# Deriving Message Sequence Charts from Use Case Maps Scenario Specifications

Andrew Miga[1], Daniel Amyot[2], Francis Bordeleau[1], Donald Cameron[3], and Murray Woodside[1]

[1] Carleton University, Ottawa, Canada
{miga, cmw}@sce.carleton.ca, francis@scs.carleton.ca
[2] Mitel Networks, Kanata, Canada
Daniel_Amyot@Mitel.com
[3] Nortel Networks, Ottawa, Canada
dcameron@NortelNetworks.com
http://www.UseCaseMaps.org

**Abstract.** A set of scenarios is a useful way to capture many aspects of the requirements of a system. Use Case Maps are a method for scenario capture which is good for describing multiple scenarios, including scenario interactions, for developing an architecture, and for analysing architectural alternatives. However once a component architecture is determined, Message Sequence Charts are better for developing and presenting the details of interactions, and provide access to well-developed methodologies and tools for analysis and synthesis. This paper considers what must be specified in UCM scenarios and the architecture to make it possible to derive MSCs automatically, and it describes our experience in executing these transformations within a prototype tool, the UCM Navigator.

## 1 Introduction

A scenario specification technique called Use Case Maps [11, 12] is part of a new proposal to ITU-T for a User Requirements Notation (URN) [13]. The role of the UCM notation is to capture functional requirements and it has been baptized URN-FR, while another and complementary component for non-functional requirements [14] is called URN-NFR. UCMs capture functional requirements in terms of *causal scenarios* that link sequences of responsibilities to (external) events. These scenarios may also be bound to underlying abstract components.

UCMs have been useful in describing a wide range of systems, including Wireless Intelligent Networks [2, 16, 26], agent systems [15], Wireless ATM [7], GPRS [3], and others discussed in [5, 25]. As suggested by the I.130 and Q.65 methodologies [17, 20] and several UML-based approaches [4], the process of creating specifications and standards is generally composed of three major stages. At Stage 1, services are described from the user's point of view in prose form and with use cases. The focus of the second stage is on control flows between the different entities involved, represented using Message Sequence Charts (MSCs) [19].

Finally, Stage 3 aims to provide (informal) specifications of protocols and procedures. Formal specifications are sometimes provided (e.g. in SDL [18]), but overall they still suffer from a low penetration, especially in North-America.

In such methodologies, scenarios are often used as a means to model system functionality and interactions between the entities such that different stakeholders may understand their general intent as well as technical details [27]. Use Case Maps are used in Stage 1, and to bridge the conceptual gap into Stage 2 descriptions [5]. UCMs are used to capture user (functional) requirements when very little design detail is available, *without reference to messages or component states*. In Stage 1 documents, UCM scenarios may or may not be bound to any particular components for execution. The organization and architecture of components can be introduced into the map when moving towards Stage 2 documents. One of the strengths of UCMs at this level is their ability to show a number of scenarios together, and to reason about architecture and behaviour over a set of scenarios. Once appropriate architectural decisions are taken, UCMs can be used to guide the generation of MSCs to complete Stage 2 descriptions. In turn, MSCs can be used for the synthesis and the validation of component-based behavioral models in SDL or similar languages [1, 22]. Many such synthesis techniques are studied and compared in [6].

This paper builds on previous work [2, 9] and describes research on a well-defined transformation from a subset of the UCM notation to MSC-96, along with preliminary results in implementing the transformation. This transformation enables the rapid and consistent generation of MSCs from UCMs, and the extraction of simple end-to-end scenarios from complex multilevel UCMs. These MSCs can further be refined for Stage-2 like documents (where the specifics of messages becomes more relevant), used for system understanding, and used for functional testing of more detailed models and of implementations. The UCM notation is briefly reviewed and illustrated in section 2. The UCM/MSC relationship is further studied in section 3, with a particular emphasis on scenario variables in section 4. Section 5 explains the proposed transformation, which is then illustrated with an example in section 6. Finally, section 7 presents our conclusions.

## 2 Use Case Maps

Use Case Maps visually describe causal relationships between *responsibilities* superimposed on organizational structures of abstract *components*. Responsibilities represent generic processing (actions, activities, operations, tasks, etc.). Components are also generic and can represent software entities (objects, processes, databases, servers, etc.) as well as non-software entities (e.g. actors or hardware). The relationships are said to be causal because they link causes (e.g., preconditions and triggering events) to effects (e.g. postconditions and resulting events) by arranging responsibilities in sequence, as alternatives, or in parallel. Essentially, UCMs show related and interacting use cases in a map-like diagram, whereas UCM *paths* show the progression of a scenario along a use case. Scenario

interactions are shown by multiple paths through the same component and by one path triggering or disabling another, to name a few. In UML terms, UCMs fill the gap between requirements described as (natural language) use cases and detailed behavioral based on components and messages (e.g. sequence, collaboration, and statechart diagrams). They exhibit several advantages over UML activity diagrams, as discussed in [4].

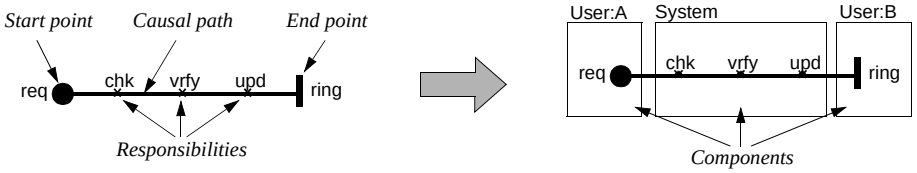


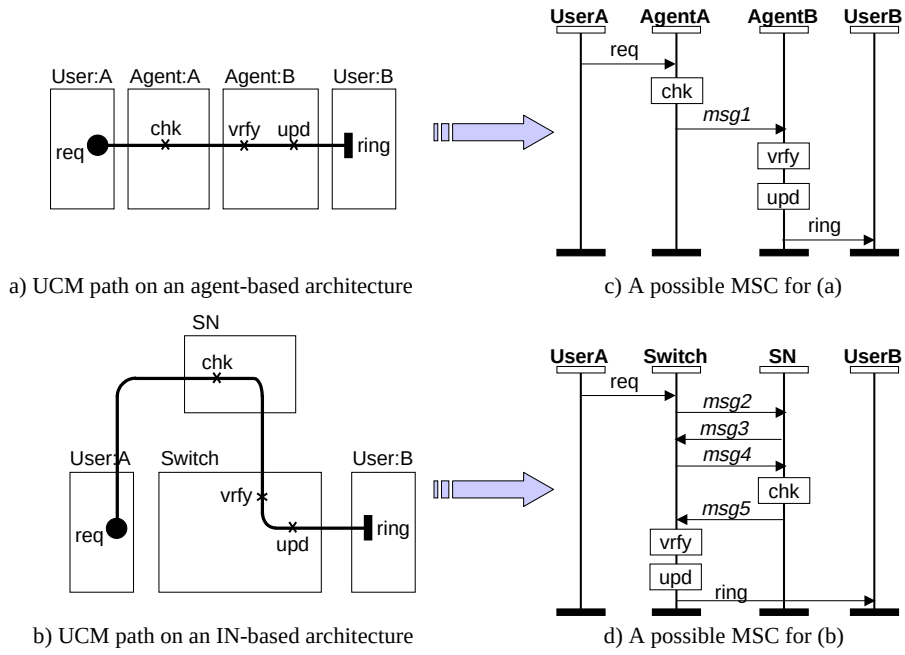

**Fig. 1.** Simple Use Case Map

The scenario in Figure 1 represents a simplified call connection initiated at a start point labelled req. The system first checks whether the call should be allowed (responsibility chk) and then verifies whether the called party is busy or idle (vrfy). In both cases here, we assume that the call request goes through as no alternative is provided. The system status then is updated (upd) and a resulting ringing event occurs (ring). Additional UCM notation elements for alternatives, concurrent paths, submaps, path interactions, dynamic components, dynamic responsibilities, etc. are described in Appendix A.

### 2.1 UCMs, Messages, and Architectural Reasoning

UCMs are useful for describing features at an early stage, even when no components are defined, and then developing a scaffolding of components to "execute" the scenarios. Alternative architectures can be developed for the same UCM, for early architectural reasoning. For instance the UCM path from Figure 1 is bound to two users connected through an agent-based architecture in Figure 2a, whereas Figure 2b uses a more conventional architecture based on Intelligent Networks (IN).

UCMs are more robust over architectural changes than the corresponding MSC. For instance, Figure 2c is an MSC capturing the scenario in Figure 2a in terms of message exchanges. Figure 2d is a potential MSC for the same scenario in an IN-based architecture. In this last MSC, complex protocols or negotiation mechanisms may be involved between the Switch and service nodes (SN), resulting in additional messages, and the Switch may be involved as a relay involving refinement of the relationship between req and chk. These figures make use of *abstract messages* ($msg_x$, in italic characters) that indicate which entities need to communicate in order to ensure the flow of causality found in the UCM.




a) UCM path on an agent-based architecture

c) A possible MSC for (a)

b) UCM path on an IN-based architecture

d) A possible MSC for (b)

**Fig. 2.** UCM path bound to two different component structures, and potential MSCs

The UCM view is a useful reference point which remains stable over changes related to messages, protocols, communications constraints and structure. UCMs avoid early commitments to detailed structures and messages and they remain focused on intended functionality and on reusable causal scenarios, within an evolving context.

### 2.2 UCM Navigator

The UCM notation is supported by a freely available editing tool: the *UCM Navigator* [21, 25]. Among other features, this tool supports the path and component notations found in Appendix A, and it maintains various kinds of bindings (plug-ins to stubs, responsibilities to components, sub-components to components, etc.). Also, it allows users to navigate much like a Web browser, and to visit and edit the plug-ins related to stubs of all levels (see screen captures in Figure 5). Editing operations maintain syntactic correctness, by either inserting correct elements, or transforming existing paths.

The UCM Navigator saves, loads, exports and imports UCM as XML files, which are valid according to a UCM Document Type Definition (DTD) [13]. This DTD describes the current formal definition of UCMs, which is based on hypergraphs, and the UCM Navigator ensures that syntactic and static semantic rules are satisfied.

The tool can also export UCM figures in three formats: Encapsulated PostScript (EPS), Maker Interchange Format (MIF), and Computer Graphics Metafile (CGM). Flexible reports can be generated as PostScript files ready to be transformed into hyperlinked and indexed PDF files. Multiple platforms are currently supported: Solaris, Linux (Intel and Sparc), HP/UX, and Windows (95, 98, 2000 and NT).

## 3 Relationships between UCM and MSC

To develop Message Sequence Charts from UCMs, we need to analyze how their respective concepts relate to each other. Here is a comparison of their main concepts:

- **Abstraction**. UCMs describe scenarios in terms of causal sequences of responsibilities, which are identified at a high level of abstraction by a label and a brief textual description. The description may abstract away some inter-component communication. On the other hand, MSCs describe scenarios in terms of sequences of inter-component messages, actions, and methods.
- **Components**. A UCM defines components in terms of the role they play in a scenario (by means of a short textual description), and the responsibilities they provide. Components are represented by rectangles which can express layered and peer-to-peer relationships by position. A MSC represents components (called instances) using timelines, which express only the existence of separate locations. UCM components are optional whereas MSC instances are mandatory.
- **Alternative and concurrent sub-scenarios**. Both provide for a main scenario with a set of alternatives. Alternative paths are combined in a UCM diagram using the OR-fork segment connector. Dynamic stubs may also be used to describe alternatives. MSCs use the inline alternative box in basic MSC (or the OR notation in HMSC). Similar concepts exist for concurrent scenarios.
- **Scenario interactions**. UCMs have explicit notation for scenario interactions, as shown in Appendix A; this feature does not exist in MSC.

For the purpose of the UCM-to-MSC transformation, we establish a one-to-one relationship between valid UCM scenarios and basic MSCs, and between the following elements of the two models:

- a UCM component and an MSC instance (UCM sub-components are not being converted) an unbound UCM triggering (or resulting) event and an MSC message from (or to) the environment
- a UCM path crossing from one component to another and an abstract MSC message
- a UCM start (or end) point that is not bound to a stub input (or output) segment and an abstract MSC message

- a UCM precondition or postcondition and an MSC condition (expressing system state) (partially implemented in the Navigator at present)
- a UCM responsibility and an MSC action
- a UCM OR-fork or a UCM dynamic stub with multiple plug-ins, and multiple basic MSCs. Although the alternative inline box could be used here, having multiple MSCs simplifies the understanding of end-to-end scenarios
- a UCM AND-fork and an MSC parallel inline box (to avoid an explosion in the number of MSCs);
- a UCM loop and an MSC loop box (not implemented in the Navigator at present)
- a UCM timer and an MSC timer, with UCM triggering events as MSC resets and UCM timeout paths as MSC timeouts (partially implemented in the Navigator at present).

It is important to note that in the transition from UCM to MSC, the explicit interactions between scenarios expressed in UCM maps is lost because basic MSCs do not allow to explicitly express interactions between scenarios. However, causal flows in end-to-end scenarios will be preserved.

## 4 Scenario Variables in UCMs

A Use Case Map describes multiple scenarios, some with separate starting points and others that share starting points but under different types of input data, or different system states. A single scenario is the path traced out by placing a "token" on a particular map start point and by tracing one path through the various choices that are offered. These tokens are assumed to duplicate and merge along AND-forks and AND-joins. A single scenario gives rise to one basic MSC.

A particular scenario can be distinguished by the designer, by defining the conditions that govern it, as the *context* of the scenario. They include the state of the system at the time the scenario is executed, and the data that triggers the start point. Typically the designer can express the context in words, drawn from some previous use case. However, a more precise definition of each scenario context is needed to allow a meaningful set of MSCs to be created. *Scenario variables* have been introduced for this purpose, to govern the choice of alternatives. There are two kinds of *choice points* where a path has alternatives:

- an *OR-fork* allows a path to branch into multiple segments, which may possibly be joined subsequently by OR-joins (or before in the case of a loop)
- a *dynamic stub* is a placeholder for a set of plug-in maps, with the choice of submap depending on the particular scenario.

Without scenario variables, the UCM indicates that all alternatives are possible at every choice point. A symptom of inadequately specified scenario contexts is a combinatorial explosion of scenarios, many of which are meaningless. To

illustrate this point, consider the example in Figure 3. While there are 64 potential combinations of OR-fork choices in the path above and thus 64 potential scenarios, when the branch choices are based on the state of the variable $x$ there are only four valid scenarios.

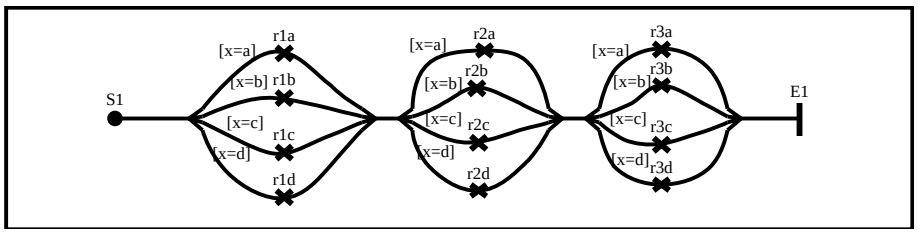



**Fig. 3.** Example of correlated OR-fork choices

As another example, dynamic stubs are often used to represent specific behaviour of features of a system with different plug-ins for each feature and many different dynamic stubs along a scenario path. Depending on which feature is active (part of the system state), only the corresponding plug-ins for that feature should be selected at each dynamic stub. For example if a path contains 10 dynamic stubs, each of which contains 10 plug-in maps to implement 10 different features, then there would be 1000 possible scenarios of which only a handful could be valid.

A *scenario data model* is added to UCM notation to allow designers to specify the context of an end-to-end scenario, and the logical selection conditions at choice points. This provides:

- precise specification of each scenario context for named end-to-end scenarios through a multi-level UCM design
- MSC generation of a specific scenario
- selection and highlighting of one scenario path in the UCM Navigator, for understanding of a design
- capability to discover invalid scenarios.

### 4.1 Scenario Variables

Selection conditions at choice points are based on global boolean variables whose values are defined for the scenarios that use the choice point. Future refinements of this most simple data model will allow for enumerated and integer values and local variables scoped to a particular token, map or stub.

There are three steps needed to specify the set of valid scenarios:

1. define the set of global variables for the map
2. specify logical selection conditions at OR-forks and dynamic stubs

3. specify a scenario definition for each valid scenario.

The first two steps are done once for the entire design. The third step, creating the scenario definitions simply sets the variable state so that the proper decisions are made at each branching point.

Usually the global variables emerge from consideration of the set of intended scenarios. They should be identified while the map is being created from use cases, with documentation either by formal notation or in words, of the role and conditions for each scenario. The full set of these scenario contexts is then described by a set of boolean variables, such that for each start point, each scenario from that start point has a unique representation in the values of the variables. The dialog box for defining these variables is shown in Figure 4. The names may not include logical operators (i.e. +, &, =, and !).

### 4.2 Scenario Definitions

Scenario definitions are the means by which named end-to-end scenarios can be specified and referenced. Apart from a name and description of the scenario, they contain the starting point of the scenario as well as a set of variable initializations. A variable initialization is a reference to a global variable coupled to the value to which it should be set for a scenario. A list of these initializes the system state to the proper values for a given scenario.

Scenario variables can be set to the values *true* and *false*, for the given scenario. If not set, they have the value *undefined*, which causes logical expressions referencing the variable to evaluate to *false* and a warning to be emitted. In addition, it would be useful to be able to define a variable as *don't care*, within a given scenario; however this has not been implemented yet.

Figure 4 shows the dialog box (at right) that is used to create and edit scenario definitions as well as invoke MSC generation and scenario highlighting operations for defined scenarios. Once scenarios are defined, another dialog box (not shown) can be invoked from a path start point to list the scenarios that begin at that point and invoke MSC generation and path highlighting operations.

Scenarios are organized into groups which are listed on the top left section of the dialog. Individual scenarios are listed in the top right section. They must be given a unique name and can be given a description (same for scenario groups). The bottom section allows for the editing of the currently selected scenario's variable initializations and for invoking operations using the scenario definition. The list of variable initializations is created by selecting an unset variable from the left list and specifying its initial value. It is then removed from the left list and the variable-initial value pair is placed in the center list. Controls allow for creating, modifying and destroying such initializations. The set of variables that need to be initialized for a given scenario is generally a subset of the total list of predefined variables.

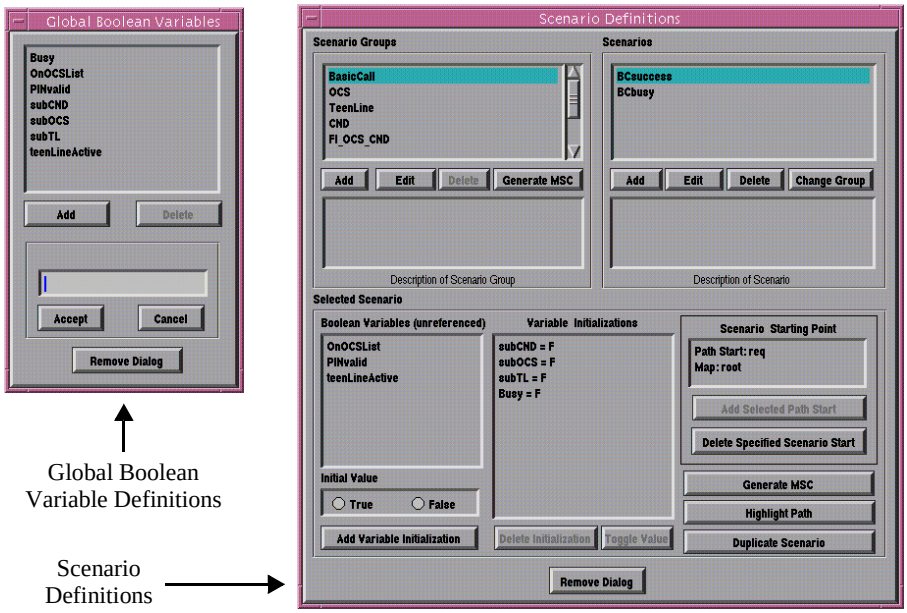



**Fig. 4.** Dialog boxes for scenario variables and definitions

### 4.3 Logical Selection Conditions at OR-Forks and Dynamic Stubs

Logical selection conditions are logical expressions designed to produce a boolean output value with a combination of variable references and logical operators (*and* '&', *or* '+', *not* '!', *equal* '=', *not equal* '!=', and brackets ()).

For example, if A, B, C, and D are variable names then ((A+B) &(C!=D)) & !(A=D) is a valid logical expression. So is A=T & B=F as the constants T (*true*) and F (*false*) may be referenced explicitly in conditions.

Syntactic validation is performed at creation time so that invalid conditions are not accepted in the tool. In addition all conditions are stored internally with references to variable objects so that the renaming of a variable causes no problem. Also reference counting on variable objects is performed so that a variable referenced in either a selection condition or a scenario definition cannot be deleted prematurely.

Figure 5 shows a set of screenshots that illustrate the relevant dialog boxes invoked for specifying logical conditions for OR-fork branches (top) and for plug-in maps (bottom). The dialog for editing conditions is the same in both cases and is opened as a subdialog of the OR-fork specification and plug-in choice dialogs. This dialog contains a list of all predefined variables for users to select.

In order to validate if the entire scenario specification operation has been performed correctly, a named scenario can be selected and highlighted with a special path colour throughout a multilevel scenario. This provides immediate

feedback to designers in UCM terms. If there are errors either the wrong path or multiple paths will be highlighted. If a branching point is reached where none of the options evaluate to *true*, an error message is given and the highlighting fails.

## 5 Transformation of UCM Paths into Message Sequence Charts

The UCM-to-MSC transformation generates basic MSCs (MSC-96) from UCMs in two main steps:

- Identification of the valid scenarios in the UCM model (this is where the scenario variables data are required)
- Generate MSC elements according to the correspondences identified in Section 3.

This section expands on this transformation with an overview of the underlying object model and of the scanning algorithm.

### 5.1 Hypergraph Model

A *hypergraph* model is used to store the use case paths, with hyperedges representing UCM path elements (start, end points, responsibilities, forks, joins, stubs) and nodes being internal objects representing connections between hyperedges. UCM components are separate objects referenced by hyperedges that are inside their boundaries. All of the objects for a given map are contained inside a map object. Connections between parent and child maps are specified by user defined stub bindings, which bind the input and output segments of a stub in a parent map to path start and end points in a submap.

### 5.2 Recursive Scanning Algorithm

There are two methods of generating MSCs:

- Generating the MSC(s) for one or more predefined scenarios by selecting scenarios from a list
- Generating all possible scenarios that start from a given point by determining all possible combinations of OR-fork choices and plug-in selections and generating a separate MSC for each (hence ignoring scenario variables).

Both methods of generating MSCs use a recursive algorithm to determine all possible combinations of OR-fork branches and plug-in maps. The difference is with use of scenario data and with branch conditions defined, the recursive algorithm only follows those OR-fork branches or plug-in maps whose selection

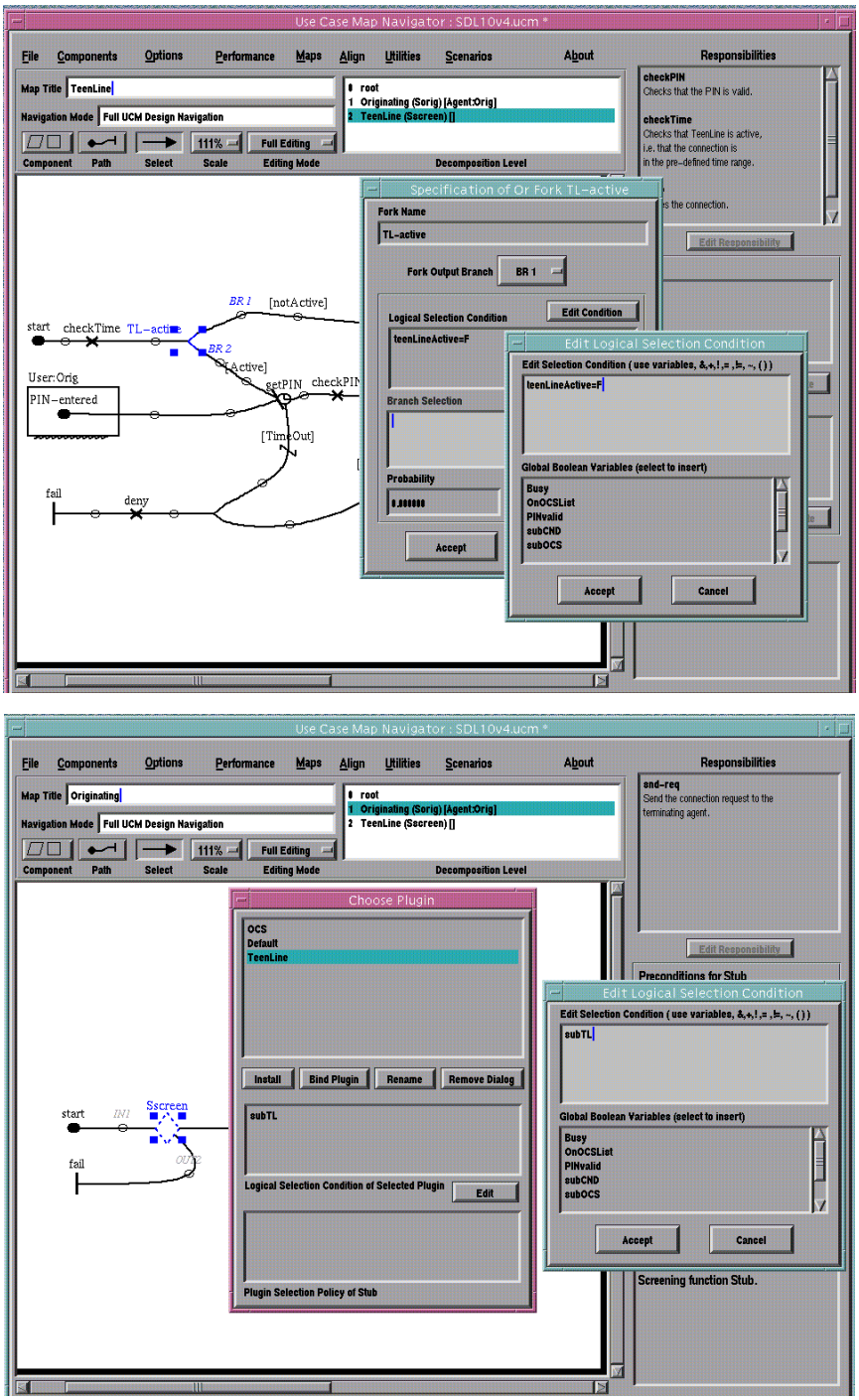



**Fig. 5.** Specification of logical selection conditions for OR-forks and dynamic stubs

conditions evaluate to *true* at run time. With fully specified selection conditions for a design this should result in only the single proper path taken. Nondeterministic UCM choices (where many conditions evaluate to *true*) will cause multiple MSC scenarios to be generated, one for each alternative.

For real designs of any complexity, scenario data must be used as the other method can generate a very large number of MSCs (as already discussed). If errors exist, it is possible multiple scenarios will be generated for a scenario definition as the same recursive algorithm is used. In this manner users are notified that elements are either over- or under-specified.

## 6 Example

To illustrate the UCM-to-MSC transformation, we extend the simple call request UCM in Figure 1 to include multiple UCM constructs, embedded maps, and system functionality (*features*).

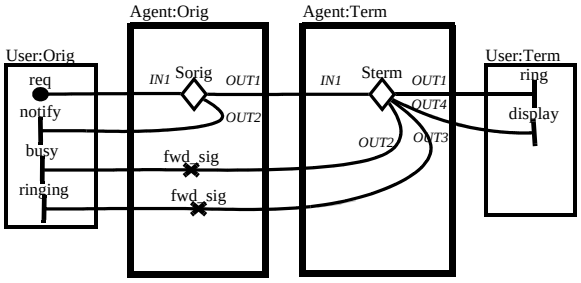



**Fig. 6.** Basic Call root map

### 6.1 Basic Call and Feature UCMs

The top-level UCM is the Basic Call of Figure 6, which contains four components (originating/terminating users and their agents) and two static stubs. The first stub (Sorig) contains the ORIGINATING plug-in whereas the second (Sterm) contains the TERMINATING plug-in. In turn, these two plug-ins have other stubs (Sscreen and Sdisplay) and another level of maps. Each of these latter stubs includes a DEFAULT plug-in (which happens to be the same in both cases) that represents how the basic call reacts in the absence of other features. These plug-ins, generated with the UCM Navigator, are presented in Figure 7.

The ORIGINATING side has two features (plug-ins) used in Sscreen:

– **Originating Call Screening** (OCS), which checks whether the call should be denied or allowed (chk). When denied, an appropriate event occurs at

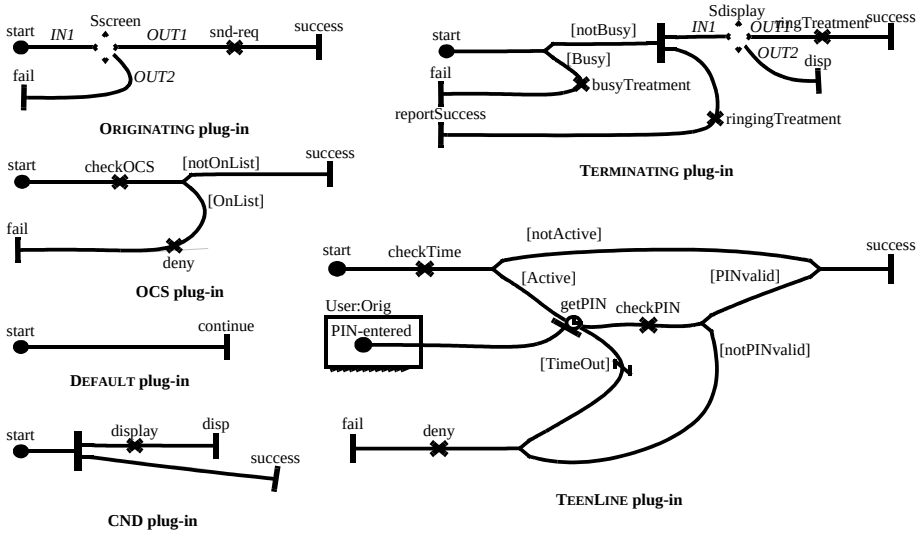



**Fig. 7.** Plug-ins for Basic Call and three features

the originator side (notify). Its binding relationship, which connects the input/output segments of a stub to the start/end points of its plug-in, is $\{< IN1, start >, < OUT1, success >, < OUT2, fail >\}$.

– **TeenLine**, which denies the call provided that the request is made during a specific time interval and that the personal identification number (PIN) provided is invalid or not entered in a timely manner. The zigzag path leaving the timer represents a timeout path. The binding relationship for this feature is also $\{< IN1, start >, < OUT1, success >, < OUT2, fail >\}$.

The TERMINATING side contains only one feature plug-in, used in Sdisplay:

– **Call Number Delivery** (CND), which displays the number of the originating party (display) concurrently with the rest of the scenario (update and ringing). The binding relationship is $\{< IN1, start >, < OUT1, success >, < OUT2, fail >\}$.

Note that the start and end points of these plug-ins are not external events but connectors to the input/output segments of their parent stub.

Adding features to such UCM collections is often achieved by creating new plug-ins for the existing stubs, or by adding new stubs containing either new plug-ins or instances of existing plug-ins. In all cases, the selection policies and pre-conditions need to be updated appropriately.

### 6.2 Scenario Variables and Conditions

Seven scenario variables were created according to the two categories discussed in section 4:

- branch selection in OR-forks: Busy, OnOCSList, PINvalid, and TeenLineActive
- subscriptions to features, i.e. selection of plug-ins in dynamic stubs: subOCS, subTL, subCND.

Branch selection conditions, found in three plug-ins (OCS, TERMINATING, and TEENLINE), use the first categories of variables. All these conditions reference only one variable, as is and complemented (e.g. PINvalid for `[PINvalid]` and !PINvalid for `[notPINvalid]`).

Plug-in preconditions use the second category of variables. They form the *selection policies* found in our dynamic stubs. These policies provide a simple mechanism for feature interaction resolution, which is local to a particular stub. Policies can hence establish precedence of one feature over another. For instance, stub Sscreen contains three plug-ins whose selection is done as follow:

subOCS → use OCS
!subOCS & subTL → use TEENLINE
!(subOCS+subTL) → use DEFAULT

Selection policies enables one to derive scenarios that involve multiple features, and hence to visualize desirable and undesirable interactions early in the development process.

### 6.3 Message Sequence Charts

From the req start point in the Basic Call root map, various scenarios can be generated. The presence of seven variables suggests an upper bound of $2^7 = 128$ such scenarios (assuming all choices are guarded and deterministic), but the path structure actually constrains this number to 15 scenarios. However, this number would increase dramatically as features are added or made more complex, as explained in section 4. Although all these MSCs can be generated, the more valuable and traceable MSCs are the ones produced by explicitly defining contexts using scenario variables. Such scenarios can be well documented, referenced, and studied through the highlighting function of the UCM Navigator.

Scenarios can be defined for the Basic Call (no feature), for one feature at a time, or for multiple features at a time. Figure 8 shows the MSC (MSC-96) corresponding to a successful call connection made by an originating user subscribed to OCS to an available terminating user. Abstract messages (e.g. *m1* and *m2*) are generated where necessary, as discussed in Section 2.1. The Z.120 textual representation is output directly from the UCM Navigator, and it can be easily converted to a graphical form by existing MSC/SDL tools or by packages such as [10] (used in this paper). Minor but automatable transformations may have to be done to the textual representation to satisfy identifier conventions (e.g. removal of underscore symbols "_" in messages).

Combinations of features are particularly interesting as the emerging behaviour is often surprising. Figure 9 (left) shows the MSC corresponding to a successful call connection made by an originating user subscribed to OCS to an

```
mscdocument OCSsuccess;
msc OCSsuccess;
User[Orig]:  instance;
Agent[Orig]: instance;
Agent[Term]: instance;
User[Term]:  instance;
User[Orig]:  out req,1 to Agent[Orig];
Agent[Orig]: in req,1 from User[Orig];
             action 'checkOCS';
             condition [notOnList];
             action 'snd_req';
             out m1,2 to Agent[Term];
Agent[Term]: in m1,2 from Agent[Orig];
             condition [notBusy];
all: par begin;
   Agent[Term]: action 'ringTreatment';
                out ring,3 to User[Term];
   User[Term]:  in ring,3 from Agent[Term];
all: par;
   Agent[Term]: action 'ringingTreatment';
                out m2,4 to Agent[Orig];
   Agent[Orig]: in m2,4 from Agent[Term];
                action 'fwd_sig';
                out ringing,5 to User[Orig];
   User[Orig]:  in ringing,5 from Agent[Orig];
all: par end;
Agent[Term]: endinstance;
Agent[Orig]: endinstance;
User[Orig]:  endinstance;
User[Term]:  endinstance;
endmsc;
```

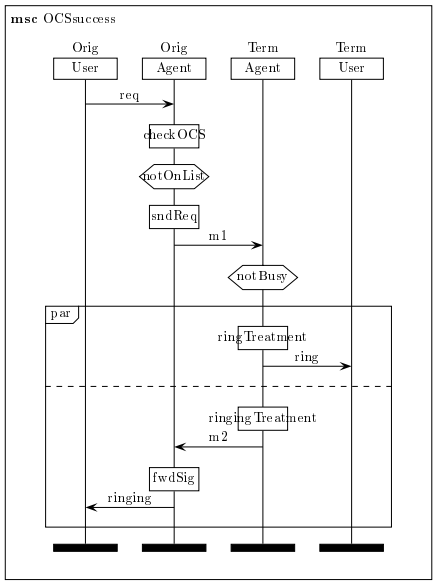



**Fig. 8.** OCS successful scenario in MSC form

available terminating user with CND. As a result, three messages are sent in parallel by the terminating agent.

The second MSC in Figure 9 (right) results from a situation where the originating user has subscribed to both OCS (but the terminating user is not on the screening list) and TeenLine (active, with an invalid PIN). The terminating user is assumed to be busy. In theory, the call should be denied by TeenLine. However, due to Sscreen's selection policy (described in section 6.2), OCS prevents TeenLine to be activated. This is a typical feature interaction that needs to be solved at the policy level (and perhaps at the UCM path level as well). MSCs generated from UCMs make it possible to find such unexpected behaviour very early in the design process, without any commitment to complex protocols.

## 7 Conclusions

This work studied transformation of a UCM into a set of MSCs, intending to jump-start the use of MSCs in design, and link them to requirements analysis. We found that UCMs must be enhanced by introducing a formal *scenario data model*, to identify the intended scenarios from all potential scenarios. The paper has described the concept of scenario variables, how they are used in small but realistic UCM examples, and how MSCs are generated from such UCMs.

Scenario variables capture designer intentions for a given scenario among all those that begin at a given start point; they capture at least some of the

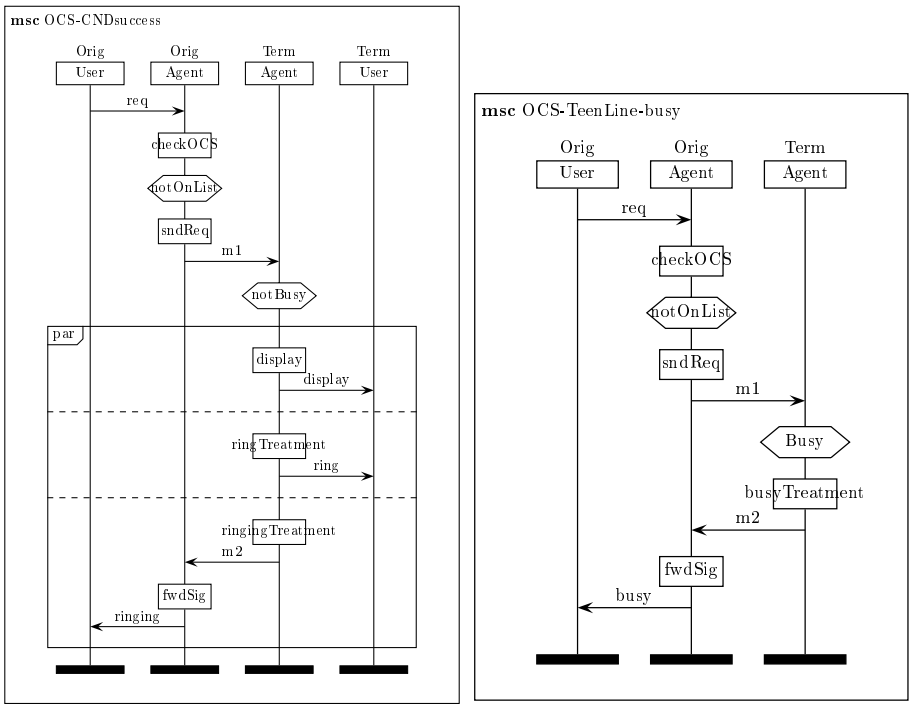



**Fig. 9.** OCS-CND and OCS-TeenLine MSCs

preconditions of the scenario as a formal part of the UCM model, and they help to remove or to identify unwanted "emergent possible paths". The intended paths are then transformed into MSCs by establishing simple correspondences between the models.

If the scenarios are the basis of design of tests, the scenario variables also define test cases. One goal in transforming to MSCs is to exploit design and test-generation tools based on MSCs and SDL or other formal languages.

The use of scenario variables in the examples described here is somewhat limited by the size of the examples, and choices are often specified by the value of a single variable. In larger specifications we have found that more complex functions of several variables are often used.

Several enhancements to this work would appear to be useful. The transformation can be expanded to include details which are identified but not modeled in the UCM. Detailed patterns of messages and activities equivalent to sub-MSCs can be defined in the UCM, for responsibilities or for inter-component responsibility transfer, and included in the MSC output either as additional detail within the MSC, or as MSC references. Complex and realistic message exchanges (e.g. as in Figure 2d, but with meaningful message names and parameters rather than abstract messages) could hence be generated.

The scenario data model could be enriched, without going to a complete data model like ASN.1. The data types described here are restricted to booleans in order to simplify them, to focus the designer on declaring intentions, and to avoid the temptation to write program code at this stage. Modest potential enhancements include:

- better support for pre/postconditions, loops and timers
- enumeration types to distinguish multiple cases, and bounded counters
- operations on scenario variables by responsibilities, to allow counters to be incremented and control flags to be set as a token moves along a path
- token variables could be introduced, instead of just global variables, for scenarios with multiple concurrent tokens.

A clearer and more precise dynamic semantics for UCMs is also desirable in the context of MSC generation. For instance, a stub could create new instances of a plug-in each time it is accessed, or else access the same instance (singleton).

It would not be difficult to transform to HMSCs instead of MSCs. The structure of alternate paths and forks and joins in the UCM could be used to define the HMSC road map, and the linear sub-paths within this structure would define the message sequences within the HMSC.

The transformation machinery described here is also being adapted to create test cases and performance models from UCMs [24], and to exploit UCM scenario information in generating SDL models [8, 23].

**Acknowledgements**

This research was supported by Nortel Networks, Mitel Networks, and by the Industrial Partnerships program of the Natural Sciences and Engineering Research Council of Canada. Discussions with John Visser, Jim Hodges, Jacques Sincennes, Luigi Logrippo and Gunter Mussbacher were helpful in developing the ideas.

## References

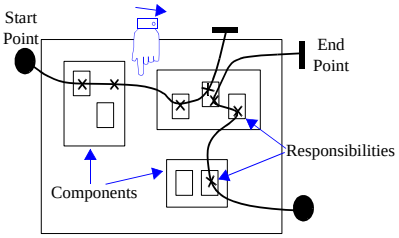
1. Abdalla, M.M., Khendek, F. and Butler, G.: "New Results on Deriving SDL Specifications from MSCs". In: *SDL'99, Proceedings of the Ninth SDL Forum*, Montreal, Canada. Elsevier (1999)
2. Amyot, D. and Andrade, R.: "Description of Wireless Intelligent Network Services with Use Case Maps". In: *SBRC'99, 17° Simpósio Brasileiro de Redes de Computadores*, Salvador, Brazil (May 1999) 418–433
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) 1135–1157
4. Amyot, D. and Mussbacher, G.: "On the Extension of UML with Use Case Maps Concepts". In: *<<UML>>2000, 3rd International Conference on the Unified Modeling Language*, York, UK (October 2000), LNCS 1939, 16–31

5. Amyot, D.: "Use Case Maps as a Feature Description Language". In: S. Gilmore and M. Ryan (Eds), *Language Constructs for Designing Features*. Springer-Verlag (2000) 27–44
6. Amyot, D. and Eberlein, A.: "An Evaluation of Scenario Notations for Telecommunication Systems Development". In: *9th Int. Conference on Telecommunication Systems (9ICTS)*, Dallas, USA (March 2001)
7. Andrade, R.: "Applying Use Case Maps and Formal Methods to the Development of Wireless Mobile ATM Networks". In: *Lfm2000: The Fifth NASA Langley Formal Methods Workshop*, Williamsburg, Virginia, USA (June 2000)
8. Bordeleau, F.: *A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical Finite State Machines*. Ph.D. thesis, School of Computer Science, Carleton University, Ottawa, Canada (August 1999)
9. Bordeleau, F. and Cameron, D.: "On the Relationship between Use Case Maps and Message Sequence Charts". In: *2nd Workshop of the SDL Forum Society on SDL and MSC (SAM2000)*, Grenoble, France (June 2000)
10. Bos, M. and Mauw, S.: "A LaTeX macro package for drawing Message Sequence Charts". Version 1.47, http://www.win.tue.nl/~sjouke/mscpackage.html (1999)
11. Buhr, R. J. A. and Casselman, R. S.: *Use Case Maps for Object-Oriented Systems*, Prentice-Hall (1996)
12. Buhr, R. J. A.: "Use Case Maps as Architectural Entities for Complex Systems". In: *IEEE Transactions on Software Engineering*, 24(12) (Dec. 1998) 1131–1155
13. Cameron, D. et al.: *Draft Specification of the User Requirements Notation*. Canadian contribution CAN COM 10-12 to ITU-T, Geneva (November 2000)
14. Chung, L., Nixon, B. A., Yu, E. and Mylopoulos, J.: *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers (2000)
15. Elammari, M. and Lalonde, W. (1999) "An Agent-Oriented Methodology: High-Level and Intermediate Models". In: *Proc. of the 1st Int. Workshop. on Agent-Oriented Information Systems (AOIS'99)*, Heidelberg, Germany (June 1999)
16. Hodges, J. and Visser, J.: "Accelerating Wireless Intelligent Network Standards Through Formal Techniques". In: *IEEE 1999 Vehicular Technology Conference (VTC'99)*, Houston (TX), USA (1999)
17. ITU-T: *Recommendation I.130, Method for the characterization of telecommunication services supported by an ISDN and network capabilities of ISDN*. CCITT, Geneva (1988)
18. ITU-T: *Recommendation Z.100, Specification and Description Language (SDL)*. Geneva (1999)
19. ITU-T: *Recommendation Z.120, Message Sequence Chart (MSC)*. Geneva (1999)
20. ITU-T: *Recommendation Q.65, The unified functional methodology for the characterization of services and network capabilities including alternative object-oriented techniques*. Geneva (2000)
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 (October 1998)
22. Mansurov, N. and Zhukov, D.: "Automatic synthesis of SDL models in use case methodology". In: *SDL'99, Proceedings of the Ninth SDL Forum*, Montreal, Canada. Elsevier (1999)
23. Sales, I. and Probert, R. L.: "From High-Level Behaviour to High-Level Design: Use Case Maps to Specification and Description Language". In: *SBRC'2000, 18° Simpósio Brasileiro de Redes de Computadores*, Belo Horizonte, Brazil (May 2000)

24. Scratchley, W. C.: *Evaluation and Diagnosis of Concurrency Architectures*. Ph.D. thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (November 2000)
25. *Use Case Maps Web Page and UCM User Group* (since March 1999) `http://www.UseCaseMaps.org`
26. Yi, Z.: *CNAP Specification and Validation: A Design Methodology Using* LOTOS *and UCM*. M.Sc. thesis, SITE, University of Ottawa, Canada (2000)
27. Weidenhaupt, K., Pohl, K., Jarke, M., and Haumer, P.: "Scenarios in System Development: Current Practice". In: *IEEE Software* (March/April 1998) 34–45

N.B. Most UCM papers are available at `http://www.UseCaseMaps.org/pub/`

# A UCM Quick Reference Guide

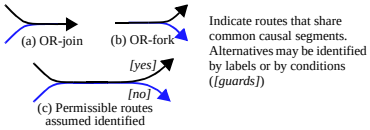



Imagine tracing a path through a system of objects to explain a causal sequence, leaving behind a visual signature. Use Case Maps capture such sequences. They are composed of:

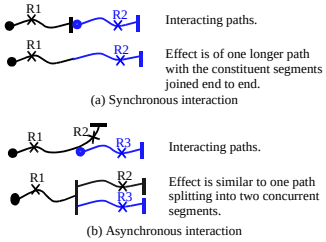
- **start points** (filled circles representing preconditions and/or triggering causes)
- causal chains of **responsibilities** (crosses, representing actions, tasks, or functions to be performed)
- and **end points** (bars representing postconditions and/or resulting effects).

The responsibilities can be bound to **components**, which are the entities or objects composing the system.

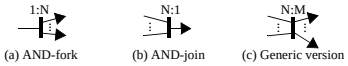
A1. Basic notation and interpretation



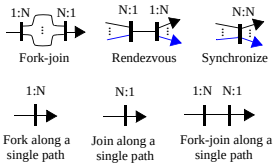

A2. Shared routes and **OR-forks/joins**.



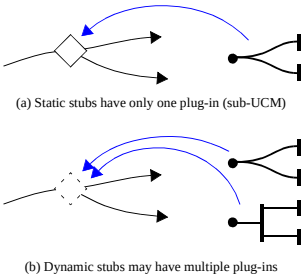

A3. Path interactions.




A4. Concurrent routes with **AND-forks/joins**.



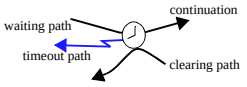

A5. Variations on AND-forks/joins.



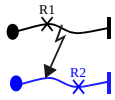
(a) Static stubs have only one plug-in (sub-UCM)

(b) Dynamic stubs may have multiple plug-ins

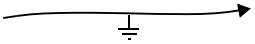
A6. **Stubs** and **plug-ins**.



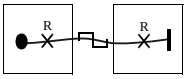

(a) Timers may be set, reset, and timed-out




(b) Top path aborts bottom path after R1



(c) Ground symbols indicate possible path failure points




(d) R is a shared responsibility

A7. **Timers, aborts**, **failures**, and **shared responsibilities**.

(a) **Team**: generic container

(b) **Object**: passive component

(c) **Process**: active component

(d) **ISR**: Interrupt Service Request

(e) **Agent**: for agent systems

(f) **Pool**: container for dynamic components as data

A8. Component types.

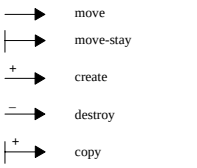
(a) **Stack**: multiple instances

(b) **Protected**: for mutual exclusion

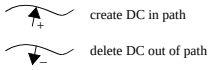
(c) **Slot**: placeholder for dynamic components as operational units

(d) **Anchored**: in a plug-in, refers to a component defined in another map

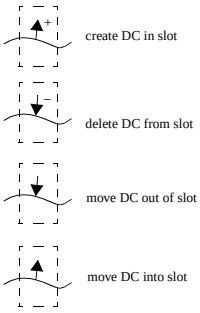
A9. Component attributes.



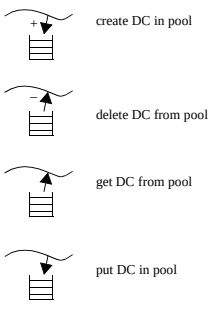

(a) Movement of DCs as data




b) Directly into or out of paths



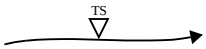

(c) Into or out of slots



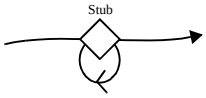

(d) Into or out of pools

A10. Movement notation for **dynamic components** (DCs).


(a) **Goal tags** are start and end points for goals in agent systems




(b) **Timestamps** are start and end points for response time requirements




(c) **Direction arrows** can be used when path direction is ambiguous

A11. Notation extensions