Use Case Maps

Quick Tutorial

Version 1.0

© Daniel Amyot

SITE, University of Ottawa
September 1999

Table of Contents

1. Introduction

1.1 Structure of this Tutorial

This quick tutorial is intended to provide an overview of *Use Case Maps* (UCMs) to people who want to familiarize themselves with this notation. The tutorial is structured as follows:

- Section 2 presents the philosophy behind the notation and the information necessary to use it.
- Section 3 introduces the basic UCM path notation. This section also provides a simple example of how to construct a Use Case Map.
- Section 4 gives a brief overview of the UCM component notation.
- Advanced notational concepts are then presented in Section 5, with a more complex version of the example.
- Finally, main uses and supporting tools are discussed in Section 6.

A summary of the UCM notation is provided as a quick reference guide in Appendix A.

1.2 Scope

This tutorial mainly targets the field of communicating systems and does not address issues related to UCM annotations for performance analysis or goal modeling. Role modeling and dynamic structures will be overviewed superficially, but these topics are better covered in [2] and [3]. Mappings to other formalisms such as LOTOS, SDL, LQN, and UML-RT are not discussed here. The tutorial focuses on the UCMs graphical notation itself, and the underlying semantics models based on hypergraphs and XML are not addressed here (see [1] and [4] for further details on these issues). Style and content guidelines for UCMs remain to be defined and are not discussed in the current version of this tutorial.

2. Aims of Use Case Maps

2.1 Philosophy of UCMs

The Use Case Map notation aims to link behavior and structure in an explicit and visual way. *UCM paths* are first-class architectural entities that describe *causal* relationships between *responsibilities*, which are bound to underlying *organizational structures* of abstract *components* [3]. These paths represent scenarios that intend to bridge the gap between requirements (use cases) and detailed design.

With UCMs, scenarios are expressed above the level of messages exchanged between components, hence they are not necessarily bound to a specific organizational structure. This feature promotes the evaluation of architectural alternatives early in the design process. UCMs provide a bird's-eye, path-centric view of system functionalities, they allow for dynamic behaviour and structures to be represented and evaluated, and they improve the level of reusability of scenarios.

Use Case Maps are primarily visual, but a formal textual representation also exists. Based on the *eXtended Markup Language* (XML) 1.0 standard [6], this representation allows for tools to generate UCMs or use them for further processing and analysis [1].

The notation was developed at Carleton University by Professor Buhr and his team, and it has been used for the description and the understanding of a wide range of complex applications (including telecommunication systems) since 1992. UCMs have raised a lot of interest and hopes in the software community, which led to the creation of a user group at the beginning of 1999, with more than one hundred members from all continents [5].

2.2 Information Needed to Construct UCMs

UCMs can be derived from informal requirements, or from use cases if they are available. *Responsibilities* need to be stated or be inferred from these requirements. For illustration purpose, separate UCMs can be created for individual system functionalities, or even for individual scenarios. However, the strength of this notation mainly resides in the integration of scenarios.

It is important to clearly define the *interface* between the environment and the system under description. This interface will lead to the start points and end points of the UCM paths, and it also corresponds to the messages exchanged between the system and its environment. These messages are further refined in models for detailed design (e.g. with Message Sequence Charts).

Because designers are often the ones who create UCMs, some design information may be relevant. In theory, UCM can be composed of paths where responsibilities are not allocated to any *component*. However, designers are likely to include architectural elements such as internal components. In this case, the description of these components, their nature, and some relationships (e.g., components that include sub-components) are required. Communication links between components are usually not required, but they can be added.

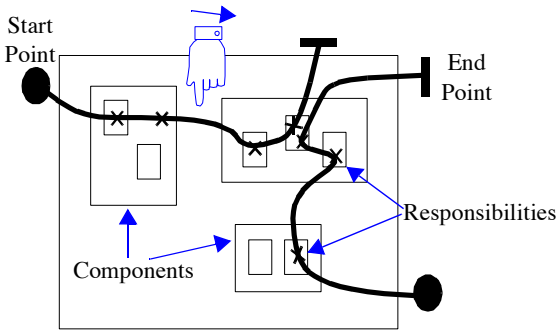
3. Basic UCM Path Notation

3.1 Notation Elements

The UCM notation is mainly composed of **path elements**, and also of **components**. The basic path notation addresses simple operators for causally linking responsibilities in sequences, as alternatives, and in parallel. More advanced operators can be used for structuring UCMs hierarchically and for representing exceptional scenarios and dynamic behavior (Section 5). Components can be of different natures, allowing for a better and more appropriate description of some entities in a system (Section 4).

Figure 1 illustrates four basic elements of UCMs: **start points**, **responsibilities**, **end points**, and **components**. In this section, simple boxes are used as components.

FIGURE 1. Basic Notation and Interpretation



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:

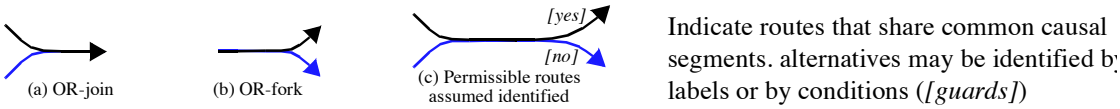
- **start points** (filled circles representing pre-conditions or triggering causes)
- causal chains of **responsibilities** (crosses, representing actions, tasks, or functions to be performed)
- and **end points** (bars representing post-conditions or resulting effects).

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

The wiggly lines are **paths** that connect start points, responsibilities, and end points. A responsibility is said to be *bound* to a component when the cross is inside the component. In this case, the component is responsible to perform the action, task, or function represented by the responsibility. Start points may have preconditions attached, while responsibilities and end points can have postconditions. We call *route* a scenario that traverses paths and associated responsibilities from a start point to an end point.

Alternatives and shared segments of routes are represented as overlapping paths (Figure 2). An **OR-join** merges two (or more) overlapping paths while an **OR-fork** splits a path into two (or more) alternatives. Alternatives may be guarded by conditions represented as labels between square brackets.

FIGURE 2. Shared Routes and OR-Forks/Joins



Concurrent and synchronized segments of routes are represented through the use of a vertical bar (Figure 3). An **AND-join** synchronizes two (or more) paths together while an **AND-fork** splits a path into two (or more) concurrent segments. Cardinalities (*N:M*) are not required to be written as they usually result from the number of incoming/outgoing path segments.

FIGURE 3. Concurrent Routes with AND-Forks/Joins, and Some Variations

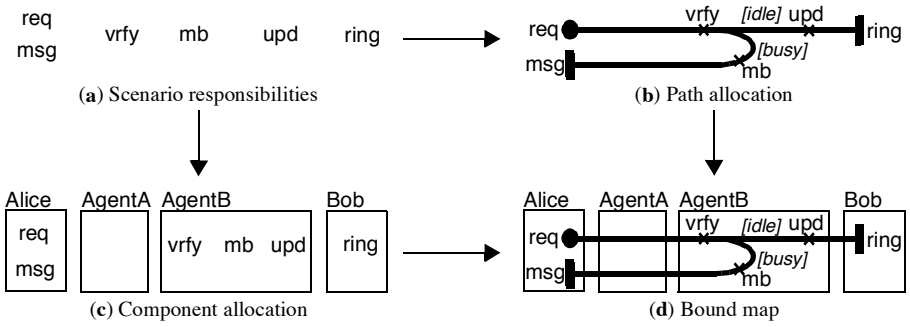
3.2 From Requirements to UCMs: an Example

Figure 4(d) shows a simple UCM where a user (Alice) attempts to call another user (Bob) through some network of agents. Each user has an agent responsible for managing subscribed telephony features such as Originating Call Screening (OCS). Alice first sends a connection request (req) to the network through her agent. This request causes the called agent to verify (vrfy) whether the called party is idle or busy (conditions are between square brackets). If he is, then there will be some status update (upd) and a ring signal will be activated on Bob's side (ring). Otherwise, a message stating that Bob is not available will be prepared (mb) and sent back to Alice (msg).

A scenario starts with a triggering event and/or a pre-condition (start point req) and ends with one or more resulting events and/or post-conditions (end points), in our case ring or msg. Intermediate responsibilities (vrfy, upd, mb) have been activated along the way. In this example, the responsibilities are allocated to abstract components (boxes Alice, AgentA, Bob and AgentB), which could be seen as objects, processes, agents, databases, or even roles, actors, or persons.

The construction of a UCM can be done in many ways. For example, one may start by identifying the responsibilities (Figure 4(a)), although not necessarily with diagrams like this one. Responsibilities can then be allocated to scenarios (Figure 4(b)) or to components (Figure 4(c)). Components can be discovered along the way. Eventually, the two views are merged to form a bound map (Figure 4(d)).

FIGURE 4. Use Case Maps Construction

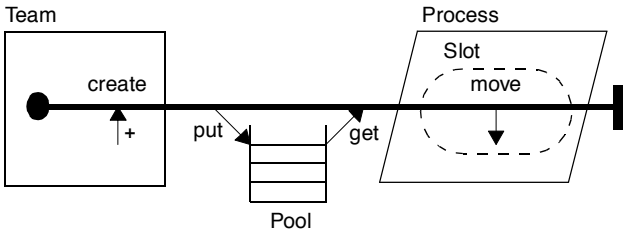


Under an apparent simplicity, UCMs such as Figure 4(d) convey a lot of information in a compact form, and they allow for requirements engineers and designers to use two dimensions (structure and behaviour) to evaluate architectural alternatives for their system.

4. UCM Component Notation

Components can be of different **types** and possess different **attributes**. Although any component notation could be used underneath UCM paths, Buhr suggests several types and attributes relevant for complex systems (real-time, object-oriented, dynamic, agent-based, etc.) [2][3]. The UCM Quick Reference Guide (Appendix A — A8 and A9) illustrates all the component types and attributes in Buhr's notation. Some of the most interesting ones are shown in Figure 5.

FIGURE 5. Dynamic Components and Dynamic Responsibilities



Rectangles are call **teams** and are allowed to contain components of any type. This is a default/generic component used most UCMs. Parallelograms are active components (**processes**) whereas rounded rectangles are passive components (**objects**). Dashed components are called **slots** and may be populated with different instances at different times. Slots are containers for **dynamic components** (DC) in execution, while **pools** are containers for DCs that are not executing (they act as data). Dynamic components can be created, moved, stored, and deleted with **dynamic responsibilities** (see Appendix A — A10) such as create, put, get, and move in Figure 5.

5. Advanced UCM Path Notation

5.1 Advanced Notation Elements

When maps become too complex to be represented as one single UCM, a mechanism for defining and structuring sub-maps becomes necessary. A top-level UCM, referred to as a *root map*, can include containers (called **stubs**) for sub-maps (called **plug-ins**). Stubs are of two kinds (Figure 6):

FIGURE 6. Stubs and Plug-ins

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

(b) Dynamic stubs may have multiple plug-ins

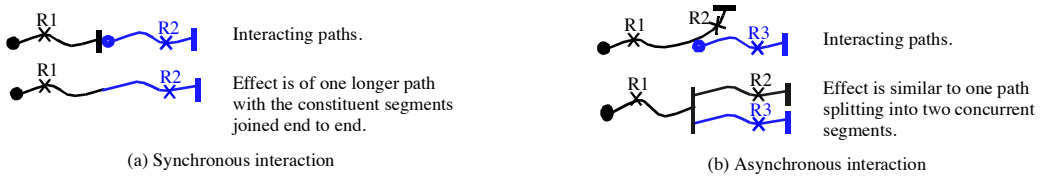
- **Static stubs**: represented as plain diamonds, they contain only one plug-in, hence enabling hierarchical decomposition of complex maps.

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

Path segments coming in and going out of stubs can be identified on the root map. Although they are not required to be shown visually, their presence helps to achieve unambiguous *bindings* of plug-ins to stubs. A binding is a set of couples <*stub_incoming_segment, plug-in_start_point*> and <*stub_outgoing_segment, plug-in_end_point*>. A dynamic stub has one such binding per plug-in.

Different paths may interact with each other synchronously and asynchronously (see Figure 7). **Synchronous interactions** are shown by having the end point of one path touching the start point (or a waiting place) of another path. A path touching the start point (or a waiting place) represents an **asynchronous interaction**.

FIGURE 7. Path Interactions



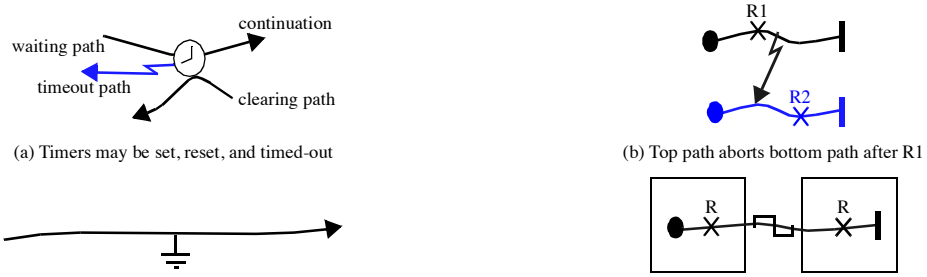
(a) Synchronous interaction

(b) Asynchronous interaction

Other notational elements include (Figure 8):

- **Timer**: special waiting place triggered by the timely arrival of a specific event. It can also trigger a time-out path when this event does not arrive in time.
- **Abort**: a path can terminate the execution of another causal chain of responsibilities.
- **Failure point**: indicates potential failure points on a path.
- **Shared responsibility**: represents a complex activity that involves negotiation between two or more components.

FIGURE 8. Timers, Aborts, Failures, and Shared Responsibilities



(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

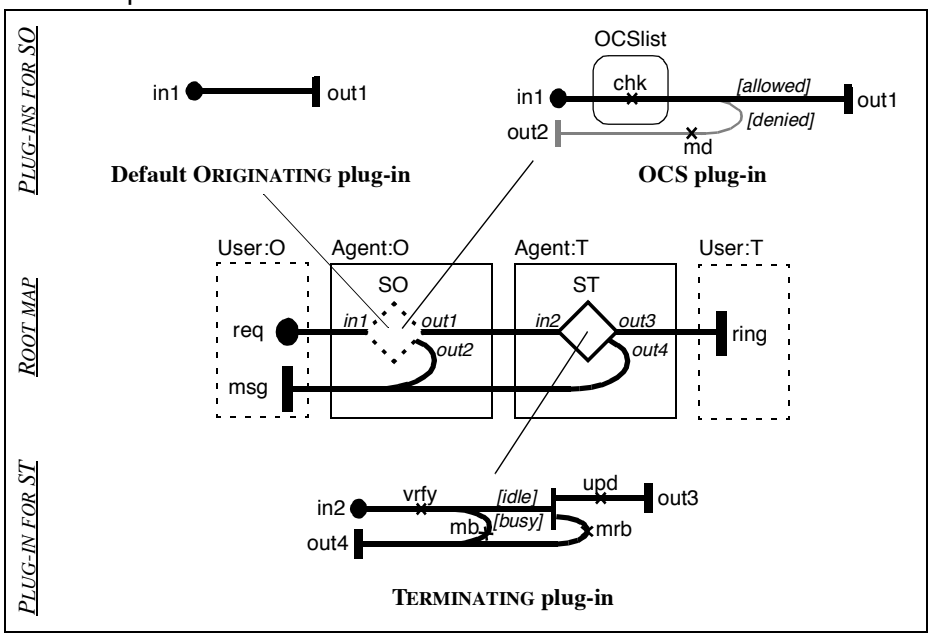
(d) R is a shared responsibility

Finally, the notation supports additional extensions specific to agent systems and to performance modeling. These extensions are not addressed in this tutorial, but their respective path annotations are included in the quick reference guide (Appendix A — A11).

5.2 Example Revisited

New features can be added to the simple example presented previously. Figure 9 abstracts from the component instances introduced in Figure 4. The components do not refer to Bob and Alice any longer, but they refer to more generic call origination and termination roles (for both users and agents). UserO and UserT are slots that can be filled with particular instances of users. The middle part of Figure 9 shows an enhanced version of the UCM from Figure 4(d). This root map represents a whole class of related use case instances.

FIGURE 9. More Complex Call Connection and New Notation Elements



The originating dynamic stub SO has two plug-ins (ORIGINATING and OCS). The start point of the ORIGINATING plug-in (in1) is bound to the incoming path segment in1, and the end point out1 is bound to the outgoing segment out1.

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

The TERMINATING plug-in improves on the original UCM by allowing the update (upd) and the ring result to be accompanied, concurrently, by the preparation of a ring-back targeted to the originating party (mrb).

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

6. Using UCMs

6.1 Main Uses

Use Case Maps are used to describe and integrate use cases representing the requirements. The construction of UCMs can reveal problems with the use cases, which may be incomplete, incorrect, ambiguous, inconsistent, or at different levels of abstraction.

UCMs include high-level design information (internal responsibilities and components), but they do not commit to messages between components (in contrast with MSCs), so they are more easily maintainable as design scenarios.

UCMs excel at integrating individual features through the use of stubs and plug-ins, while at the same time allowing for reasoning about potential undesirable interactions.

UCMs are not executable as is, but they can be manually translated to a model that allows for fast prototyping and validation. LOTOS is especially well suited for representing UCMs. Mappings to hierarchical finite state machines (used in UML-RT) and to Layered Queuing Networks (for performance modeling) also exist.

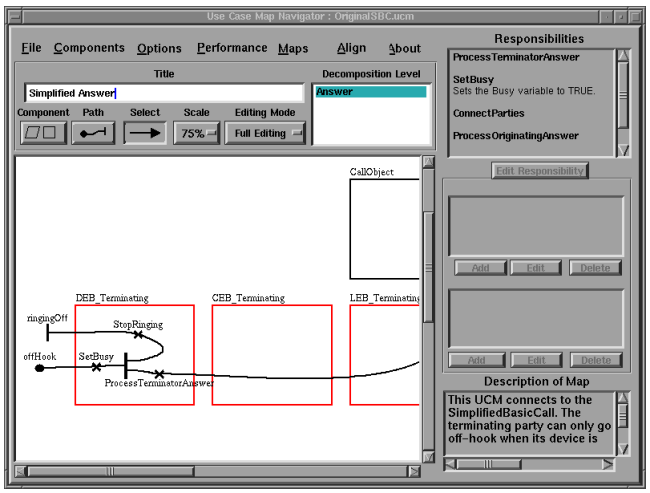
UCMs can also serve as a basis for the definition of abstract validation test suites based on the design. This represents a level of completeness different from (and often better than) plain functional testing.

Finally, the use of the UCM Navigator tool enables the automated generation of documentation and of XML code, which can be processed for further analysis and potentially for partial generation of formal models.

6.2 UCM Tools

There currently exists only one tool that supports the UCM notation and the XML format: the *UCM Navigator* (UCMNAV) [4]. Although still a prototype under development, this tool is already robust enough for the creation and maintenance of UCMs. The path and component notations are fully supported (Figure 10). UCMNAV ensures the syntactical correctness of the UCMs manipulated, generates XML descriptions, exports UCMs in Encapsulated Postcript or Maker Interchange Format (for Adobe's FrameMaker) formats, and generates reports in PostScript (enabled for Adobe's Portable Document Format).

FIGURE 10.The Use Case Maps Navigator (UCMNAV)



Main Features of the UCM Navigator

- Maps are always syntactically correct
- Path transformations and connections based on internal hypergraph-based semantics
- Nested level of stubs and plug-ins (sub-maps), with binding
- Export/import mechanism for UCM integration
- Intelligent binding of paths to structural components (paths and responsibilities move with components)
- UCM extensions for agent goals and performance modeling
- Descriptions and pre/post condidions attached to elements
- Generation of XML files valid w.r.t. UCM DTD
- Flexible report generation in PostScript, ready for the generation of hyperlinked and indexed PDF documents
- GUI with scalable maps, zooms, and scrool bars
- Export of maps in EPS and in FrameMaker's MIF formats
- Four platforms supported (Linux-Intel, Linux-Sparc, Solaris, HP/UX), with a fifth one on the way (Windows NT)

Alternatively, any drawing package or word processors (such as FrameMaker or Microsoft Word) could be used to draw UCMs. However, syntactic errors may be introduced in the UCMs, and no XML code can be generated.

7. References

The UCM book [2] and the IEEE TSE paper [3] represent more complete tutorials on the UCM notation, with examples. The UCM Navigator tool, together with examples, manuals, and design information, can be found in [4]. The XML format generated by this tool is defined in [1]. Finally, the *UCM User Group* and other useful information on UCMs (including a virtual library) can be found in [5].

[1] Amyot, D. and Miga, A.: *Use Case Maps Linear Form in XML, version 0.13*, May 1999. http://www.UseCaseMaps.org/UseCaseMaps/xml/

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

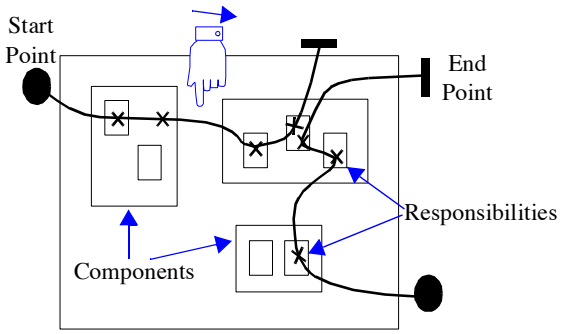
[3] Buhr, R.J.A.: "Use Case Maps as Architectural Entities for Complex Systems". In: *Transactions on Software Engineering*, IEEE, December 1998, pp. 1131-1155. http://www.UseCaseMaps.org/UseCaseMaps/pub/tse98final.pdf

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

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

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

Appendix A: UCM Quick Reference Guide

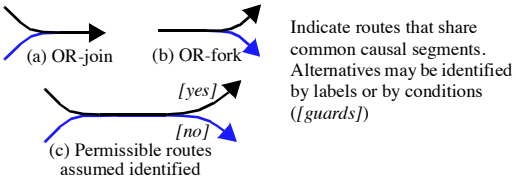


Imagine tracing a path through a system of objects to explain a causal sequence, leaving behind a visual signature. Use Case Maps capture such sequences. They are composed of:

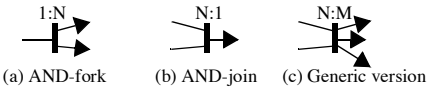
- **start points** (filled circles representing pre-conditions or triggering causes)
- causal chains of **responsibilities** (crosses, representing actions, tasks, or functions to be performed)
- and **end points** (bars representing post-conditions or resulting effects).

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

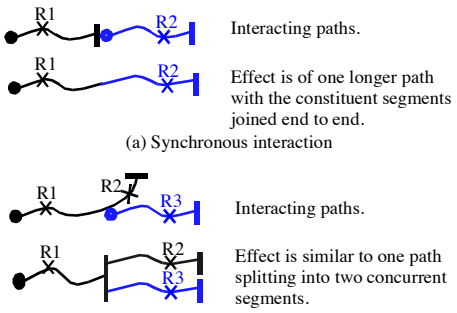
A1. Basic notation and interpretation



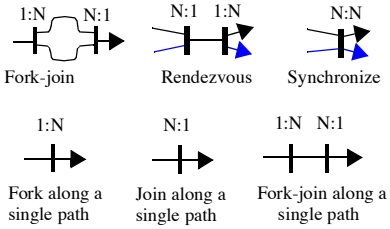
A2. Shared routes and **OR-forks/joins**.



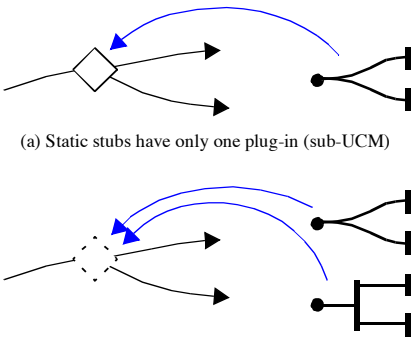
A4. Concurrent routes with **AND-forks/joins**.



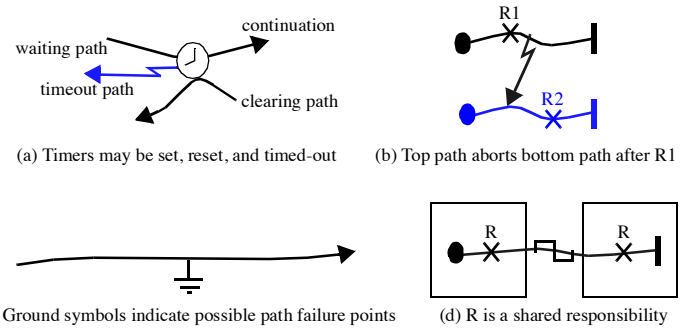
A3. Path interactions.



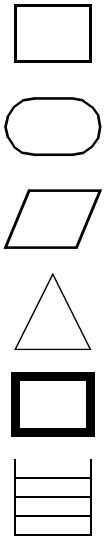
A5. Variations on AND-forks/joins.



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



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



(a) **Team**: generic container

(b) **Object**: passive component

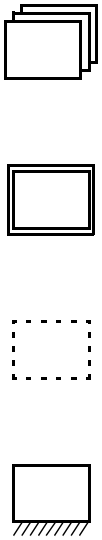
(c) **Process**: active component

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

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

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

A8. Component types.



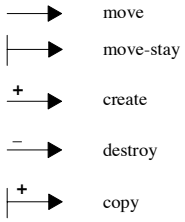
(a) **Stack**: multiple instances

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

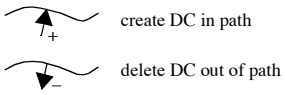
(c) **Slot**: placeholder for dynamic components as operational units

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

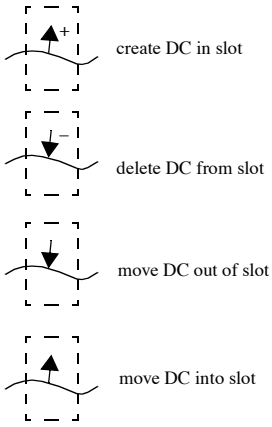
A9. Component attributes.



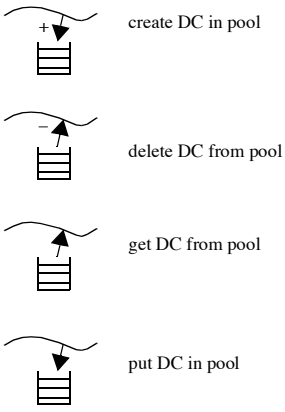
(a) Movement of DCs as data



b) Directly into or out of paths

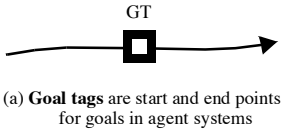


(c) Into or out of slots

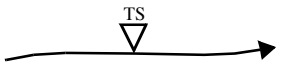


(d) Into or out of pools

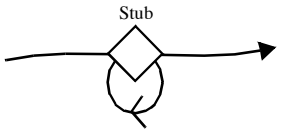
A10. Movement notation for **dynamic components** (DCs).



(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