On the Extension of UML with Use Case Maps Concepts

Daniel Amyot[1,2] and Gunter Mussbacher[2]

[1] SITE, University of Ottawa, 150 Louis-Pasteur, Ottawa (ON), Canada, K1N 6N5
[2] Mitel Corporation, 350 Legget Dr., Kanata (ON), Canada, K2K 2W7
damyot@site.uottawa.ca | gunter_mussbacher@Mitel.com

Abstract. Descriptions of reactive systems focus heavily on behavioral aspects, often in terms of scenarios. To cope with the increasing complexity of services provided by these systems, behavioral aspects need to be handled early in the design process with flexible and concise notations as well as expressive concepts. UML offers different notations and concepts that can help describe such services. However, several necessary concepts appear to be absent from UML, but present in the Use Case Map (UCM) scenario notation. In particular, Use Case Maps allow scenarios to be mapped to different architectures composed of various component types. The notation supports structured and incremental development of complex scenarios at a high level of abstraction, as well as their integration. UCMs specify variations of run-time behavior and scenario structures through sub-maps "pluggable" into placeholders called stubs. This paper presents how UCM concepts could be used to extend the semantics and notations of UML for the modeling of complex reactive systems. Adding a "UCM view" to the existing UML views can help bridging the gap separating requirements and use cases from more detailed views (e.g. expressed with interaction diagrams and statechart diagrams). Examples from telecommunications systems are given and a corresponding design trajectory is also suggested.

1 Introduction

The modeling of reactive (event-driven) systems requires an early emphasis on behavioral aspects such as interactions between the system and the external world (including the users), on the cause-to-effect relationships among these interactions, and on intermediate activities performed by the system. Scenarios are particularly good at representing such aspects so that various stakeholders can understand them.

Owing to their distributed and critical nature, telecommunications systems are representative of complex reactive systems. Emerging telecommunications services require industries and standardization bodies (ANSI, ETSI, ISO, ITU, TIA, etc.) to describe and design increasingly complex functionalities, architectures, and protocols. This is especially true of wireless systems, where the mobility of users and of terminals brings an additional dimension of complexity. Recent and upcoming technologies based on agents and IP, which involve complex and sometimes unpredictable policy-driven negotiations between communicating entities, also raise new modeling issues as protocols and entities become more dynamic in nature and evolve at run time.

The design and standardization of telecommunication systems and services results from a design process frequently comprised of three major stages. At stage 1, services are first described from the user's point of view in prose form, with use cases, and with tables. The focus of the second stage is on control flows between the different entities involved, represented using sequence diagrams or *Message Sequence Charts* — MSC [13]. Finally, stage 3 aims to provide (informal) specifications of protocols and procedures. Formal specifications are sometimes provided (e.g. in SDL [11]), but they are still of marginal use [1]. ITU-T developed this three-stage methodology two decades ago to describe services and protocols for ISDN. Naturally, such descriptions emphasize the reactive and behavioral nature of telecommunications systems. In this methodology, scenarios are often used as a means to model system functionalities and interactions between the entities such that different stakeholders may understand their general intent as well as technical details.

1.1 Requirements for a Scenario Notation

Due to the inherent complexity and scale of emerging telecommunications systems, special attention has to be brought to the early stages of the design process. The focus should be on system and functional views rather than on details belonging to a lower level of abstraction, or to later stages in this process. Many ITU-T members recognize the need to improve this process in order to cope with the new realities cited above. In particular, Study Group 10, which is responsible for the evolution of standards such as MSC, SDL, and TTCN, recently approved a new research question that could lead to a new recommendation by 2003 [14].

This research question will focus on what notation may be developed to complement MSCs, SDL and UML in capturing user requirements in the early stages of design when very little design detail is available. Such notation should be able to describe user requirement scenarios without any reference to states, messages or system components. Reusability of scenarios across a wide range of architectures is needed with allocation of scenario responsibilities to architectural components. The notation should enable simpler modeling of dynamic systems, early performance analysis at the requirements level, and early detection of undesirable interactions among services or scenarios.

While UML activity diagrams provide some capability in this area [17], a notation with dynamic (run-time) refinement capability and better allocation of scenario responsibilities to architectural components is required.

1.2 Extending UML with Use Case Maps Concepts

Use Case Maps (UCMs) [7][8] visually represent causal scenarios combined with structures. UCMs show related use cases in a map-like diagram. They have a history of applications to the description of reactive systems of different natures (e.g. [1][2][3][9]), to the avoidance and detection of undesirable interactions between scenarios or services (e.g. [2][9][16]) and to early performance analysis (e.g. [21]). A more extensive bibliography can be found in the *UCM User Group* library [23].

The addition of several useful concepts found in the UCM notation to UML would make the latter a more appealing tool for designing reactive systems. UCMs have a

number of properties that satisfy many of the requirements described in Section 1.1: scenarios can be mapped to different architectures, variations of run-time behavior and structures can be expressed, and scenarios can be structured and integrated incrementally in a way that facilitates the early detection of undesirable interactions. This paper presents how UCM concepts could be used to extend the semantics and notations of UML for the modeling of complex reactive systems. Adding a "UCM view" to the existing UML views can also help bridging the gap separating requirements and use cases (e.g. as found in stage 1 of the ITU-T methodology) from more detailed views expressed with interaction diagrams (stage 2) and statechart diagrams (stage 3).

Through examples from the telecommunications domain, this paper illustrates how UCM concepts can be used to extend UML semantics and notations in the modeling of reactive systems. Section 2 introduces Use Case Maps and defines its core concepts in terms of the UML semantic metamodel (version 1.3). Section 3 shows how UCMs combine behavioral scenarios and structures in a single view. The benefits of capturing dynamic run-time behavior are illustrated in Section 4. Section 5 goes beyond the core concepts and presents other potential benefits offered by UCMs, together with a UCM/UML design trajectory suitable for reactive systems.

2 Use Case Maps Core Concepts

The Use Case Maps notation is based on various core concepts. This section focuses on a subset of these concepts and links them to the existing UML semantic metamodel. Advanced concepts more specific to UCMs, which will lead to suggestions on how to improve the UML metamodel and notations, are discussed in Sections 3 and 4.

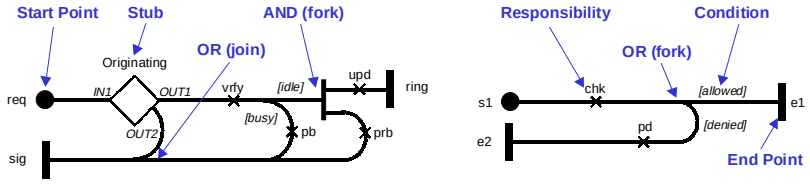
2.1 Overview of the UCM Notation

Use Case Maps are used as a visual notation for describing *causal relationships* between *responsibilities* bound to underlying organizational structures of abstract *components*. Responsibilities are generic and can represent actions, activities, operations, tasks to perform, and so on. Components are also generic and can represent software entities (objects, processes, databases, servers, etc.) as well as non-software entities (e.g. actors). The relationships are said to be causal because they involve concurrency and partial orderings of activities and because they link causes (e.g., preconditions and triggering events) to effects (e.g. postconditions and resulting events).

UCMs are useful for describing telecommunications services at an early stage in the design cycle, even when no component is involved. For example, Fig. 1(a) describes the basic call process of a simplified telecommunications system, the *Tiny Telephone System* (TTS), accompanied by the *Originating Call Screening* (OCS) feature in Fig. 1(b).

This map contains many of the elements that form the core of the UCM notation. Informally, a start point (filled circle) is where scenarios are caused, upon the arrival of a triggering event and/or the satisfaction of preconditions. The scenario effect is represented as an end point (bar), which describes some resulting event and/or postconditions. Responsibilities are represented as crosses. UCM paths, which connect the different elements discussed so far and show the progression of a scenario along a use

case, may fork and join in different ways. OR-forks represent alternative paths, which may be guarded by conditions. AND-forks (narrow bars) represent concurrent paths along which the scenario evolves. OR-joins are used to merge common paths whereas AND-joins (not shown here) are used to synchronize concurrent paths. The diamonds represent stubs and act as containers for sub-maps called plug-ins. For instance, the OCS map in Fig. 1(b) could be plugged in the `Originating` stub of the Basic Call map. This requires a *binding relationship* that would specify how the start and end points of the plug-in map would be connected to the path segments going into or coming out of the stub. In this example, the binding relationship is `{<IN1, s1>, <OUT1, e1>, <OUT2, e2>}`.



a) Basic Call map b) *OCS* plug-in map

Fig. 1 Tiny Telephone System example, with one feature.

According to the UCM notation and semantics, the TTS scenarios are interpreted in the following way. Upon the arrival of a call request (`req`), the OCS feature checks whether the call should be allowed or denied (`chk`). If denied, then a call denied reply is prepared (`pd`) and signaled (`sig`). If allowed, then the system verifies whether the called party is busy or idle (`vrfy`). If busy, then a busy reply is prepared (`pb`) and signaled. If idle, then a ringback reply is prepared (`prb`) and signaled (`sig`) concurrently with the update of the system status (`upd`) and a resulting ringing event (`ring`).

This example illustrates how UCM descriptions abstract from messages, data and control, while focusing on general causal flows between causes, responsibilities, and effects. UCMs are neither dataflow diagrams nor control flow diagrams (where control is often associated to procedure calls or method invocations)

2.2 Current UCM Semantics

The UCM abstract syntax and static semantics are currently based on a graph structure (more specifically a hypergraph) and described in a XML document type definition [4]. The concrete syntax is visual and was introduced in the previous section. All of these are supported by a visual editing tool, the *UCM Navigator* [15], together with a set of valid transformations that ensure the satisfaction of well-formed rules. Dynamic semantics is yet informal, although one has been indirectly provided in terms of the formal language LOTOS [10], whose underlying semantics is based on labeled transition systems, on CCS and on CSP [2][3].

However, the main semantic concepts of Use Case Maps are hidden behind various details and implementation-related concerns. Fig. 2 captures, in general terms and independently of the current hypergraph-based semantics, the UCM core concepts in the form of a class diagram. This diagram abstracts from many class attributes, relationships, and constraints in order to focus on the essence of what concepts should be preserved in the context of an integration with UML.

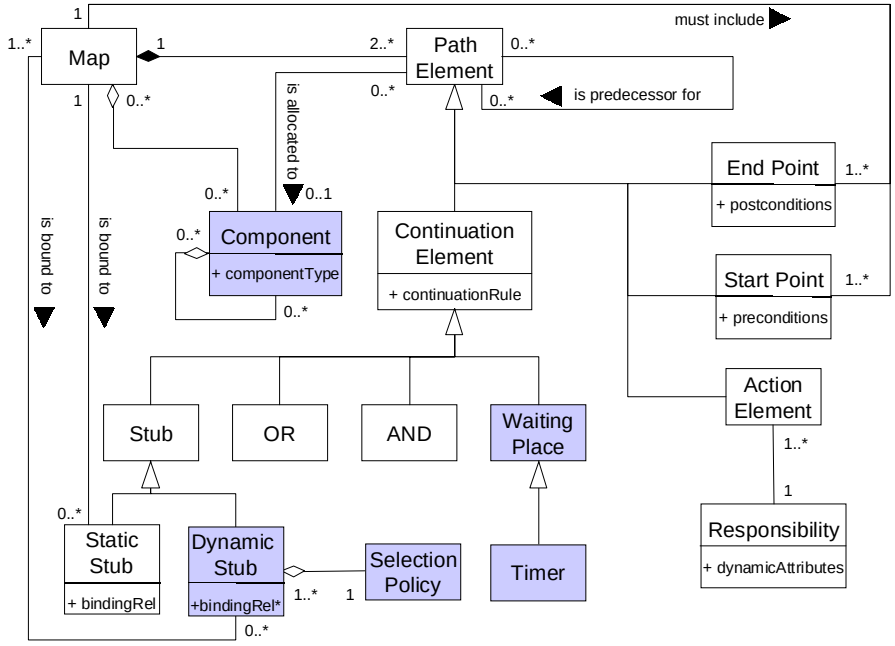


Fig. 2 Overview of UCM core concepts.

In this diagram, the white classes are the ones implicitly used by the TTS example in Fig. 1 and they are described in Table 1. The shaded classes will be discussed in other examples to be given in the remaining sections.

Table 1 Description of UCM concepts.

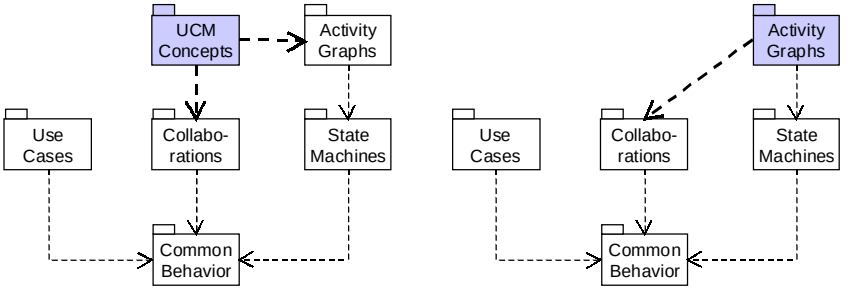
Class Name	Description
Map	Composition of path elements and components. Maps can be used as plug-ins for stubs. Maps must contain at least one start point and one end point.
Path Element	Superclass similar to a node in a connected graph.
Start Point	Beginning of a causal scenario (cause) possibly with preconditions.
End Point	End of a causal scenario (effect), possibly with postconditions.
Action Element	A path element on a causal path. References a responsibility.
Responsibility	Performs an action, activity, task, function, etc. Dynamic responsibilities (not discussed in this paper) possess additional attributes.
Continuation Element	Superclass representing a location where multiple path elements can connect together in a non-sequential way (i.e. with multiple predecessors and/or successors) as specified by a continuation rule. Each subclass defines its continuation rule (may be user-specified). In the case of a stub, the sub-map is the continuation rule.
OR	Composition (fork or join) of paths as alternatives. Conditions (guards) can be attached to paths that fork.
AND	Composition (fork or join) of concurrent paths.
Stub	Superclass representing a container for sub-maps (plug-ins).
Static Stub	Stub with a single sub-map (plug-in) and its binding relationship.

2.3 Linking UCM Concepts to UML

UCM concepts can be linked to UML in many ways. In this paper, we take advantage of similarities between UCMs and activity diagrams to facilitate this connection. Activity diagrams share many concepts with UCMs. They have common constructs and even the notations are alike, to some extent. The TTS example in Fig. 1 could effectively be described in terms of activity diagrams without any difficulty. The suggested uses of these notations are however slightly different. They both target the modeling of system-wide procedures and scenarios, but activity diagrams focus on internal processing, often found in business-oriented models (e.g. workflows), whereas UCMs are also concerned with external (asynchronous) events, which are essential to the modeling of systems that are reactive in nature.

Despite these usage differences, it seems appropriate to link the UCM concepts to the semantic model of activity diagrams: the *Activity Graphs* metamodel. Use Case Maps could be cast into this metamodel by using extension mechanisms that UML proposes, such as stereotypes, tagged values, additional OCL constraints and appropriate notation icons [17]. However in this paper, we suggest that several UCM concepts not supported by activity diagrams are important, simple, and useful enough to be included as part of the UML metamodel itself. In this way, the whole UML community would benefit from the suggested enhancements, whereas the use of extensions would lead to yet another notation based on UML which would not really be integrated to other notations that also extend UML (hence leading to interworking and compatibility issues).

Even in this context, the integration of UCMs to UML could be done in many ways. Fig. 3(a) presents an option where the UML *Behavioral Elements* package, found in the UML metamodel layer, is extended with a new sub-package for UCM concepts (shaded package). The latter depends on metaclasses found in Activity Graphs (for UCM paths) and on *Collaborations* (for UCM components, to be covered in Section 3,). However, this option would result in a new package with a lot of duplication in an already crowded set of packages and metaclasses.



a) UCM concepts in the UML metamodel b) Extending Activity Graphs to support UCMs

Fig. 3 Integrating UCM concepts to the UML metamodel layer.

Fig. 3(b) illustrates an alternative where the modifications are done directly to the Activity Graphs package. The latter would still depend on *State Machines* and, unlike the current UML standard, it would also depend on Collaborations.

Table 2 presents, through a simple mapping, how Activity Graphs metaclasses already support many UCM concepts discussed in Table 1.

Table 2 Mapping UCM concepts to Activity Graphs metaclasses.

UCM Concept	Corresponding Metaclasses
Map	ActivityGraph (from Activity Graphs), a child class of StateMachine.
Path Element	StateVertex (from State Machines), the parent class of State and Pseudo-State, which is also similar to a node in a graph.
Start Point	SimpleState (from State Machines), a State without nested states.
End Point	SimpleState (from State Machines), a State without nested states.
Action Element	ActionState (from Activity Graphs), an atomic action. In UML, an Action-State is a SimpleState with an entry action whose only exit transition is triggered by the implicit event of completing the execution of the entry action. This is similar to a UCM responsibility.
Responsibility	Associated with ActionState (from Activity Graphs), an atomic action referenced by an Action Element (in UCM terms).
Continuation Element	StateVertex (from State Machines), the parent class of PseudoState and (indirectly) of SubActivityState.
OR	PseudoState (from State Machines), of kind choice for an OR-fork and of kind junction for an OR-join.
AND	PseudoState (from State Machines), of kind fork for an AND-fork and of kind join for an AND-join.
Stub	CompositeState (from State Machines), which may contain submachines.
Static Stub	SubActivityState (from Activity Graphs), which may reference only one sub-ActivityGraph, just like a UCM static stub contains only one plug-in.

The Activity Graphs metamodel hence possess all the necessary elements to support the UCM concepts discussed so far. The different relationships in Fig. 2 are also covered by the underlying State Machines metamodel: "is predecessor for" is captured by the transitions linking the different states, "is bound to" is taken care of internally by the SubActivityState representing the UCM static stub, and "must include" could be refined as a new OCL constraint.

There are still minor differences between these UCM concepts and the semantics of Activity Graphs. In Activity Graphs, all of the paths leaving a fork must eventually merge in a subsequent join, and multiple layers of forks and joins must be well nested. There is currently no such restriction on UCMs. Also, UCM static stubs may have multiple incoming path segments, and plug-ins can have multiple start points, whereas a SubActivityState is limited to one initial state in the corresponding sub-ActivityGraph. However this can be overcome in a number of ways as SubActivity-Graph is a child class of SubmachineState (see Fig. 7), which does not have such limitation. Enhancements to the binding relationship between SubActivityState and ActivityGraph could also solve this problem.

The next section will discuss the role of UCM components for linking scenarios to structures, as well as their potential impact on the UML metamodel.

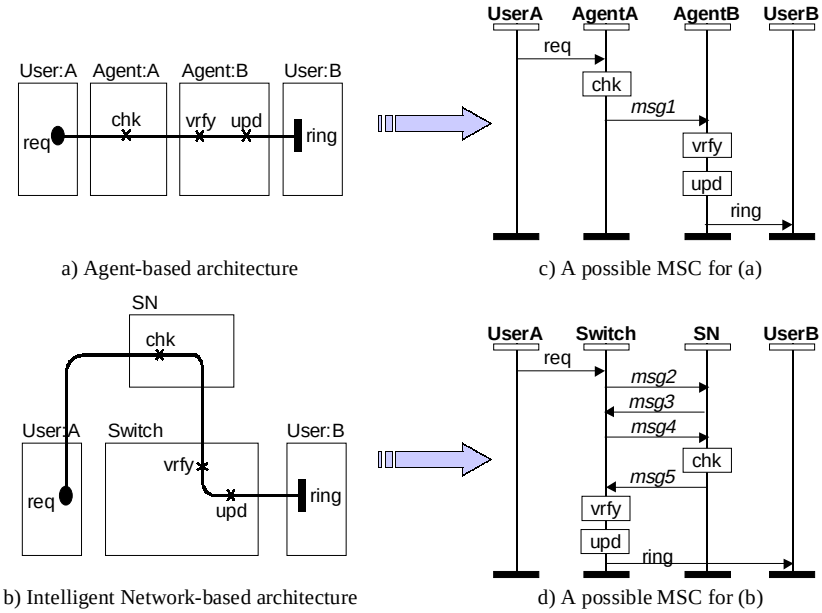
3 Combining Scenarios and Structures

One of the main strengths of UCMs resides in their capacity to visually integrate scenarios and structural components in a single view. In the design of reactive systems, such view is most useful for understanding scenario paths in their context, and for enabling high-level architectural reasoning.

3.1 UCM Component Notation

UCM scenarios can be bound to structures by visually allocating path elements to underlying components. The UCM notation distinguishes, through different shapes, several types of components useful in a reactive system context (e.g. processes, objects, agents, interrupt service routines, etc.). However, such distinctions are beyond the scope of this paper, and simple rectangles will be used as a notation for generic UCM components representing software entities as well as non-software entities.

UCM paths can be bound to various component structures. Fig. 4 uses a simplified version of the TTS system in Fig. 1 to illustrate this concept. In Fig. 4(a), a UCM path is bound to an agent-based architecture where `Users` can communicate only through their respective `Agents`. Start points indicate where users initiate events (causes) whereas end points indicate where resulting events (effects) are observed. The various components are responsible for performing the responsibilities allocated to them.



a) Agent-based architecture

c) A possible MSC for (a)

b) Intelligent Network-based architecture

d) A possible MSC for (b)

Fig. 4 UCM path bound to two different component structures, and potential MSCs.

As they can easily be decoupled from structures, UCM paths improve the reusability of scenarios and lead to behavioral patterns that can be utilized across a wide range of applications. For instance, Fig. 4(b) reuses the same scenario path in a very different context where components are based on the Intelligent Network (IN) reference model [1]. In this architecture, the `Switch` is the component responsible for establishing communication between users. However, the "intelligence" behind many IN features is located outside the switch, inside network elements called service nodes (`SN`). In this context, the `chk` responsibility, which is associated to the OCS feature, is performed by the `SN` component.

This UCM view, where scenarios and structures are combined, is most useful for architectural reasoning early in the design cycle. UCM paths are also more likely to

survive evolutions and other modifications to the underlying architecture than scenarios described in terms of message exchanges or interactions between components. For instance, note how the two following message sequence charts differ in nature and complexity (UML sequence diagrams could have been used just as well). Fig. 4(c) is an MSC capturing the scenario from Fig. 4(a) in terms of message exchanges. This is a straightforward interpretation with artificial messages (in italic characters). Other such MSCs could possibly be derived from the same scenario path. Fig. 4(d) is a potential MSC of the same scenario path, but this time bound to the IN-based architecture. Complex protocols could be involved between the switch and the service node, hence resulting in multiple messages. Communication constraints also prevent users from communicating directly with service nodes; therefore the switch needs to be involved as a relay. By using a UCM view, all these issues related to messages, protocols, communication constraints, and structural evolutions (e.g. from one version of the structure to the next) can be abstracted from, and the focus can be put on reusable causal scenarios in their structural context. If a structure is modified, path elements need only to be rebound to appropriate components.

3.2 UML Semantics Support

The UCM core concepts (Fig. 2) contains a class representing the Component concept. Components may be of different types and may contain other sub-components. Path elements are allocated to such components, as discussed in the previous section.

In the UML metamodel, the *ClassifierRole* metaclass (from the Collaborations package) seems to fit best the concept of a UCM component. Like ClassifierRoles, UCM components are interpreted as roles rather than as particular instances; e.g. Fig. 4(a) uses two generic component types for users and agents, with roles A (originating party) and B (terminating party). Being UML *Classifiers* themselves, ClassifierRoles may declare other ClassifierRoles nested in their scope, just like UCM components may contain sub-components. A ClassifierRole specifies a restricted view of a more generic Classifier. Similarly, a UCM component shows only a partial view of the overall behavior of that component type. For all these reasons, we believe that the current ClassifierRole can support the UCM component concept as is. Note that the UCM component concept is not equivalent to the UML *Component* metaclass (from the Core package), which represents a replaceable part of a system that packages implementation and provides the realization of a set of interfaces.

The need for ClassifierRole explains the additional dependency between Activity Graphs and Collaborations in Fig. 3(b). Still, the allocation of UCM path elements to their components cannot be easily captured by Activity Graphs. The latter possess the concept of *Partition*, which is a mechanism for dividing the states of an activity graph into groups. Partitions are visualized as *swimlanes* in UML activity diagrams. Unfortunately, Partitions have poor semantics because they simply regroup instances of the very abstract *ModelElement* metaclass (from the Core package) and they are quite removed from the rest of the UML metamodel. Hence, the metamodel needs to be enriched to support the useful "is allocated to" relationship found in Fig. 2. In that regard, Fig. 5 proposes two potential solutions:

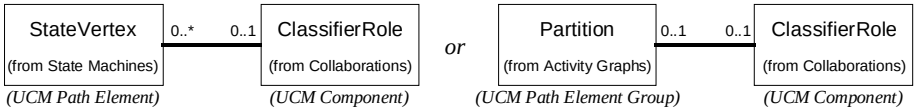


Fig. 5 Two possible solutions for the support of the UCM component concept.

Since StateVertex captures the essence of UCM path elements (see Table 2), they could be allocated directly to components, represented by ClassifierRole. Another solution would be to reuse the Partition concept of Activity Graphs and allocate a partition to a ClassifierRole (all relevant classes of Activity Graphs, and even StateVertex, are subclasses of ModelElement). Both solutions achieve our goal, but we prefer the second one because it is expressed in terms of Activity Graphs, not in terms of State Machines, and it has the capability of supporting non-StateVertex elements.

4 Modeling Dynamic Run-Time Behavior

Another important characteristic of the UCM notation is its capability of combining and integrating scenario paths in a way that enables the modeling of dynamic run-time behavior. This section presents how dynamic stubs can be used to localize and visualize, at design time, how alternative behavioral scenarios could evolve at run time.

4.1 UCM Notation for Dynamic Stub and Timers

UCMs can help structuring and integrating scenarios in various ways. The most interesting construct to do so is certainly the Dynamic Stub, shown as a dashed diamond. While static stubs contain only one plug-in map, dynamic stubs may contain multiple sub-maps, whose selection can be determined at run-time according to a *selection policy*. Such a policy can make use of preconditions, assertions, run-time information, composition operators, etc. in order to select the plug-in(s) to use. Selection policies are usually described with a (formal or informal) language suitable for the context where they are used.

Fig. 6(a) extends our original TTS Basic Call with two dynamic stubs. Whether the underlying architecture is based on agents, IN, or other types of components is not essential to the understanding of this example, hence components are not included.

This new system contains three features: the original *OCS* feature, Call Name Delivery (*CND* — displays the caller's phone number with `disp`), and *TEENLINE*. This last feature prevents several users (often teenagers) to use the phone for pre-set time intervals (e.g. from 7 PM to 10 PM), although users (e.g. parents) who provide a valid personal identification number (PIN) in a timely fashion can establish a call.

All these features are captured by plug-in maps (Fig. 6(b-e)). To simplify bindings, plug-in start/end points have been given the same names as the input/output stub segments to which they are bound. The `Originating` stub contains three plug-ins: *DEFAULT* (b), *TEENLINE* (d), and *OCS* (e). The `Display` stub contains only two: *CND* (c) and *DEFAULT* (b). The latter shows that a plug-in can be reused in multiple stubs.

Together, these five UCMs integrate multiple end-to-end sequential scenarios in a structured and concise way.

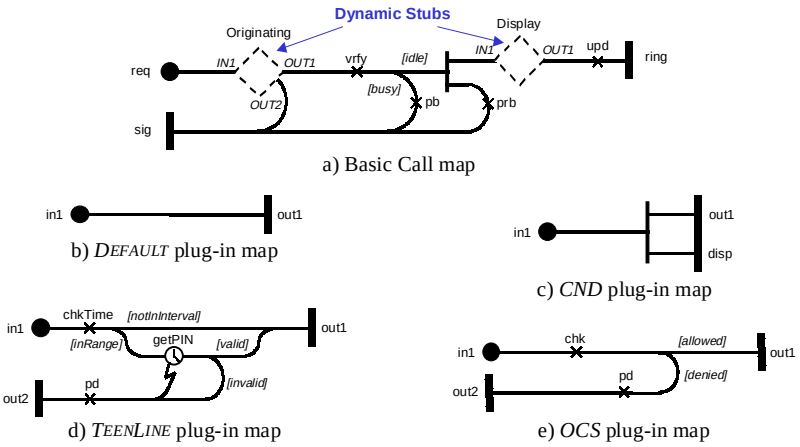


Fig. 6 Basic Call UCM with dynamic stubs and their plug-ins.

The *TEENLINE* plug-in contains a Timer named `getPIN` and shown with a clock symbol. Timers are special waiting places awaiting an event from their environment (which is the case here) or from other scenarios when visually connected to an end point (synchronous triggering) or to an empty path segment (asynchronous triggering). If the required event does not arrive in time, then the timeout path (shown with a zigzag symbol) is followed.

The selection policy for the `Display` stub could be as simple as: use *CND* if the called party has subscribed to the CND feature, else use *DEFAULT*. For the `Originating` stub however, the selection policy would need to be more complex because a user could have subscribed to both the *OCS* and *TEENLINE* features. There could potentially be an undesirable interaction between these two features, and the selection policy can be used to solve it, either in a fixed way (e.g. with priorities) or by stating a run-time resolution rule. Dynamic stubs make more local the potential conflicts that could arise between scenarios (hence leading to simpler analysis), and their selection policies can help avoiding or resolving these conflicts [2][9]. This particular way of looking at scenario combinations is at the basis of a feature interaction filtering method where undesirable interactions can be detected and dealt with early in the design cycle [16].

4.2 UML Semantics Support

As shown in Fig. 2, a Dynamic Stub is a Stub to which multiple Maps (plug-ins) are bound and to which a Selection Policy is associated. A selection policy instance should be defined as a potentially reusable object rather than as a mere attribute.

Stubs and Static Stubs were respectively mapped to CompositeState and SubactivityState in Table 2. Currently, the Activity Graphs and State Machines metamodels cannot represent, in a simple way, multiple bindings of sub-maps (i.e. ActivityGraph)

to a stub. As a consequence, extensions appear necessary. Fig. 7 proposes a solution with *DynamicStub* as a new child class of CompositeState. Dynamic stubs may reference possibly many sub-maps, and they handle a binding relationship for each reference (instead of only one as in SubactivityState and SubmachineState). A *SelectionPolicy*, which is an abstract *Relationship*, is associated to each dynamic stub.

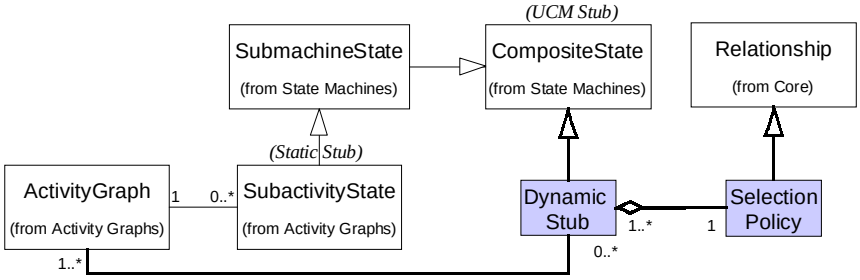


Fig. 7 Extending Activity Graphs with dynamic stubs.

The support of the Waiting Place and Timer concepts (as illustrated in the *TEENLINE* plug-in map, Fig. 6(d)) could be achieved with StateVertex. When such "wait states" are required, many UML methods suggest that statechart diagrams be used instead of activity diagrams [19]. However, we believe that wait states have their place in activity diagrams and UCMs for the modeling of reactive systems.

5 Beyond the Core Concepts

This section describes our vision on how Use Case Maps would fit in a UML-based design process. We go beyond the core concepts to address issues like connections between models and design trajectories suitable for telecommunications systems.

UML regroups various diagram techniques, which capture different views or partial representations of a same system. Some are more appropriate for the early stages of design (close to the requirements) while others are more appropriate for later stages (e.g. detailed design and implementation). These diagram techniques often focus on two orthogonal axes. *Structural diagrams* target software and conceptual entities and their relationships (e.g. class, object, component, and deployment diagrams), whereas *behavioral diagrams* emphasize behavior (e.g. sequence, collaboration, and statechart diagrams).

Structural diagrams can capture some aspects of system requirements such as the architecture and the application domain. They also share connections with behavioral diagrams, where the referenced entities often come from structural diagrams. Yet, there exists a conceptual gap between functional requirements (and use cases) and their realization in terms of behavioral diagrams, as illustrated in Fig. 8.

A UCM view represents a useful piece of the puzzle that helps bridge this gap. Requirements and use cases usually provide a black-box view where the system is described according to its external behavior. UML behavioral diagrams have a glass-box view describing the internal behavior in a detailed way. UCMs can provide a traceable progression from functional requirements to detailed views based on states, com-

ponents and interactions, while at the same time combining behavior and structure in an explicit and visual way. Whereas sequence and collaboration diagrams usually show the behavior of several objects within a single use case and statechart diagrams show the behavior of a single object across many use cases, UCMs show the behavior of many objects for many use cases. In our experience, this view contributes greatly to the understanding of complex reactive systems.

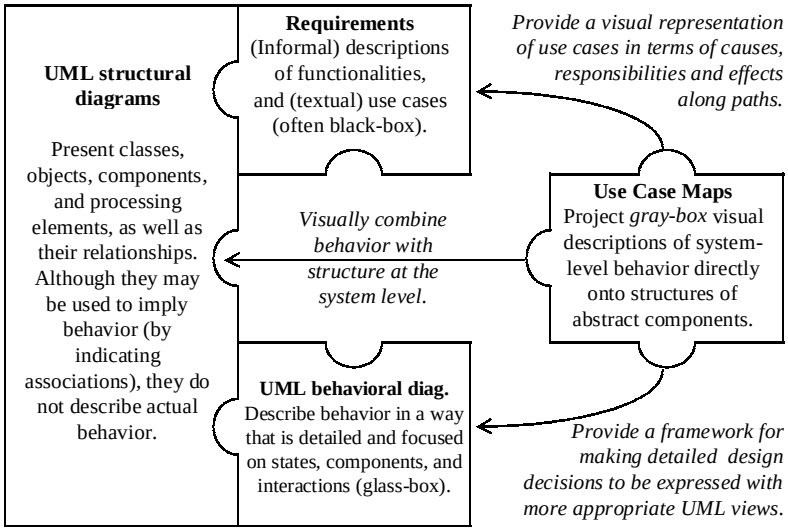


Fig. 8 UCMs as a missing piece of the UML puzzle.

Investment in UCMs can also be leveraged by connecting them to other UML views or to other modeling/specification languages. For instance:

- Buhr and Casselman use UCMs to generate class diagrams [7]. Similarly, Paech uses activity diagrams as a bridge between use cases and class diagrams [18].
- Once protocols and communication constraints are known (may be described by AssociationRoles connecting ClassifierRoles), UCMs can lead to various MSCs, sequence diagrams, and collaboration diagrams (e.g. Fig. 4). This generation is in the process of being formalized and automated in the UCM Navigator tool.
- Bordeleau presents a method for the generation of MSCs and hierarchical state machines (e.g. statechart diagrams and ROOMcharts) from UCMs [5][6]. Sales and Probert are doing similar work for the generation of SDL models [20].
- Amyot *et al.* use LOTOS as a formal means to validate UCMs and high-level designs and to detect undesirable interactions early in the design cycle [2][3].
- Other research projects include the generation of performance models (e.g. in Layered Queuing Networks—LQNs) and of abstract test cases (e.g. in the Tree and Tabular Combined Notation—TTCN).

These connections enable the creation of many design trajectories relevant to telecommunications systems, as suggested in the introduction. In particular, we envision the following trajectory, inspired from [1][6]: requirements capture and architectural reasoning is done with UCMs (stage 1), which are first transformed into MSCs or interaction diagrams (stage 2), then into state machines in SDL/UML [12] or UML-

RT [22] statechart diagrams (stage 3), and finally into concrete implementations (possibly through automated code generation). Validation, verification, performance analysis, interaction detection, and test generation could be performed at all stages.

6 Conclusions and Future Work

This paper illustrates how UCMs satisfy some of the requirements described in Section 1.1. Section 2 shows, at a semantic level, how the UML Activity Graphs metamodel already supports most of the core concepts behind the UCM notation. Advanced UCM concepts can be used to combine structure and behavior (through the allocation of path elements to components) and to model dynamic run-time behavior (with dynamic stubs and selection policies). These concepts can be integrated to the UML metamodel by extending Activity Graphs and connecting them to Collaborations (Sections 3 and 4). Other UCM concepts and notations not discussed in this paper include exceptions, failure points, (a)synchronous interactions between paths, and dynamic components and responsibilities [8], and they are left for future work.

In this paper, we attempted to minimize the number of modifications to the semantics of UML 1.3. However, in UML 2.0, Activity Graphs may be decoupled from State Machines. A reorganization of these packages could represent a good opportunity for including the UCM concepts discussed here.

Both the UCM notation and activity diagrams could be used to visualize these concepts. However, the latter would need to be extended to support bidimensional structures (swimlanes show partitions in one dimension only). As a consequence, straight lines used to represent causality between activities might need to become splines in order to adapt better to complex structures and to distinguish them from component boundaries. Also, the symbol used for activities might be too large to fit into these components. Hence, from a layout perspective, there are some advantages in using splines (as in UCM paths) and small crosses (as in UCM responsibilities). Note also that UCM may diminish the need for use case diagrams as many relationships between use cases (e.g. inclusion, extension, generalization) can be, to some extent, modeled by judicious use of stubs and plug-ins. Again, a good compromise between use case diagrams, activity diagrams, and UCMs requires further study.

The UCM notation enjoys an enthusiastic community of users and it has been used successfully in the domains of telecommunications and other reactive systems. However, users also complain about the lack of formal semantics and of a few concepts found in UML (such as actors distinguished from other components). We see in this an opportunity to add UCMs as a useful view of UML models, to integrate new expressive concepts to the UML metamodel, to precise the semantics of UCMs, to link UCMs to other languages and methodologies, and to move beyond reactive systems.

Acknowledgement. We are indebted towards Bran Selic and other members of the UCM User Group for their judicious comments, and to CITO for its financial support.

References

1. 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/pub/sbrc99.pdf

2. Amyot, D., Buhr, R.J.A., Gray, T., and Logrippo, L.: "Use Case Maps for the Capture and Validation of Distributed Systems Requirements". In: *RE'99, Fourth IEEE International Symposium on Requirements Engineering*, Limerick, Ireland, June 1999, pp. 44-53. http://www.UseCaseMaps.org/pub/re99.pdf
3. Amyot, D. and Logrippo, L.: "Use Case Maps and LOTOS for the Prototyping and Validation of a Mobile Group Call System". In: *Computer Communication*, 23(12), May 2000, pp. 1135-1157. http://www.UseCaseMaps.org/pub/cc99.pdf
4. Amyot, D. and Miga, A.: *Use Case Maps Document Type Definition 0.19*. Working document, June 2000. http://www.UseCaseMaps.org/xml/
5. Bordeleau, F. and Buhr, R.J.A.: "The UCM-ROOM Design Method: from Use Case Maps to Communicating State Machines". In: *Conference on the Engineering of Computer-Based Systems*, Monterey, USA, March 1997.http://www.UseCaseMaps.org/pub/UCM-ROOM.pdf
6. Bordeleau, F.: *A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical Finite State Machines*. Ph.D. thesis, SCS, Carleton University, Ottawa, Canada, August 1999. http://www.UseCaseMaps.org/pub/fb_phdthesis.pdf
7. Buhr, R.J.A. and Casselman, R.S.: *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, USA, 1995. http://www.UseCaseMaps.org/pub/UCM_book95.pdf
8. 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/pub/tse98final.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: *PAAM'98, 3rd Conf. on Practical Application of Intelligent Agents and Multi-Agents*, London, UK, March 1998. http://www.UseCaseMaps.org/pub/4paam98.pdf.
10. ISO, Information Processing Systems, OSI: *LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. IS 8807, Geneva, 1989.
11. ITU-T: *Recommendation Z.100, Specification and Description Language*. Geneva, 2000.
12. ITU-T: *Recommendation Z.109, SDL combined with UML*. Geneva, 2000.
13. ITU-T: *Recommendation Z. 120, Message Sequence Chart (MSC)*. Geneva, 2000.
14. ITU-T, SG10: *Proposal for a new question to define a notation for user requirements*. Canadian contribution, COM10-D56, November 1999.
15. 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, October 1998. http://www.UseCaseMaps.org/tools/ucmnav/
16. Nakamura, M., Kikuno, T., Hassine, J., and Logrippo, L.: "Feature Interaction Filtering with Use Case Maps at Requirements Stage". In: *Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00)*, Glasgow, Scotland, UK, May 2000. http://www.UseCaseMaps.org/pub/fiw00_filter.pdf
17. Object Management Group: *Unified Modeling Language Specification, Version 1.3*. June 1999. http://www.omg.org
18. Paech, B. "On the Role of Activity Diagrams in UML". In: *Int. Workshop, UML'98*, pp. 267-277. http://www4.informatik.tu-muenchen.de/papers/Pae_UML98.html
19. Rumbaugh, J., Jacobson, I., and Booch, G.: *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.
20. Sales, I. and Probert, R.: "From High-Level Behaviour to High-Level Design: Use Case Maps to Specification and Description Language". Submitted to *SBRC'2000, 18th Brazilian Symposium on Computer Networks*, Belo Horizonte, Brazil, May 2000.
21. Scratchley, W.C. and Woodside, C.M.: "Evaluating Concurrency Options in Software Specifications". In: *MASCOTS'99, Seventh International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, College Park, MD, USA, October 1999, pp. 330-338. http://www.UseCaseMaps.org/pub/mascots99.pdf
22. Selic, B.: *Turning Clockwise: Using UML in the Real-Time Domain*. In: Communications of the ACM, 42(10), October 1999, pp. 46-54.
23. *Use Case Maps Web Page* and *UCM User Group*, 1999. http://www.UseCaseMaps.org