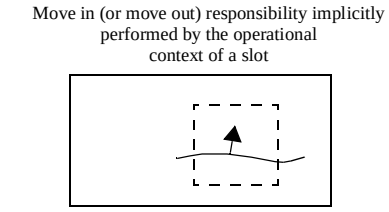


Figure 32 Move responsibilities are not performed by slots.

5.2 Dynamically Pluggable Path Stubs

Dynamically pluggable path stubs (Figure 33) provide the UCM means of expressing how a system may modify its own behavior structures. The concept is that the system selects from among alternative plug-ins for a stub at the last minute according to prevailing system conditions. To distinguish stubs used only for static decomposition from ones intended to be dynamically pluggable in this way, the latter are shown in dashed outline, like component slots. Figure 19 provided practical examples.

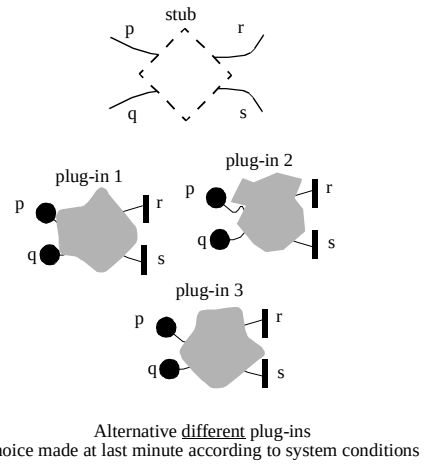


Figure 33 Concept of dynamically pluggable stubs.

To show how plug-ins may be created, selected and moved to where they are needed, create/move/destroy notations are needed, and also a pool notation. The notations of Section 5.1 may be borrowed, with slightly different interpretations and some constraints, considering the different nature of plug-ins and

stubs (Figure 34). Carrying over the complete compo-

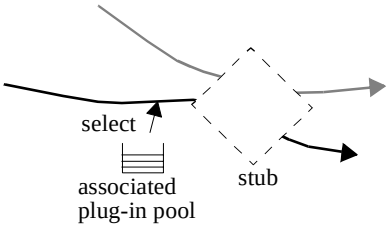


Figure 34 Selecting and moving plug-ins.

nent-movement model is neither required nor desirable. A restricted movement model is sufficient for the practical applications we have studied. In the restricted model, plug-ins never move directly from stub to stub; the source for plug-ins for a particular stub is always a local pool associated with the stub. The stub-pool association is a type of responsibility-component binding, because stubs are generalized responsibilities and pools are components (a dashed line could shown from the pool to the stub to indicate the relationship, but visual proximity seems sufficient in the cases we have studied).

In the restricted model, movement of plug-ins across a system may occur, but is constrained to be either from a pool to a pool or from a point of creation to a pool. Movement of a plug-in into or out of a stub is implied by a move arrow from the associated pool into the path before the stub (shown in Figure 34), or to the pool from the path after the stub (not shown). In some examples we have studied (e.g., Figure 19), only one path enters the stub and it always selects a new plug-in from the associated pool. In such cases, the selection and movement, and the pool itself, may be assumed to be implied by the path crossing the stub.

How such plug-in mechanisms would be implemented is less obvious than for the corresponding case with component slots. Software components are easily stored and moved around (literally or figuratively) in forms directly supported by programming languages, such as byte code and pointers. However, UCM plug-ins have no direct programming language representation. Some insight into the issues can be gained as follows. Imagine that the detailed design process (e.g., as suggested in Figure 35) has created different box-arrow diagrams for implementing different plug-ins. Then think of the implementation of a particular plug-in as setting up and exercising the appropriate boxes and arrows and imagine that, instead of hard-coding this, high level scripts are developed that can be interpreted to yield the same results, scripts that can be stored, retrieved and moved around. This is of practical interest for agent systems (Figure 19 and [10]).

6 Other Aspects of UCMs

6.1 From UCMs to Conventional Design Models

UCMs are able to defer detail because they rely on design models at lower levels of abstraction to supply it when the time comes (UCMs do not need the detail for their own purposes). An overview of how missing detail is intended to flow from UCMs is provided by Figure 35.

In Figure 35, “determine” means with the aid of human input; some of the detail flows directly from UCMs, some is implied by UCMs but requires human input to express, and some requires creative decisions. From a tool perspective, the process would be one of machine assistance, not machine translation. Design patterns, such as those in [11][22][23][24][25], may help in providing standard interfaces, interaction sequences, objects, and so forth. For object-oriented implementations, some objects at the detailed level may be the same as in the UCMs, except with missing details added to reflect detailed design decisions, and some may be new, for example, small scale objects may be added that were omitted from the UCMs as too fine-grained.

Edge crossings (where paths cross component edges) are important design elements in themselves because they provide the starting point for defining *interfaces* and associated *interaction protocols.* The UCM notation does not itself treat edge crossings as design elements, but transitioning from UCMs to more detailed levels of design requires them to be so treated. If components are decomposed, internal components will themselves have edge crossings that will become interfaces.

6.2 UCMs and Object-Orientation

UCMs are inherently object-oriented because they are concerned with systems of collaborating objects (in the most general sense of the term). An integrated high level view of object-oriented systems is provided by a combination of UCMs and high-level class-relationship diagrams (omitting methods and messages because UCMs are relied upon for behavior and functionality at this level of abstraction). See [5][6] for more.

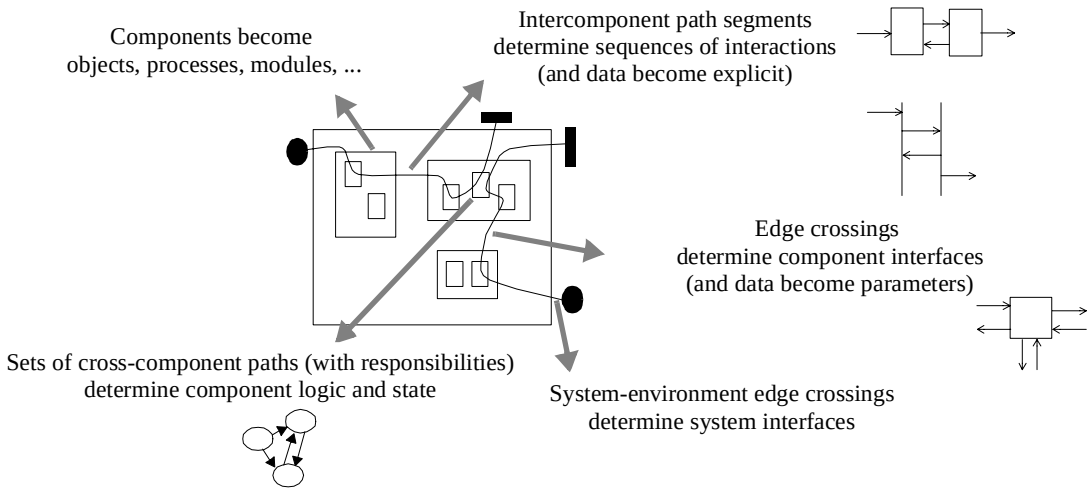


Figure 35 From UCMs to design models that are closer to code.

6.3 Diagramming Conventions

Because UCMs are intended to provide insight by presenting relationships visually, the style of presentation should be as regular, clean and uncluttered as possible. Stimuli of related types that follow the same path, perhaps with minor variations, should have the same start point, not be OR-joined to the path from different start points. Stimuli that come from the same source but carry information (e.g., data or commands) that requires them to follow different paths should have the same start point, with paths OR-forking after entering the system. Paths should cross as little as possible, and crossing points should be indicated by standard diagramming conventions (e.g., line breaks). Relative positions of the same components in different diagrams should not be changed. And so forth.

6.4 Documentation

UCMs will mean nothing to anyone but their creators without documentation, which must include at least the content outlined below. In the examples presented earlier in this paper, this content was given in unstructured prose. Documentation format is an important issue outside the scope of this paper.

Each scenario has a name, a short prose description, and an indication of which route is followed (or routes, for concurrent scenarios). Preconditions and postconditions of each scenario (and plug-in) are defined in prose, including whether or not a new scenario may start while one is still in progress. Start and end points are named to give clues to the nature of triggering stimuli and observed results at the end. Stimuli and responses are named, described, and associated with start and end points (in general, this may be a many-to-one association). Any input quantities coming from the source of the stimulus, or output quantities at the end, are named and described. Legal and illegal routes through multi-path UCMs are identified.

Responsibilities are named and explained by short, active prose statements that indicate their effect on the state of the component or the system (this must be informal, because these states are not defined by UCMs). Preconditions and postconditions may be defined for some responsibilities.

Cross-reference lists are provided of short-form names on maps and associated long-form names and/or brief characterizing phrases. Where appropriate, identification of associated classes or types is provided (e.g., types of parameters, classes of components). A name dictionary provides descriptions and cross-references. If a UCM is part of set of related UCMs, then relationships among the members of the set are identified.

6.5 Styles of Plug-ins

Two diagram styles for plug-ins are useful, distinguished by how they treat components. Figure 36 illustrates the two styles.

The *constrained-path style* requires paths and components of a plug-in to be confined within the area of the stub. In other words, paths of a plug-in cannot touch components of the UCM in which the stub is shown. This style was assumed in Figure 24-Figure 26. It provides for orderly, stepwise decomposition and recomposition of paths, but is too rigid for some purposes.

The *unconstrained-path style* allows a plug-in directly to extend its paths beyond the confines of a stub, to touch components of the UCM in which the stub is

located (but not components internal to other stubs). This type of stub is known as a *shared stub*, because its paths are shared between the area inside the stub location and the area outside it. Shared stubs are symbolized by an overlapping-diamond icon. Components in plug-in diagrams that belong to the larger UCM outside the stub are known as *anchored components* (suggesting they are anchored outside the plug-in). Anchored components are identified by cross hatching underneath, suggesting anchored to solid ground.

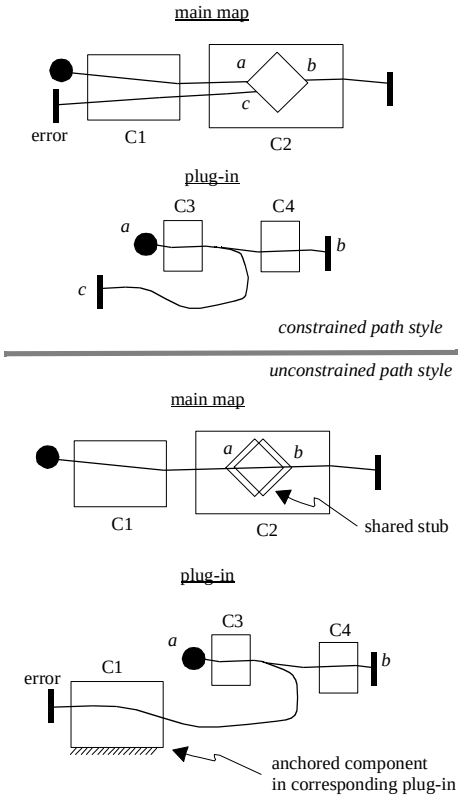


Figure 36 Different diagramming styles for plug-ins.

The unconstrained style enables a person to use plug-ins to define standard forms of paths that may traverse different anchored components in the same way in different places in the main UCM. It also enables main UCMs to be simplified by showing only the general trend of paths through a set of components, treating local meanderings among the same components as details deferred to plug-ins (for example, error paths). This provides considerable flexibility in filling in details, because stub edge crossings do not have to be defined first. However, it leaves the big picture incomplete, requiring it to be mentally pieced together from different plug-ins. Therefore this style can partially defeat the purpose of UCMs. In spite of this, the unconstrained-path style can be useful, when used with care. Both styles can be used to focus on different issues.

At the stub level, the unconstrained-path style means that paths may, in effect, extend outside the edges of the stub, without explicit edge crossings being defined.

Naming of anchored components in plug-ins depends on binding conventions that would be supported by a tool, but are not specific in the notation. For example, C2 in Figure 36 might be the actual name of an anchored component in the bigger picture, or it might be a formal name that can be bound to different anchored components associated with different stub locations.

6.6 Simplifying Paths

Situations in which complex path variations are required to express different scenarios for the same set of responsibilities, or many back-and-forth traversals are required between many components to express a single scenario may be indications that some judicious simplification is required.

A complex path may be product of trying to express algorithmic detail that properly belongs at a lower level of abstraction, such as localized looping or localized sequences of function calls. Then there is a simple solution: redefine the responsibilities to get rid of the path complexity, thus deferring the details to a lower level of abstraction.

However, situations may be deeper than that. Here are some useful tricks for dealing with such situations.

Figure 37 provides a notation for *shared responsibilities*. Each component is viewed as having a part of a single responsibility, r. In Figure 37(a), the square-wave symbol along the intercomponent path segment indicates that back-and-forth interactions—deferred to a lower level of abstraction—are required between the components *as part of the responsibility*, to enable the responsibility to complete (this notation is different from [5] because experience has shown it to be more expressive, but the concept is the same). The symbol indicating responsibility sharing may be duplicated for each responsibility, but it is simpler to show it only once along the shared path segment.

Shared responsibilities are useful to indicate negotiation: the first component is assumed to make a request, completing the request requires negotiating its parameters, and we want to indicate the *existence* of negotiation while deferring its details. Contrast Figure 8(a) with Figure 37(a): although intercomponent interactions may be required to implement the inter-component path segment of the earlier figure, any such interactions occur after the first component has completed r1 and before the second component has started r2.

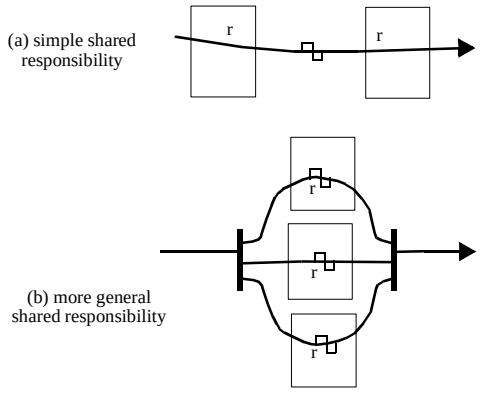


Figure 37 Shared responsibilities.

Figure 37(b) shows how the same notation may be used to indicate multi-component negotiations in which the order of component involvement is not predetermined.

Although these shorthands can be useful, they defer everything to a lower level of abstraction and so provide no clean way of associating a negotiation failure with a follow-up path to take alternative action.

Shared stubs are generalizations of shared responsibilities and have the same detail-hiding purpose, except they do it at the same level of abstraction. The difference between shared responsibilities and shared stubs is illustrated by Figure 38.

In Figure 38(a), suppose the shared responsibility r indicates a negotiation that must be performed *before* allocation of some resource is to be performed by the stub AR. The implication is that a message sequence is required between components C1 and C2 to complete r. Now suppose that the resource allocation must be part of the negotiation because the negotiation involves discovering whether or not the resource required to give the desired service can be allocated, and agreeing on a degraded service with a different resource if it cannot. One approach (not shown) is to put resource allocation into the part of r that is in C2 (thus leaving it to a lower level of abstraction).

Another approach, shown in Figure 38(b), is to replace the shared responsibility by a shared stub S between C1 and C2, the plug-in for which will include both negotiation and resource allocation in UCM terms. The possibility of negotiation may be indicated by an OR-fork to a return cycle.

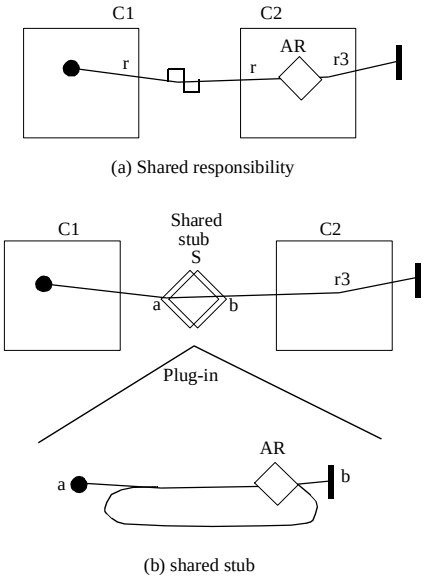


Figure 38 Shared responsibilities and stubs.

An implication of Figure 38(b) is that the plug-in contains C1 and C2 as anchored components for its start and end points (these do not have to be shown in the plug-in because the positioning of the stub in between the components implies it).

A situation analogous to Figure 37(b) may be represented with a shared stub along the path through each component. In this case, the plug-in would be shared, but the paths through it from each component could be different.

Complex paths may be avoided by layering. However, layering completely hides underlying functionality. A visual indication of what is hidden may sometimes be useful. Figure 39 shows how to indicate that different responsibilities use the same layer (perhaps a shared data base that is deliberately left implicit in the higher-layer UCM). Layered components made visible this way are not regarded as part of the UCM in which they are shown (meaning paths cannot cross them and responsibilities cannot be bound to them).

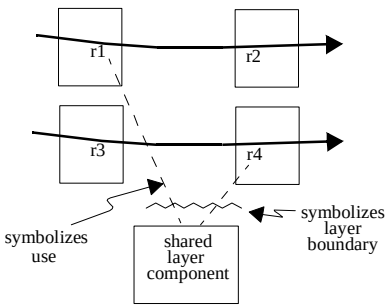


Figure 39 Making layering visible.

6.7 Nuances of Slots and Dynamic Stubs

Figure 40 shows that slots may be black boxes that can be opened up as glass boxes, just like fixed components (this was illustrated by the Gateway Server slot in Figure 17). In UCMs, DCs are never shown directly, either as black boxes or glass boxes, but only implied by slots and pools. Slot decomposition is a way of implying DC decomposition. A DC that may be in multiple slots would have the same decomposition in each slot, but might have different paths traversing its components in different slots, as appropriate for different roles. Fixed components in such decompositions (including pools) are fixed relative to the DC that occupies the slot. They are created at the time the DC is created and move with it into and out of pools and slots. Slots in such decompositions are initially empty and require paths traversing the primary slot to populate them. The DC that occupies the primary slot is not itself directly represented *in* the slot, because the concept is that the slot *becomes* the DC.

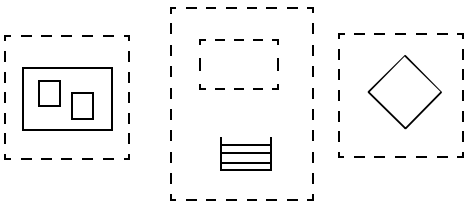


Figure 40 Slot decomposition.

The apparent inconvenience of not having a direct way defined for showing a single decomposition of a DC that may appear identically in many slots does not seem to be much of an inconvenience in practice, because experience indicates little requirement to do it.

Figure 40 also shows that a slot may have a stub bound to it. This provides more decomposition flexibility because different DCs with different stubs could be used in different slots, provided they are capable of filling the roles required by the slots. Dynamic stubs provide the same flexibility more simply because they do not require extra DCs (however, if the system design demands the extra DCs, then flexibility is not an issue).

What is the difference between a slot with a static stub bound to it (one of the options in Figure 40) and a dynamic stub? The answer is (Figure 41) that the former *is a* component and the latter *is not a* component, but is a type of generalized responsibility. Using a slot with a static stub to get the effect of a dynamic stub would introduce an extra component level, gratuitously.

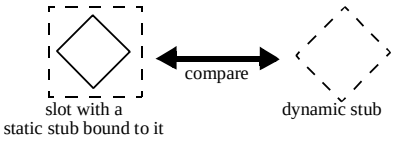


Figure 41 Nature of dynamic stubs.

6.8 Expressing Exceptions in UCMs

As illustrated by Figure 42, an exception-handler path may be indicated visually by a small zig-zag just after the start (this was used earlier for a specific case, namely timeout paths).

However, visually showing the coupling between such paths and the source of the exceptions tends to be impossible in practice because of the large number of possible points along source paths that may generate exceptions. The figure shows notations that allow us to talk about the issues but that are of limited usefulness in most practical situations the author has encountered. The most that can usually be done is to show exception paths originating in the environment (implicitly, they originate in an underlying system layer).

A scenario that generates an exception is viewed as coming to an abnormal end immediately after the error is observed. This is indicated visually by a diversion from the main path (shown by an OR-fork) that generates the error even and then ends.

Apart from the way they are triggered, handler paths are just ordinary paths that may wind through the system, aborting activity in progress and undoing damage by re-initializing fixed components, destroying old dynamic components and replacing them with new ones, and so forth. If a handler path specifically aborts activity along other paths, a special shorthand may be useful: a lightning-strike-shaped abort symbol, drawn between the paths. This is a shorthand for a handler path winding through all the components touched by the to-be-aborted path, restoring everything to some stable initial state.

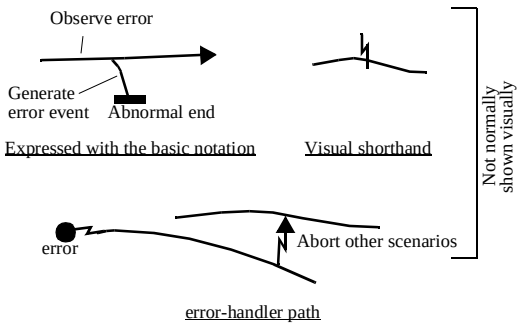


Figure 42 Exceptions.

7 Discussion

7.1 Properties of UCMs

The main important properties of UCMs are summarized below:

- Scenarios are represented as architectural entities that give a whole-system view of intertwined behavior and structure. This view of architecture goes beyond conventional views, such as [27], by including within its scope both organizational structures (UCM components) and behavior structures (sets of UCM paths). (From the perspective of this paper, software/system design notations, such as those of [3][31][14][16][26], are not considered architectural descriptions, but detailed design descriptions, so any comments about properties of architectural descriptions are not meant to apply to them.)
- Scenarios are not specified, only paths for scenarios are identified. This makes the notation more abstract and more useful in architectural terms than stand-alone scenario notations, such as [15][18], scenario notations embedded in software/system design notations, such as [3][31][14][16][26], or formal scenario specifications, such as [12].
- The notation directly represents, as architectural entities, large-scale units of emergent behavior cutting across systems, such as network transactions, that are difficult to come to grips with as whole-system entities in any other way.
- Causality is shown directly, avoiding the need to infer it from diagrams that express scenarios in terms of temporal sequences along timelines, such as those of [3][31][15][16][18][26].
- The parts of causal sequences threading through and between components are presented in a uniform way, as path segments, reducing the mental effort required to grasp the big picture. With other techniques, a person may have to piece together representations that are different in kind and often in different diagrams (e.g., *between components*, sequences are often viewed in terms of messages or calls; *inside components*, sequences are often viewed in terms of state transitions).
- The combination of behavior structures (wiggly lines) and organizational structures (arrangements of boxes) in the same diagram helps a person to understand intertwined behavior and structure directly without having to mentally integrate information from different diagrams.
- Because paths are continuous and notationally lightweight, many may be combined in a single diagram in a way that enables the mind's eye both to see them together and to sort them out. This also enables UCMs to give a more complete scenario picture than other techniques, in the sense of being able to include more scenarios without unreasonable effort.
- The notational elements stand back from the kinds of details that make practical systems so diverse, to focus on the high-level aspects that make them similar. The component notation takes the one-concept-fits-all view that a rectangular box represents any type of runtime component implemented in either software or hardware (a notation for a small number of different universal component types is provided in [5]).
- The approach expresses system self modification in a coherent way that scales up. It enables us to have our cake (describe self modifying systems) and eat it too (use fixed diagrams). It stands back from details such as software messages that request the creation and destruction of components, intercomputer messages that pass components from node to node in a network (e.g., as with Java applets), and messages of any kind that pass information to set up new relationships among components (e.g., to identify the components to be involved, and the required protocols). It includes a concept of behavior structures that may be used to describe self-modifying behavior entities, such as transactions.
- The notation provides a vehicle for back-of-the-envelope-style thinking and discussion during all phases of system development. Although this paper has not addressed all phases specifically, it has given enough information to make this statement at least plausible.
- UCMs may provide helpful visual patterns that stimulate thinking and discussion about system issues and that may be reused.
- The lightweight nature of the notation causes fewer problems with scaleup than more detailed notations and explicit techniques are provided to support scaleup. (This does not mean there can be no problems with scaleup. The ability of UCMs to give a quickly-understandable, end-to-end view can be compromised by overuse of techniques such as stubbing.)
- Although not covered by this paper, we are investigating whether and to what extent system performance requirements can be stated by attaching them to UCM paths (e.g., response time along a path, throughput along a set of paths). This offers the prospect of tying system behavior requirements to specific system functionality, rather than regarding them as non-functional requirements as is traditionally the case.
- UCMs are by no means a complete notation for all issues that arise in system development. They only deal with the issue of getting a high level view of emergent behavior of systems and are important

only because this is such a difficult view. UCMs supplement other techniques that give views of emergent behavior in more detailed terms, to give a different perspective. They neither replace such techniques nor depend on them.

- UCMs may be usefully integrated with other high-level techniques that have other purposes than behavior description, such as high-level class relationship diagrams in object-oriented design (see [5][6]).

7.2 What's in a Name?

The notation was originally given the name *timethreads*, which set it apart as a new notation, different from anything else, but seemed to require too much explanation to the uninitiated, especially considering that the name did not accurately reflect the central idea of causal sequences. The name was eventually changed to *use case maps* because the use of the by-then-widely-understood term *use case* reduced the amount of explaining required (this change was endorsed by the popularizer of the use-case approach, Ivar Jacobson, who provided a forward to the book that used the term for the first time [5]). This name was (and is) an accurate description of the notation as a way of projecting the scenarios of use cases onto solutions. Unfortunately, it fails to convey the idea of UCMs as architectural entities, thus tending to give the incorrect impression that UCMs are just another notation for use cases. Among other things, this paper has aims to correct that impression.

7.3 Related Work

UCMs come at system issues from such a novel angle that there are few directly related techniques known to the author.

As summarized in Section 7.1, well known notations for software *design* and scenario *specification* are more detailed or narrower in scope, or both, and therefore do not do the same job as UCMs. UCMs have something in common with the CRC method of object-oriented design [2], in the sense that both are centered around causal sequences of responsibilities, but the CRC method offers no visual notation to record its results and UCMs go deeper (e.g., Section 4 and Section 5). The concept of slots in UCMs is related to the concept of roles in the OORAM method [28] because slots are viewed as places where dynamic components may play different roles. Our idea of a first class characterization of behavior structures is closely related to Kiczales' idea of *emergent entities* in object-oriented programs. His group is working on an extension to object-oriented programming called *aspect-oriented programming* [17] that will express emergent entities directly. UCMs are a possible high-level design notation for aspect-oriented programming. The use of UCMs for adaptive systems has been explored in [29].

Aspects of UCMs have been made executable with LOTOS [1]. A formal XML syntax has been developed for use in a tool that covers most of what is in this paper [30], but it is evolving and somewhat tool-specific and so would be out of place in this paper.

8 Conclusions

This paper has presented an unconventional approach to scenarios, centering around the use of diagrams of scenario paths as visual *behavior structures* that can be created, manipulated, reused, and understood by people as first class architectural entities of a system. The diagrams are called Use Case Maps (UCMs) and their usefulness is broader than their name suggests. The aim of the approach is to leverage human understanding of the big picture during all phases of the life cycle, not just to specify scenario sequences. The paper has showed how the approach may be applied to systems that have some or all of the attributes of being concurrent, distributed, complex, large scale, and self modifying. The paper argues that UCMs merit being added to the suite of system description techniques for a number of reasons. They give a bird's eye view of whole-system behavior that is first class (meaning independent of how the behavior will, eventually, be made to emerge from details), uniquely compact and expressive for complex systems of all kinds, and general enough to supplement almost any other technique that defines system behavior in more conventional, detailed terms. The notation is lightweight enough to learn quickly and to be useful for sketching design ideas or explanations quickly in back-of-the-envelope fashion. A novel and powerful feature of the notation is the way it represents highly dynamic situations in a direct and understandable way. This paper provides an up-to-date, compact reference for all important features of UCMs, including interpretations and extensions that have emerged from recent experience.

Acknowledgments

Research into use case maps has been supported by TRIO (now CITO), NSERC, Mitel, and Nortel. Many students, coworkers, and collaborators have contributed to the development of these ideas and their application over the long term. I would particularly like to mention the following relatively recent ones (in alphabetical order): Daniel Amyot, Don Bailey, Francis Bordeleau, Jeromy Carriere, Ron Casselman, Mohammed Elammari, Tom Gray, Serge Mankovski, Andrew Miga, Yannick Morin, and many students in several deliveries of Carleton graduate course 94.586. I am grateful for ongo-

ing encouragement from (in alphabetical order) Don Cameron, Ivar Jacobson, Luigi Logrippo, Bran Selic, and Murray Woodside.

References

[1] D. Amyot, L. Logrippo, and R.J.A. Buhr, *Spécification et conception de systèmes communicants: une approche rigoureuse basée sur des scénarios d'usage*, Colloque Francophone sur l'Ingénierie des Protocoles CFIP'97, Liège, Belgique, September 1997. http://www.csi.uottawa.ca/~damyot/phd/cfip97/cfip97.html

[2] K. Beck, W. Cunningham, *A Laboratory for Teaching Object-Oriented Thinking*, ACM/SIGPLAN Proc. OOPSLA 89, New Orleans, Oct 1989.

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

[4] F. Bordeleau and R.J.A. Buhr, *The UCM-ROOM Design Method: from Use Case Maps to Communicating State Machines*, Proc. Conference on the Engineering of Computer-Based Systems, Monterey, March 1997. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/UCM-ROOM.ps.

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

[6] R.J.A. Buhr, *Understanding Macroscopic Behaviour Patterns in Object-Oriented Frameworks, with Use Case Maps*, chapter of a forthcoming Wiley book on OO Application Frameworks, ed. Mohamed Fayad. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/uoof.ps

[7] R.J.A. Buhr, *Use Case Maps for Attributing behavior to Architecture*, Proc. 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.

[8] R.J.A. Buhr, *Design Patterns at Different Scales*, Carleton report, presented at PLoP96, Allerton Park Illinois, Sep. 1996. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/plop.ps.

[9] R.J.A. Buhr, A. Hubbard, *Use Case Maps for Engineering Real Time and Distributed Computer Systems: A Case Study of an ACE-Framework Application*, Hawaii International Conference on System Sciences, Jan 7-10, 1997, Wailea. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/hicss.ps.

[10] 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 1998. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/4paam98.pdf.

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

[12] M. Glinz, *An Integrated Formal Model of Scenarios Based on Statecharts*, In W. Schaefer, P. Botella (eds): Software Engineering - ESEC95, Proceedings of the Fifth European Software Engineering Conference, Sitges, Spain. Lecture Notes on Computer Science 989, Springer, pp. 254-271.

[13] I. Graham, *In Search of the Three Best Books*, JOOP, September 1997, pp. 43-45.

[14] D. Harel, E. Gery, *Executable Object Modeling With Statecharts*, IEEE Computer, July 1997, pp. 31-42.

[15] ITU (formerly CCITT), *Recommendation Z.120: Message Sequence Charts (MSC)*, Geneva, 1996.

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

[17] G. Kiczales, OOPSLA97 invited talk, Aspect Oriented Programming. See http://www.parc.xerox.com/aop for talk notes and papers.

[18] 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.

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

[20] 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.

[21] 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

[22] 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

[23] 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.

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

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

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

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

[28] Trygve Reenskaug, *Working with Objects*, Manning Publications.

[29] Tuomas Ihme, *Adaptive Scenarios and Component-Based Embedded Software*, Object Magazine Online, July 1997, http://www.sigs.com/omo/tips/9707/9707.ihme.html

[30] UCM syntax description, in http://www.sce.caleton.ca/rads/agents

[31] *Unified Modeling Language (UML) Notation Guide*, version 1.1 alpha 6 (1.1 c), 21 July 1997, ftp://ftp.omg.org/pub/docs/ad/97-07-08.ps