Use Case Maps and UML for Complex Software-Driven Systems

Daniel Amyot

School of Information Technology and Engineering, University of Ottawa
150 Louis-Pasteur, Ottawa, Ontario, K1N 6N5, Canada
damyot@site.uottawa.ca http://www.UseCaseMaps.org

Abstract. The Use Case Map (UCM) notation allows the description of complex software-driven systems in terms of high-level causal scenarios. By superimposing scenario paths on a structure of abstract components, UCMs provide an integrated view of behavior and structure at the system level. This paper presents interesting features of UCMs in relation with several types of diagrams defined in UML. It also shows how UCMs can bind scenarios and structures at the architectural level, how they help visualizing dynamic systems, how they enable architectural reasoning, and how they help bridging the conceptual gap between use cases and sequence, activity, and statechart diagrams.

1 Introduction

Complex software-driven systems are of many kinds, including object-oriented, agent-based, real time, and distributed systems. They are characterized by many of the following attributes, which make them difficult to understand both in terms of technical and management complexity: large scale, concurrency, decentralized control, timeliness, dependability, diverse and feature-rich functionality, fluidity of run-time organization, and evolutionary requirements [12][25]. Such systems are often encountered in the areas of telecommunications, defense, aerospace, and industrial control [6].

The Unified Modeling Language (UML) is a general-purpose modeling language for specifying, visualizing, constructing and documenting the artifacts of software systems (in particular object-oriented and component-based systems), as well as for business modeling and other non-software systems [27]. It includes many concepts and notations useful for the description and documentation of multiple models, and it enjoys a strong support from academic and industrial communities.

An important feature of UML, *use cases* are defined as sequences of actions a system performs that yield observable results of value to a particular user (actor) [27]. Notations for scenarios and use cases, as well as design processes based on them, have become very popular over the last few years [16][30]. For instance, the Rational Unified Process is a methodology based on UML that is *use-case driven*, i.e., where use cases bind together five types of models (requirements, analysis, design, implementation, and testing) [22]. These models describe partial representations of the system. UML 1.3 allows the description of complex software-driven systems and models through the use of nine different diagram techniques. Each diagram provides a view of a model from the aspect of a particular stakeholder, and each diagram must be seman-

Submitted to UML'99.

tically consistent with all the others. In this paper, these diagrams are categorized into two sets. The first set, called *behavioral diagrams*, focuses mainly of functional and dynamic aspects of systems. It is comprised of five types of UML diagrams:

- **Use case diagrams**: Show actors and use cases together with their relationships. They describe system functionalities from the user's point of view.
- **Sequence diagrams**: Describe patterns of interaction among objects, arranged in a chronological order. They originate from Message Sequence Charts [20].
- **Collaboration diagrams**: Show generic structure and interaction behavior of the system.
- **Statechart diagrams**: Show the state space of a given context, the events that cause the transitions of one state to another, and the actions that result.
- **Activity diagrams**: Capture the dynamic behavior of a system in terms of operations. They focus on flows driven by internal processing.

The second set, called *structural diagrams*, relates more to components and static characteristics of systems. It includes these four types of UML diagrams:

- **Class diagrams**: Capture the vocabulary of a system. They show the entities in a system and their general relationships.
- **Object diagrams**: Snapshots of a running system. They show object instances (with data values) and their relationships at some point in time.
- **Component diagrams**: Show the dependencies among software components.
- **Deployment diagrams**: Show the configuration of run-time processing elements and the software components, processes, and objects that live on them.

UML includes several implicit links between these two sets of diagrams (e.g., sequence and collaboration diagrams can use the entities defined in class diagrams). However, UML does not emphasize any first-class and compact way of describing large-scale units of behavior that emerge from the collective efforts of many system components (e.g., transactions spanning a network) [12].

This paper describes a diagraming technique called *Use Case Maps* (UCMs) [8] as a means to link behavior and structure in an explicit and visual way. UCMs are first-class architectural entities that describe *causal relationships* between *responsibilities*, which are bound to underlying organizational structures of abstract *components*. This paper attempts to illustrate how UCMs can help bridging the conceptual gap between the use cases in the use case model and other behavioral diagrams (sequence, statechart, and activity) in the analysis and design models. At the same time, UCM provide a bird's-eye view of activities from behavioral diagrams allocated to organizations of components (and objects) in structural diagrams. This enables architectural reasoning throughout the evolution of a system design.

Although this paper does not intend to be a tutorial on the UCM notation (the interested reader is invited to consult [8][11][13][28] for further details), it illustrates several UCM features of potential interest to the UML community. Section 2 provides an overview of the notation, through a simple telephony system example, and its relation to use cases. Sections 3 and 4 present several relations between UCMs and, respectively, behavioral diagrams and structural diagrams. Recent and future delevopments related to UCMs are discussed in Section 5, then a conclusion follows.

2 Use Case Maps

2.1 Basics of the Notation

Use Case Maps are used to emphasize the most relevant, interesting, and critical functionalities of a system. Responsibilities along causal paths can be internal to a component or be observable. UCMs can represent specific scenarios, or else be abstract (generic) and cover multiple scenario instances. With UCMs, scenarios are expressed above the level of messages exchanged between components, hence they are not necessarily bound to a specific organizational structure. UCMs provide a path-centric view of system functionalities and improve the level of reusability of scenarios.

Figure 1(d) shows a simple UCM where a user (Alice) attempts to establish a telephone call with another user (Bob) through some network of agents. Each user has an agent responsible for managing subscribed telephony features such as Originating Call Screening (OCS). Alice first sends a connection request (req) to the network through her agent. This request causes the called agent to verify (vrfy) whether the called party is idle or busy (conditions are italicized). If he is, then there will be some status update (upd) and a ring signal will be activated on Bob's side (ring). Otherwise, a message stating that Bob is not available will be prepared (mb) and sent back to Alice (msg).

A scenario starts with a triggering event and/or a pre-condition (filled circle labeled req) and ends with one or more resulting events and/or post-conditions (bars), in our case ring or msg. We call *route* a path that links a cause to an effect. Intermediate responsibilities (vrfy, upd, mb) have been activated along the way. Think of responsibilities as tasks or computational functions to be performed. In this example, the responsibilities are allocated to abstract components (boxes Alice, AgentA, Bob and AgentB), which could be seen as objects, processes, agents, databases, or even roles, actors, or persons. We call such superposition a *bound map*.

The construction of a UCM can be done in many ways. For example, one may start by identifying the responsibilities (Figure 1(a)), although not necessarily with diagrams like this one. They can then be allocated to scenarios (Figure 1(b)) or to components (Figure 1(c)). Components can be discovered along the way. Eventually, the two views are merged to form a bound map (Figure 1(d)).

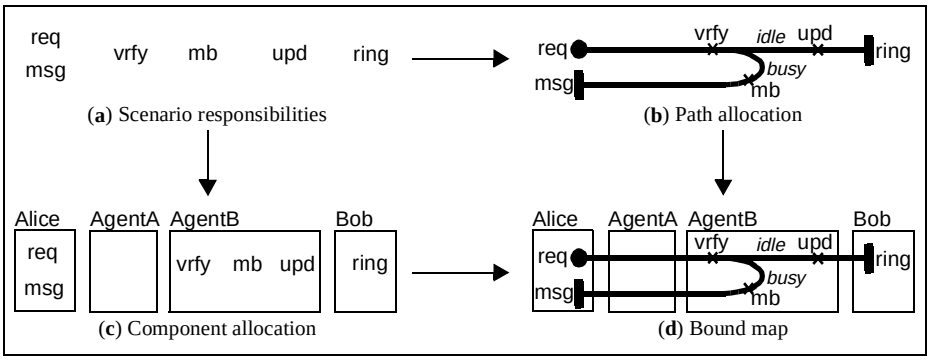


Fig. 1. Use Case Maps construction.

Figure 1(d) is quite a simple diagram, yet it conveys a lot of information in a compact form, and it allows for requirements engineers and designers to use two dimensions (structure and behavior) to evaluate architectural alternatives for their system.

2.2 Additional Notation

To introduce further notation elements, new features can be added to this basic use case. Figure 2 abstracts from the component instances introduced in Figure 1. The components do not refer to Bob and Alice any longer, but they refer to more generic call origination and termination roles (for both users and agents). Dashed components are called *slots* and may be populated with different instances at different times. They can represent roles of a particular class of components. In [8][11], Buhr introduces an architectural notation with different types of components and richer semantics (active processes, passive objects, groupings, pools of objects, interrupt service requests, agents, mutex, etc.). He also discusses how to relate such components to classes of objects. Since their definition would be outside the scope of this paper, the next examples will provide only but a few intuitive descriptions. The nature of the components involved here does not really impact the UCM features emphasized in this paper.

The middle part of Figure 2 shows an enhanced version of the UCM in Figure 1(d) that represents a whole class of related use case instances. It is referred to as the *root map* because this UCM possesses containers (called *stubs*) for sub-maps (called *plug-ins*). Stubs are of two kinds:

- **Static stubs**: represented as plain diamonds (see stub ST), they contain only one plug-in, hence enabling hierarchical decomposition of complex maps.

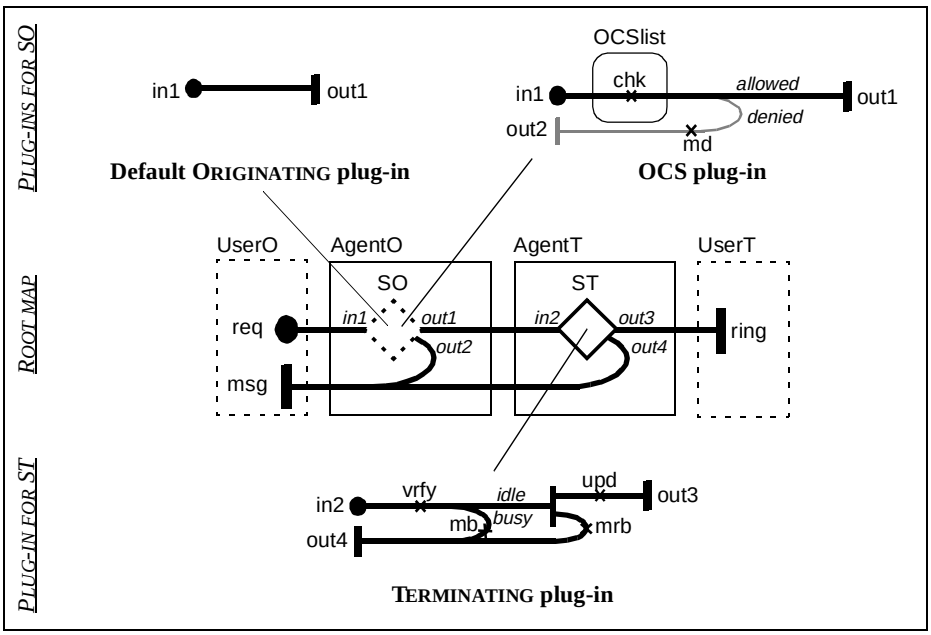


Fig. 2. More complex call connection and new notation elements.

- **Dynamic stubs**: represented as dashed diamonds (see stub SO), they may contain several plug-ins, whose selection can be determined at run-time according to a *selection policy* (often described with pre-conditions). It is also possible to select multiple plug-ins at once (sequentially or in parallel), although the composition then requires to be detailed outside the UCM diagram.

Path segments coming in and going out of stubs have been identified on the root map (italicized labels). Although they are not required to be shown visually, their presence helps to achieve unambiguous bindings of plug-ins to stubs. For instance, the originating dynamic stub SO has two plug-ins (ORIGINATING and OCS). The start point of the ORIGINATING plug-in (in1) is bound to the incoming path segment *in1*, and the end point out1 is bound to the outgoing segment *out1*. Figure 2 makes use of similar labels for a clear binding relation between plug-ins and stubs, but in general names are different and the relation has to be described explicitly.

The OCS plug-in shows a new component (the passive object OCSlist) that represents a list of screened numbers that the originating user (UserO) is forbidden to contact. If UserO is subscribed to the Originating Call Screening service, then the OCS plug-in is selected instead of the ORIGINATING plug-in. In this case, the called number is checked against the list (chk). If the call is denied, a relevant message is prepared to be sent back to the originating party (md).

The TERMINATING plug-in improves on the original UCM by allowing the update (upd) and the ring result to be accompanied, concurrently, by the preparation of a ring-back signal to be prepared and sent back to the originating party (mrb). Concurrency is represented here by an AND-fork. The notation allows for alternative paths (OR-fork and OR-join, as in the TERMINATING plug-in), concurrent paths (AND-fork and AND-join), shared responsibilities, exception paths, timers, failure points, error handling, and (a)synchronous interactions between paths, to name but a few elements.

By selecting plug-ins for the stubs in the integrated view, one can obtain a flattened map, which still contains multiple possible end-to-end scenarios. Once stubs are defined at key points on a path, it becomes easy to add new plug-ins, which could represent new features in our example. Existing maps and plug-ins can further be decomposed or extended (e.g., when a radically different service is added) with new paths and new static and dynamic stubs.

2.3 UCMs and Use Cases in the Puzzle

A UML use case defines a set of use case instances, which are sequences of actions a system performs that yield observable results of value to a particular actor [22]. Use cases are usually described in plain text, although this representation can on occasion be substituted by other behavior description techniques such as activity diagrams, statecharts, or pre/post-conditions. When describing the interactions between the system and the relevant external actors, a use case generally considers the system as a black box where the internals are not shown. There exists a large conceptual gap between use cases and their realization in terms of behavioral diagrams where the system's internals are refined with sub-components. Reasoning about this gap and the big picture using the current UML diagrams is often puzzling since much mental effort is required to integrate many details from many diagrams of different styles.

Figure 3 (adapted from [12]) presents the current UML pieces of the puzzle, and Use Case Maps as the missing piece. UCMs reduce the effort required to put the other pieces together and understand the big picture. UCMs should not be seen solely as an extra step but more importantly as a rational, *gray-box*, and traceable progression from use cases, where the focus is on system behavior (black-box), to more detailed behavioral diagrams, where the focus is eventually put on component behavior (glass-box). We use the term "gray-box" to represent that some design information is visible.

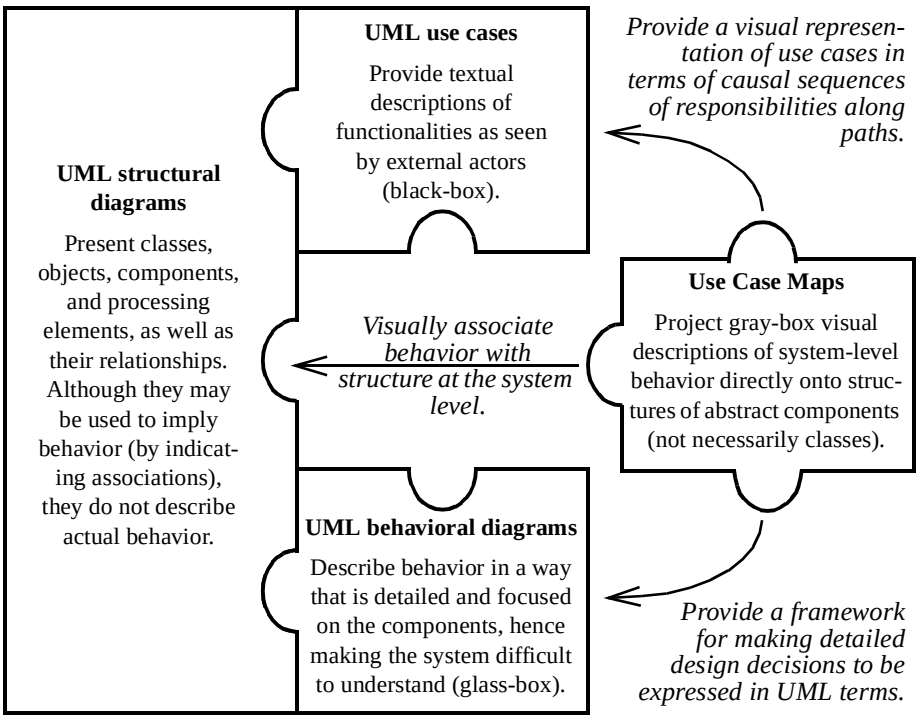


Fig. 3. UCMs as a missing piece of the puzzle.

In addition, the UCM notation contains features for expressing dynamic situations that span whole systems in a compact form. Firstly, dynamic organizations of components can be expressed using slots, pools, and dynamic responsibilities (not discussed here) at the system level, while abstracting from code construction and deployment aspects. Secondly, time-varying scenario patterns are representable using stubs and plug-ins, as shown in Figure 2.

This paper does not claim that bridging the aforementioned conceptual gap or that expressing dynamic situations is impossible with current UML diagrams and processes. However, Use Case Maps represent a unique perspective that seems advantageous for solving this puzzle and visualizing the big picture. The next sections further illustrate several of these advantages in terms of existing UML diagrams.

3 UCMs and Behavioral Diagrams

3.1 UCMs and Use Case Diagrams

Use case diagrams show actors and use cases together with their relationships. They are particularly relevant to capture functional requirements or existing functionalities in the use case model, but they can be used in other types of models as well.

UML use cases are black-box descriptions, while UCMs are more gray-box as they show some of the details inside the system (e.g., topology of abstract components, internal flow of actions, etc.). Like use cases, UCMs use signals, events, or messages when communicating with actors outside the system (especially at start points and end points), while they may use other communication semantics when communicating with elements inside the system. No premature decision that would overspecify the system should be taken at this level of abstraction. These decisions are left for other models that make use of more appropriate notations, e.g., sequence diagrams.

UML use case diagrams have access to three types of relationships between use cases, namely *include*, *extend*, and *generalization* [22]. To a great extent, the UCM notation appears comprehensive enough to represent, in a compact way, use cases as well as these relationships:

Include relationship: its purpose is to help clarify a use case by isolating and encapsulating complex details (so they do not obscure the real meaning of the use case), and by improving consistency (by factoring behavior included in several base use cases).

Inclusion can be achieved by placing a static stub on the path of a base use case. This stub hides the details contained in its plug-in (the inclusion use case), and the plug-in can be reused in multiple stubs, hence improving consistency among the UCMs. The location of the inclusion point is stated visually on the path, and many static stubs can be used to represent multiple inclusions. For example, see stub ST and plug-in TERMINATING in Figure 2.

Extend relationship: the goal of extensions is to show that part of a use case is (potentially) optional, that a subflow is executed only under certain (sometimes exceptional) conditions, or that there may be a set of behavior segments of which one or several may be inserted at an *extension point* in a base use case.

This relationship can be expressed in UCM terms with OR-forks, which may have more than two (possibly guarded) alternatives. The *denied* path of the OCS plug-in and the *busy* path of the TERMINATING plug-in (Figure 2) are both extensions of their respective base use cases. Using visual hints like path labeling, color, shading, or thickness, UCMs can emphasize the original base case (to distinguish the basic flow of events from the alternative or exceptional ones), which otherwise could be lost into the details. As an illustration, the *allowed* path of the OCS plug-in (in bold) represents the base case, as opposed to the *denied* path which is the extension. The UCM notation also provides other visual clues for exceptional, time-out, and error-handling paths.

Dynamic stubs represent another level of extension relationships. Such stubs may have a default behavior (a plug-in that often contains an empty path), which can be overridden by other plug-ins. The conditions under which a plug-in other than the

default one is chosen are described in the selection policy. For instance, the stub SO in Figure 2 has a default plug-in (ORIGINATING) whose selection will be overridden in favor of the OCS plug-in when the subscriber's OCS feature is active.

UML use cases explicitly define extension points where additional behavior can be added. There is no such concept in UCMs, as any path segment is an implicit point of potential extension (e.g., for an OR-fork), except perhaps for dynamic stubs, which are explicit extension points.

Use Case Generalization: generalization is used when two or more use cases have commonalities in behavior, structure, and purpose. The shared part can then be described in a new *parent* use case specialized by *child* use cases.

UCM scenarios that share common segments and purposes can be integrated together with a combination of OR-forks and OR-joins, or more likely with multiple dynamic stubs. The parent UCM represents the common parts in the original use cases, and it contains dynamic stubs for the parts where the behaviors diverge (the latter become plug-ins). A child UCM is constituted of a parent UCM whose stubs are occupied by the appropriate plug-ins. However, generalization from multiple parents (multiple inheritance) would require the parent UCMs to be integrated together before defining the plug-ins and how child UCMs would use them.

As an example, a *Basic Call UCM* could be represented as a flattened version of the root map of Figure 2 where the default ORIGINATING plug-in occupies stub SO and TERMINATING occupies ST. An *OCS Call UCM* would however use the OCS plug-in in stub SO. Both the Basic Call and the OCS Call would be child UCMs of their parent UCM (the root map), whose structure and behavior has been inherited and modified.

3.2 UCMs and Interaction Diagrams

UML defines two types of interaction diagrams. *Sequence diagrams* show the explicit sequence of stimuli along vertical time axis (called lifelines) and are better for real-time specifications and for complex scenarios. *Collaboration diagrams* show the relationships among instances and are better for understanding all of the effects on a given instance and for procedural design. They essentially cover similar concepts, but in different forms. This section focuses mainly on sequence diagrams.

Use Case Maps can help deriving interaction diagrams (in the analysis and design models) from use cases (in the use case model). UCMs do not explicitly define message exchanges between components, but messages need to be constructed in such a way that the causal relationships between responsibilities from different components are satisfied. There are usually many ways to do so, depending on the available interfaces, communication channels, and protocols.

The causal path <req, vrfy, upd, ring>, which represents the base use base extracted from the UCM in Figure 1(d), will serve as an example. In Figure 4(a), this sequence is bound to the same component substrate, to which explicit communication channels (lines) have been added, hence constraining the potential senders and receivers of each message. Different decisions about the protocols and control can lead to multiple solutions.

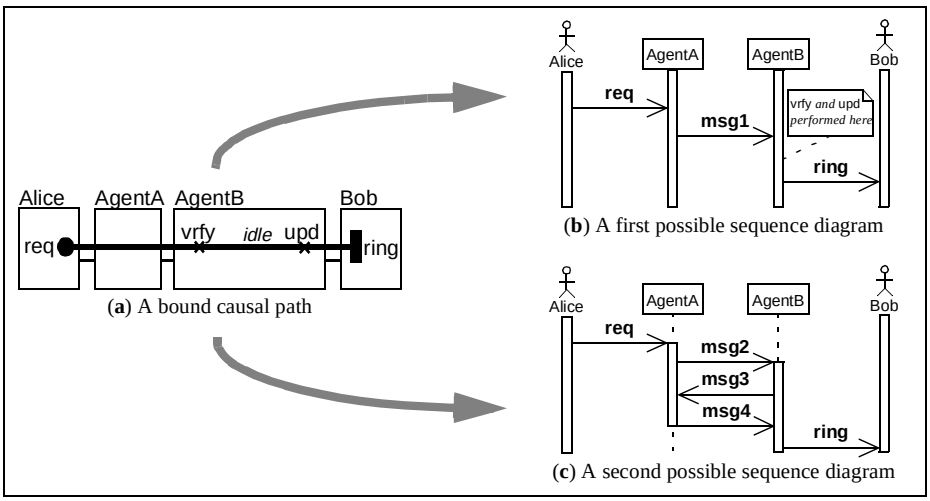


Fig. 4. Admissible sequence diagrams derived from a UCM path.

Figure 4(b) shows a situation where the four concurrent entities communicate through simple protocols, resulting in straightforward exchanges of messages. However, if a more complex protocol is used between the two agents (e.g., a negotiation), and if the control is attributed differently, then Figure 4(c) might be an admissible sequence diagram derived from the same causal path. Whichever is the most appropriate depends on design decisions that are not taken at the UCM level, but which needs to be documented in the appropriate model for a better traceability.

Several papers have illustrated the derivation of valid Message Sequence Charts (MSCs) from Use Case Maps [1][4][5]. Basic MSCs are similar in nature to sequence diagrams. In [7], the authors introduce a mapping of UCMs to High-Level MSCs [20], a notation that allows the recursive structuring of MSCs with constructs for sequence, concurrency, alternative, iteration, and others (basic MSCs do have similar constructs, but for messages, not sub-MSCs).

3.3 UCMs and Statechart Diagrams

This state machine formalism is an object-based variant of Harel's Statecharts [15]. It incorporates several concepts similar to those defined in ROOMcharts, a variant of statecharts defined in the ROOM modeling language [24].

With statechart diagrams, the focus definitely becomes component behavior. UCMs do not replace these diagrams, but they can guide their construction. Paths segments from (possibly many) UCM scenarios that are bound to a component need to be integrated together to determine the component logic and states. At the same time, the synthesis needs to cover the causal relationships between responsibilities in different components that are refined in terms of message exchanges. Path segments that cross component boundaries also help describing component interfaces. Statechart diagrams may be influenced by the available classes of objects defined (possibly independently)

in the class diagram. There is a mapping required between the UCM abstract components and the objects, processes, and modules for which state machines are constructed. Again, this synthesis procedure can result in many valid solutions, hence the design decisions need to be motivated (possibly by requirements outside the UCMs) and documented in the appropriate models.

Figure 5(a) presents the UCM paths crossing the component AgentB from Figure 1(d). A potential statechart for this path is illustrated in Figure 5(b), where responsibilities, guards, and messages have been mapped respectively to states, conditional transitions, and plain transitions. This particular example assumes that agents have their own threads and are initially awaiting a specific message. Obviously, different assumptions and requirements will lead to different statechart diagrams.

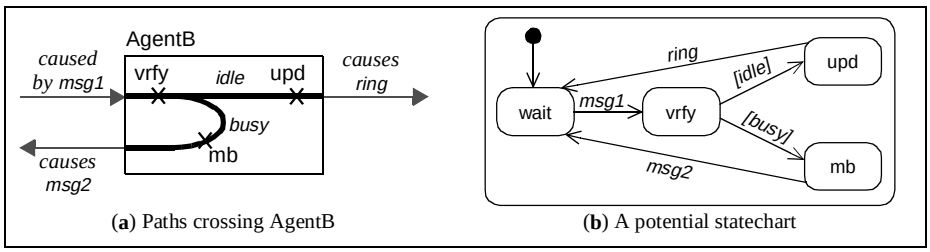


Fig. 5. Potential statechart diagram derived from UCM paths

The mapping from UCM paths to statechart diagrams is not always so straightforward. For instance, the components of Figure 2 would require more complex statecharts in order to integrate multiple plug-ins (AgentO), to integrate multiple path sources and destinations (AgentO), and to cover concurrent paths (AgentT).

Moving directly from Use Case Maps to state machines usually represents a big step. Often, sequence diagrams can be used as an intermediate step. Decisions related to the refinement of inter-component causal relationships would then be made at the sequence diagram level. State machine still need to integrate these sequences together in order to cover the different roles played by each component.

A method that generates communicating finite state machines (ROOMcharts) from UCMs is presented in [20]. High-level MSCs are used as an intermediary step. UCMs can also lead to other types of communicating entities. In [1][2][5], the authors use the formal algebraic language LOTOS [18] to model the component-based behavior of the system, while agent behaviors (for high-level prototypes) are generated in [9][10].

3.4 UCMs and Activity Diagrams

An activity diagram is a special case of a state diagram whose purpose is to focus on flows driven by internal processing (as opposed to external events in ordinary statechart diagrams), hence it is essentially used to represent the state machine of a procedure or a business workflow. Activity diagrams focus more on sequences of actions and on conditions than on the components performing those actions.

Activity diagrams share many concepts (and even notation elements) with basic Use Case Maps. UCM responsibilities are similar to activities. Both notations support

sequences of actions, as well as alternatives and concurrency. Start points and end points also have similar purposes.

A complex activity may be refined into another activity diagram, just like UCMs use static stubs for path decomposition. However, stubs appear to be more generic; they allow for multiple incoming and outgoing paths, and dynamic stubs permits the use of many plug-ins (refinements) whose selection is based on some policy. UCM stubs proved to be a very useful artifact for expressing dynamic behaviors and structures in complex systems.

One of the strengths of UCMs resides in their ability to bind responsibilities to components. Activity diagrams are usually not used in that way, although they support such mapping to a limited extent. An activity diagram may be divided visually into *swimlanes*, each separated from neighboring swimlanes by vertical solid lines on both sides. Each swimlane represents responsibility for part of the overall activity, and may eventually be implemented by one or more objects. Each action is assigned to one swimlane, and transitions may cross lanes. There is no significance to the routing of a transition path. Swimlanes can be interpreted as components in their simplest form; they are one-dimentional and do not show in any way how components relate to each other (e.g., by their relative position, or their very nature). UCMs provide an integrated bird's-eye view that includes this information. Such a view is almost essential for understanding how behavior, represented as paths and (dynamic) responsibilities, affect and modify the run-time structure of components in dynamic systems.

4 UCMs and Structural Diagrams

4.1 UCMs and Component-Based Diagrams

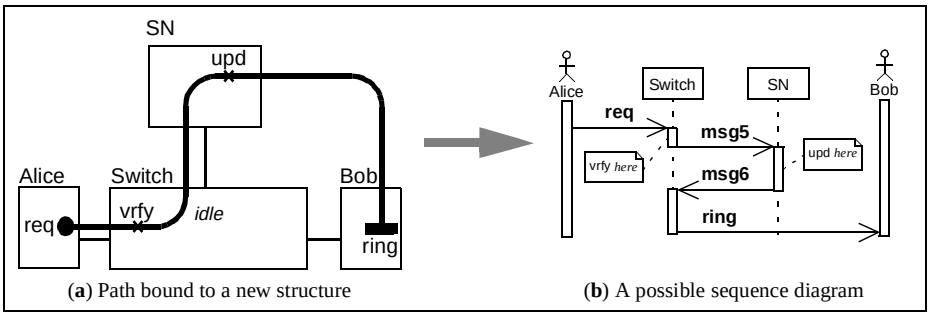
A UML *component diagram* is a graph of components connected by dependency relationships. Components may also be connected to components by physical containment representing composition relationships. UML *deployment diagrams* show the configuration of run-time processing elements and the software components, processes, and objects that live on them. UML *object diagrams* present snapshots of running systems in terms of object instances and their relationships at a particular moment in time.

The default UCM component notation, as defined by Buhr and Casselman in [8], is abstract enough to represent many important aspects found in these three types of UML structural diagrams. They can illustrate containment and other dependencies, be of different types (passive, active, etc.), and even represent run-time instances (without data). However, their main strength resides in the ability to describe dynamic structures with static diagrams. With the help of slots, pools (of components) and UCM paths with dynamic responsibilities, components can be created or destroyed, moved around, made visible to other components, and so forth. UCMs with such components can express, in an apparent static and concise way, complex dynamic issues that would otherwise need to be stated with many snapshots of UML component-based diagrams.

This does not prevent one from using another structural notation underneath UCM paths. Such paths can be used on top of several types of component-based diagrams such as the ones in UML or the like.

4.2 Architectural Reasoning with UCMs

Use Case Maps allow for the early evaluation of architectural alternatives by acting as a link between function (use cases) and form (structure). By decoupling the functions from the structure, one can play on one aspect concurrently as well as independently of the other. The previous UCMs illustrated different paths on the same structure of components. UCMs also enable one to reuse the same paths on different alternative structures. For example, Figure 6(a) reuses the same causal path as Figure 4(a), but on a different set of components. Here, no communicating agents are involved. Instead, the responsibilities are allocated to more traditional telephony components such as a Switch and a service node (SN), with different dependencies (e.g., communication links). This will in turn lead to yet another different set of valid sequence diagrams and different state machines further down the design cycle.



(**a**) Path bound to a new structure (**b**) A possible sequence diagram

Fig. 6. Admissible sequence diagrams derived from a UCM path.

As they can easily be decoupled from structures, UCM paths improve the reusability of scenarios and lead to behavior patterns that can be utilized across a wide range of applications. On many occasions, UCMs may provide helpful visual patterns that stimulate thinking and discussion about system issues and that may be reused [11].

Note also that the evaluation of architectural alternatives is done at a high level of abstraction, without any early commitment to messages and protocols as in sequence diagrams. Such diagrams require more efforts that could be wasted when the underlying structure is modified.

Architectural reasoning also needs to cope with evolving system requirements. Complex systems are seldom built from scratch. Instead, they evolve to accept new technology and to accept new features. As shown by Velthuijsen [29], the addition of new features is non-monotonic; they can and will change the operation of existing functionalities. New technology can also change the assumptions on which functionalities are based. Use Case Maps provide mechanisms (e.g., stubs and plug-ins) for handling the non-monotonic nature of system evolution. Moreover, it has been shown how UCM practice can distinguish between chained *decomposition* (e.g., small scale objects, threads, processes, modules, packages, etc.) and *layering* (e.g., operating systems, communication stacks, network middleware, etc.) [8][11][12]. The distinction between these two architectural concepts help coping, among other things, with the scalability and the maintainability of systems.

5 Discussion

5.1 Semantics and Tools for UCMs

The semantics of Use Case Maps and well-formedness rules are defined in terms of hypergraphs [21]. A textual linear form for UCMs, expressed in XML [31], has also been defined [3]. This form is suitable for input to different tools and for generating documentation. Having this XML Document Type Definition also enables an easier integration of UCMs with upcoming standards for UML such as the *XML Metadata Interchange* (XMI) [17] and the *UML eXchange Format* (UXF) [26].

The *UCM Navigator*, a tool for constructing and editing Use Case Maps, makes use of the hypergraph semantics and rules to provide sound transformations that ensure the construction of correct maps [21]. This tool supports the path notation and Buhr's component notation, and it uses the XML form as its file format. Nested stubs and plug-ins can be created, responsibilities can be easily bound to components, notation extensions for agent systems and performance modeling are supported, documentation in PDF-enabled PostScript is generated, and the tool is available for X11R5 on three platforms (Linux, Solaris, and HP-UX).

5.2 UCMs for Formal Validation and Verification

Although UCMs possess a semi-formal semantics, they can be used to guide the generation of more formal models and specifications for complex systems. Over the years, much work has been done on the derivation of LOTOS specifications [18] from UCMs [1][5]. LOTOS is an algebraic language that can formalize the ordering of events found in UCMs, even in the absence of a component structure. This enables formal verification and verification of requirements, specifications, and designs, something that lacks from many (OO) case tools. Other target formalisms include ROOM [7][24] and soon SDL [2][19]. Note also that there exists recent work on how ROOM models can be used to implement LOTOS specifications [1][14].

UCMs are currently used in several projects, some of which addressing issues related to dynamic agencies [9][10] (where UCMs proved to be strong at describing complex agent relationships), the avoidance and detection of undesirable interactions between telephony features [5][9][10], the generation of functional test suites, and the description of standards for emerging mobile telephony services [1][2][4].

Performance modeling is yet another application of UCMs. In [23], performance becomes a property of paths, rather than a non-functional property of a whole system, as it is usually considered to be. The notation was extended to include timestamps, time constraints, event distributions, associations of processes and tasks to devices, etc. Both the XML form and the UCM Navigator support these extensions. Integrating other types of non-functional requirements is also under study.

5.3 Integrating UCMs and UML

UML is intended to be broadly applicable without extensions, because they might not be universally understood, supported, and agreed upon. Instead, UML *profiles* provide a standard way to use UML in a particular area without having to extend or modify UML [27]. A profile is a predefined set of stereotypes, tagged values, constraints, and

notation icons that collectively specialize and tailor UML for a specific domain or process (e.g., Objectory Process profile and Business Engineering profile). A profile does not extend UML by adding any new basic concepts. Instead, it provides conventions for applying and specializing standard UML to a particular environment or domain.

Integrating the UCM concepts in UML could be achieved to some extent by tailoring an appropriate profile. Although this would not require any modification to the UML standard, many of the most interesting UCM concepts would not be easily covered with current UML diagrams and semantics.

A second and obvious option would consist in adding the UCM notation to the set of UML diagrams. Although this looks simple and sound, this would also add to the redundancy that already exists among a somewhat large collection of UML diagrams.

The extension of existing diagramming techniques (e.g., activity diagrams) and semantics to support original UCM concepts could represent a third option.

Finally, the substitution (or the reorganization) of one or more UML diagrams by UCMs may also be considered as a potential option, which might however be difficult to realize due to the existing (legacy) investment in the current standard and tools.

The best and most appropriate option is still a research topic. Nevertheless, it seems important that standardization of these UCM concepts be achieved, independently of the selected option. UML is certainly an excellent candidate where such standardization could occur in the near future.

6 Conclusion

Use Case Maps relate very much to existing UML diagramming techniques, yet they help filling the conceptual (gray-box) gap that exists between use cases and behavioral diagrams. They also represent an interesting viewpoint for architectural reasoning, particularly in the context of complex and dynamic software-driven systems where the behavior emerging from multiple components is often difficult to visualize.

This paper illustrated some of the most important concepts behind the UCM notation and usage. UCMs establish a useful linkage between behavioral diagrams and structural diagrams at the system level, while allowing people to work independently on these two dimensions. Architectural reasoning is promoted early in the design cycle through the use of stubs, plug-ins, and dynamic components. Unbound UCM paths become reusable scenario patterns that can be bound to multiple underlying component structures. Though they are defined at an abstraction level higher than that of exchanges of messages, UCM can guide the generation of detailed diagrams (e.g., sequence diagrams and statechart diagrams) and even formal specifications.

As the UCM notation becomes used in different projects, it becomes more stable and robust. Tools started to emerge, and a UCM User Group was initiated at the beginning of the year [28]. UML could benefit from many concepts found in UCMs. The best place for this piece of the puzzle however still remains to be clarified.

Acknowledgements. I am grateful to Ray Buhr and Luigi Logrippo for numerous discussions on UCMs over the last six years. Many thanks to Tom Gray and Darcy Quesnel for their useful comments on an earlier draft. This work was supported in part by FCAR, NSERC, and CITO.

References

[1] Amyot, D., Hart, N., Logrippo, L., and Forhan, P.: "Formal Specification and Validation using a Scenario-Based Approach: The GPRS Group-Call Example". In: *ObjecTime Workshop on Research in OO Real-Time Modeling*, Kanata, Canada, January 1998. http://www.csi.uottawa.ca/~damyot/wrroom98/wrroom98.pdf

[2] Amyot, D., Andrade, R., Logrippo, L., Sincennes, J., and Yi, Z.: "Formal Methods for Mobility Standards". In: *IEEE 1999 Emerging Technology Symposium on Wireless Communications & Systems*, Dallas, USA, April 1999. http://www.UseCaseMaps.org/UseCaseMaps/pub/ets99.pdf

[3] Amyot, D. and Miga, A.: *Use Case Maps Linear Form in XML, version 0.12*, April 1999. http://www.UseCaseMaps.org/UseCaseMaps/xml/

[4] Amyot, D. and Andrade, R.: "Description of Wireless Intelligent Network Services with Use Case Maps". In: *SBRC'99, 17th Brazilian Symposium on Computer Networks*, Salvador, Brazil, May 1999. http://www.UseCaseMaps.org/UseCaseMaps/pub/sbrc99.pdf

[5] Amyot, D., Buhr, R.J.A., Gray, T., and Logrippo, L.: "Use Case Maps for the Capture and Validation of Distributed Systems Requirements". In: *ISRE'99, Fourth International Symposium on Requirements Engineering*, Limerick, Ireland, June 1999. http://www.UseCaseMaps.org/UseCaseMaps/pub/re99.pdf

[6] Booch, G.: *Software Architecture and the UML*. Slide package, Rational Software, 1998. http://www.rational.com/uml/img/arch.zip

[7] Bordeleau, F. and Buhr, R.J.A.: "The UCM-ROOM Design Method: from Use Case Maps to Communicating State Machines". *Conference on the Engineering of Computer-Based Systems*, Monterey, USA, March 1997. http://www.UseCaseMaps.org/UseCaseMaps/pub/UCM-ROOM.pdf

[8] Buhr, R.J.A. and Casselman, R.S.: *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, USA, 1995. http://www.UseCaseMaps.org/UseCaseMaps/pub/UCM_book95.pdf

[9] Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S.: "High Level, Multi-agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Example". In: H.S. Nwana and D.T. Ndumu (Eds), *PAAM'98, Third Conference on Practical Application of Intelligent Agents and Multi-Agents*, London, UK, March 1998, 277-295. http://www.UseCaseMaps.org/UseCaseMaps/pub/4paam98.pdf.

[10] Buhr, R.J.A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S.: "Feature-Interaction Visualization and Resolution in an Agent Environment". In: K. Kimbler and L. G. Bouma (Eds), *Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98)*, Lund, Sweden, September 1998. IOS Press, 135-149. http://www.UseCaseMaps.org/UseCaseMaps/pub/fiw98.pdf.

[11] Buhr, R.J.A.: "Use Case Maps as Architectural Entities for Complex Systems". In: *Transactions on Software Engineering*, IEEE, December 1998, pp. 1131-1155. http://www.UseCaseMaps.org/UseCaseMaps/pub/tse98final.pdf

[12] Buhr, R.J.A.: "Use Case Maps and UML". Slide package, Carleton University, December 1998. http://www.UseCaseMaps.org/UseCaseMaps/pub/ucm_umlSlides98.pdf

[13] Buhr, R.J.A.: "Making Behaviour a Concrete Architectural Concept". In: *32nd Annual Hawaii International Conference on System Sciences (HICSS'99)*, Hawaii, USA, January 1999. http://www.UseCaseMaps.org/UseCaseMaps/pub/hicss99.pdf

[14] Hart, N.: *Protocol Validation and Implementation: A Design Methodology Using LOTOS and ROOM*. M.Sc. thesis, SITE, University of Ottawa, Canada, April 1999.

[15] Harel, D. and Gery, E.: "Executable Object Modeling with Statecharts". In: *Proceedings of the 18th International Conference on Software Engineering*, Berlin, IEEE Press, March, 1996, pp. 246-257.

[16] Hurlbut, R. R.: *Managing Domain Architecture Evolution Through Adaptive Use Case and Business Rule Models*. Ph.D. thesis, Illinois Institute of Technology, Chigago, USA. http:// www.iit.edu/~rhurlbut/hurl98.pdf

[17] IBM, Unisys et al.: *XMI (XML Metadata Interchange) Proposal*, OMG document ad/98-10-05, October 1998. http://www.software.ibm.com/ad/features/xmi.html

[18] ISO, Information Processing Systems, Open Systems Interconnection: *LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, IS 8807 (1989).

[19] ITU: *Recommendation Z.100, Specification and Description Language (SDL)*. Geneva, 1994.

[20] ITU: *Recommendation Z. 120: Message Sequence Chart (MSC)*. ITU, Geneva, 1996.

[21] Miga, A.: *Application of Use Case Maps to System Design with Tool Support*. M.Eng. thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1998. http://www.UseCaseMaps.org/UseCaseMaps/ucmnav/

[22] Rational Software: *Rational Unified Process 5.0*, Cupertino, CA, 1998.

[23] Scratchley, C. and Miga, A., "A Use Case Map Editing Tool with Performance Prediction Capabilities". In: *ObjecTime Workshop on Research in OO Real-Time Modeling*, Kanata, Canada, January 1998.

[24] Selic, B., Gullekson, G., and Ward, P.T.: *Real-Time Object-Oriented Modeling*, Wiley & Sons, 1994.

[25] Selic, B. and Rumbaugh, J.: *Using UML for Modeling Complex Real-Time Systems*. White paper, ObjecTime Ltd., March 1998. http://www.ObjecTime.com/otl/technical/umlrt.pdf

[26] Suzuki, J. and Yamamoto, Y.: "Making UML Models Exchangeable over the Internet with XML: UXF approach". In: *Proceedings of the First International Conference on the Unified Modeling Language (UML '98)*, Mulhouse France, June, 1998. http://www.yy.cs.keio.ac.jp/~suzuki/project/uxf/index.html

[27] UML Revision Task Force: *OMG Unified Modeling Language Specification, version 1.3 beta R1*, April 1999. http://uml.systemhouse.mci.com/

[28] *Use Case Maps Web Page* and *UCM Users Group*, 1999. http://www.UseCaseMaps.org

[29] Velthuijsen, H.: "Issues of non-monotonicity in feature-interaction detection". In: *Third Interntional Workshop on Feature Interactions in Telecommunications Software Systems*, Kyoto, Japan, October 1995.

[30] Weidenhaupt, K., Pohl, K., Jarke, Matthias, and Haumer, P.: "Scenarios in System Development: Current Practice". In: *IEEE Software*, March/April 1998, 34-45.

[31] W3 Consortium: *Extensible Markup Language (XML) 1.0*. W3C Recommendation, 10 February 1998. http://www.w3.org/TR/REC-xml