Annotated UCM Linear Form

The following document presents the Use Case Maps linear form in terms of annotated EBNF rules. The following conventions are used:

Diagrams

• $\boxed{\text{Rectangles}}$ (with plain font) :	rules
• **Ellipses** (with **bold** font) :	keywords
• *Rounded-corner rectangles* (with *italic* font):	litterals

EBNF Rules

- Terms in plain font : rules
- **Terms** in **bold** font : keywords
- *Terms* in *italic* font : litterals

• [Terms] :	Term is optional
• (Terms)* :	Zero or more Term
• (Terms)+ :	One or more Term
• Terms | Terms :	Term1 or Term2

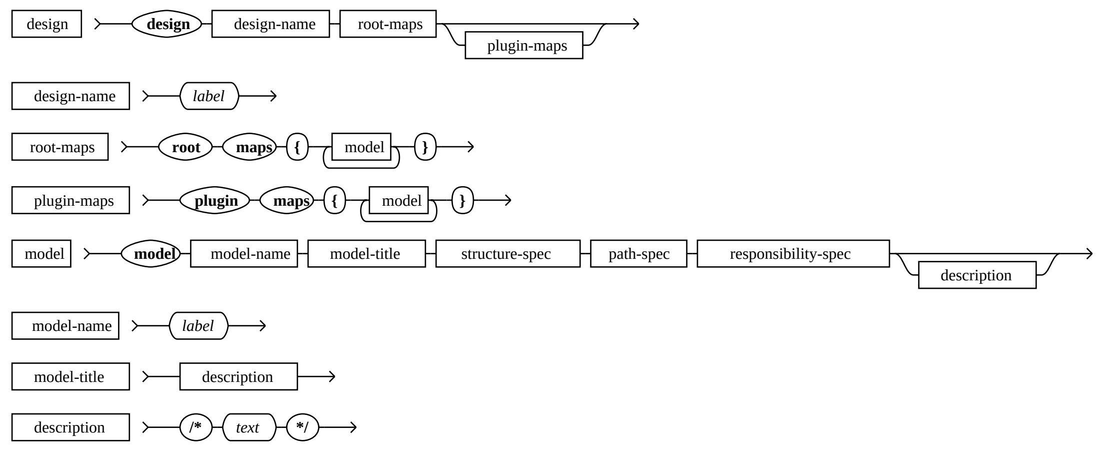
As for litterals, three of them are used in this document:

text :	Sequence of any visible ASCII character (including space) without “*/”.
label :	Sequence of -, _, a..z, A..Z, 0..9.
unsigned-integer :	Sequence of 0..9.

Java-style comments (“//” format) will be allowed in linear form descriptions. For example:

```
model MyModel  // This is a comment.
```

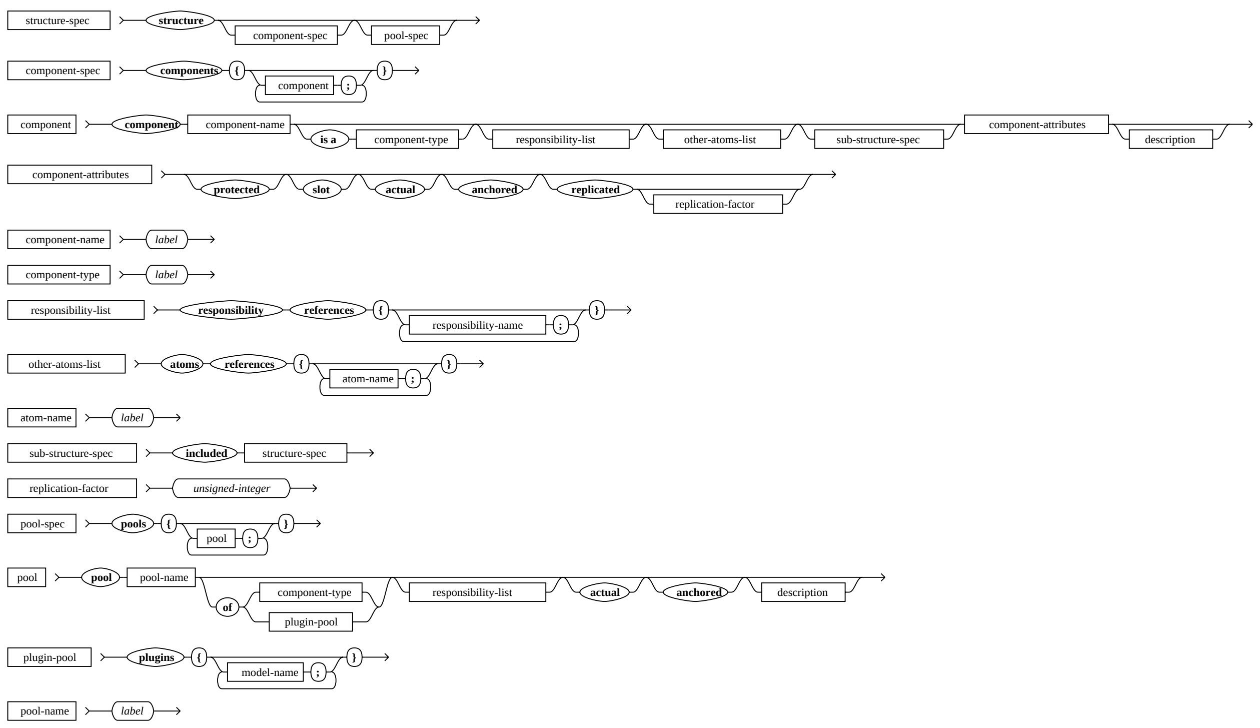
1. Design and Model



design ::=	**design** design-name root-maps [plugin-maps]	*(R1)*
design-name ::=	*label*	*(R2)*
root-maps ::=	**root maps {** (model)+ **}**	*(R3)*
plugin-maps ::=	**plugin maps {** (model)+ **}**	*(R4)*
model ::=	**model** model-name model-title structure-spec path-spec responsibility-spec [description]	*(R5)*
model-name ::=	*label*	*(R6)*
model-title ::=	description	*(R7)*
description ::=	**/*** *text* ***/**	*(R8)*

A design is composed of a collection of root (top-level) maps and, possibly, of a collection of plugin maps. Each map is a model. A UCM model can be defined by the 4-tuple $u = (n, s, p, r)$ where n is the name of the model, s is the specification of the structural aspect of the model, p is the specification of the path aspect of the model and r is the specification of responsibilities which are referenced in s and p. Every named construction in this grammar has the possibility to hold a free format text description.

2. Structure Specification


structure-spec structure component-spec pool-spec
component-spec components { component ; }
component component component-name is a component-type responsibility-list other-atoms-list sub-structure-spec component-attributes description
component-attributes protected slot actual anchored replicated replication-factor
component-name label
component-type label
responsibility-list responsibility references { responsibility-name ; }
other-atoms-list atoms references { atom-name ; }
atom-name label
sub-structure-spec included structure-spec
replication-factor unsigned-integer
pool-spec pools { pool ; }
pool pool pool-name of component-type plugin-pool responsibility-list actual anchored description
plugin-pool plugins { model-name ; }
pool-name label

structure-spec ::= **structure** [component-spec] [pool-spec] *(R9)*

The structure specification gives the relationship between the structural entities and the location at which responsibilities are executed. The structural entities of a UCM model can be divided in two differents categories: components and pools. The former can represent a process, an object, a team, or other structures. Components, such as teams, may act as containers that hold instances of other components. Pools, by themselves, cannot perform responsibilities other than move components in or out of a path.

component-spec ::= **components {** (component **;**)* **}** *(R10)*

A model may or may not contain components.

component ::= **component** component-name [**is a** component-type]
[responsibility-list] [other-atoms-list] [sub-structure-spec]
component-attributes [description] *(R11)*

component-attributes ::=[**protected**] [**slot**] [**actual**] [**anchored**]
[**replicated** [replication-factor]] *(R12)*

A component is identified by a name and can be attributed an arbitrary type (such as process, team, etc). It also holds a list of responsibility references. These references point to the specification given later on (responsibility-spec). A component can be decomposed in a lower level structure and can be given properties like protected, slot, actual, anchored, and replicated. If a fixed quantity of a component is known in advance, it can be indicated by a replication factor.

component-name ::= *label* *(R13)*

component-type ::=	*label*	*(R14)*
responsibility-list ::=	**responsibility references {** (responsibility-name **;**)* **}**	*(R15)*
other-atoms-list ::=	**atoms references {** (atom-name **;**)* **}**	*(R16)*
atom-name ::=	*label*	*(R17)*

Responsibility references list names of responsibilities explicitly specified later on. References tto other atoms within the component are also considered.

sub-structure-spec :=	**included** structure-spec	*(R18)*

The decomposition of a component points back to a structure that can hold components and pools.

replication-factor ::=	*unsigned-integer*	*(R19)*
pool-spec ::=	**pools {** (pool **;**)* **}**	*(R20)*

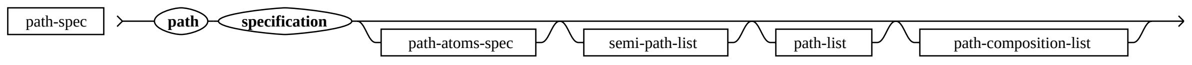
A model may or may not contain pools.

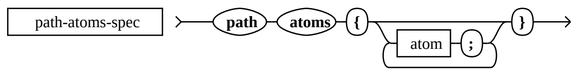
pool ::=	**pool** pool-name [**of** (component-type | plugin-pool)] [responsibility-list] [**actual**] [**anchored**] [description]	*(R21)*
plugin-pool ::=	**plugins {** (model-name **;**)* **}**	*(R22)*

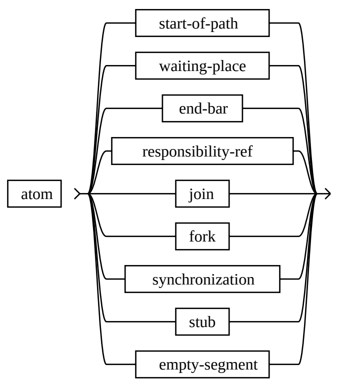
A pool is identified by a name and can be attributed an arbitrary component type (such as process, team, etc) or a list of plugins for dynamic stubs. It can be actual or formal, and anchored or not. A pool also holds a list of responsibility references. These references point to the specification given later on (responsibility-spec). Pools mainly serve the dynamic self-configuring aspect of UCMs.

pool-name ::=	*label*	*(R23)*

3. Path Specification







path-spec ::= **path specification** [path-atoms-spec] [semi-path-list] [path-list] [path-composition-list] *(R24)*

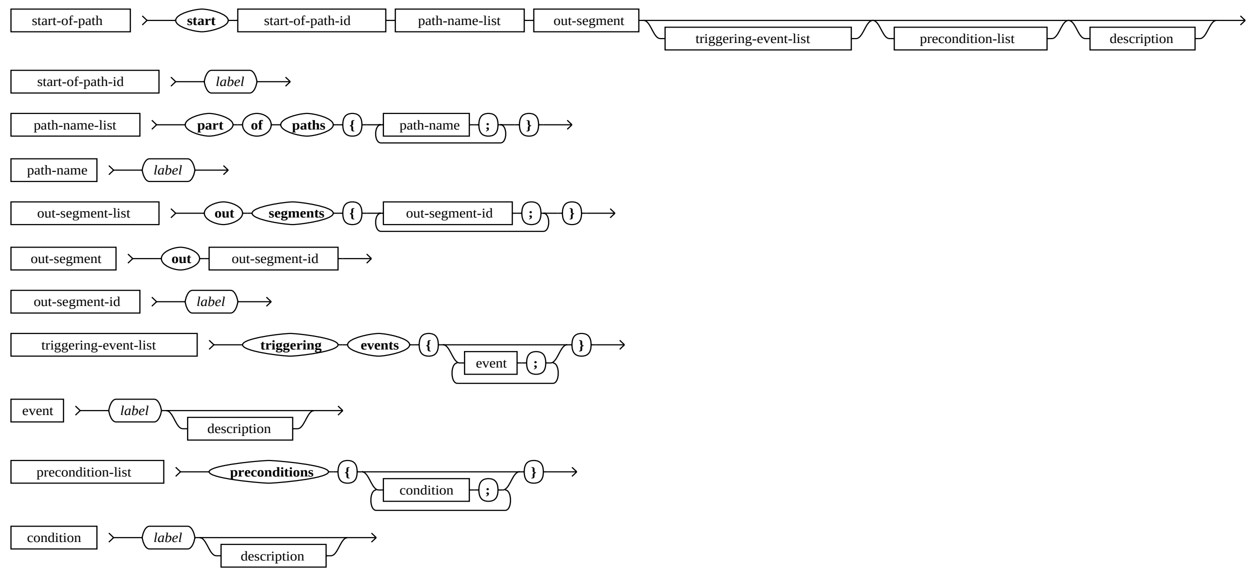
The path specification of a model details the differents scenarios and the causal sequences of a map. The path atoms specify all the parts that make up a path. The semi-path list gives the partial view of the sequencing of these atoms. The path list gives the full view of the sequencing of the atoms since the semi-path are then listed in sequence to create a complete path. The path composition list indicates how path related with one another. Semi-paths are necessary due to the ambiguity of unlabeled maps. Some atoms (stubs for example) hide the continuity of a path thus allowing several possibilities for path trajectory. Only explicit path labeling can eliminate this ambiguity. Since path labeling is optional at the time of creation, the semi-path mechanism is required to capture unlabelled maps.

path-atoms-spec ::= **path atoms {** (atom **;**)* **}** *(R25)*

A model may or may not contain path atoms.

atom ::= start-of-path | waiting-place | end-bar | responsibility-ref | join | fork | synchronization | stub | empty-segment *(R26)*

3.1 Atom: start-of-path



start-of-path ::=	**start** start-of-path-id path-name-list out-segment [triggering-event-list] [precondition-list] [description]	*(R27)*

A start of path is identified by a name and can be part of several paths at once. It has one labelled *out* segment. The triggering event list gives the set of events that initiate the sequence of actions in a path. The precondition list must be satisfied in order for the sequence to start.

start-of-path-id ::=	*label*	*(R28)*
path-name-list ::=	**part of paths {** (path-name **;**)+ **}**	*(R29)*

A path atom is always part of at least one path.

path-name ::=	*label*	*(R30)*
out-segment-list ::=	**out segments {** (out-segment-id **;**)+ **}**	*(R31)*
out-segment ::=	**out** out-segment-id	*(R32)*
out-segment-id ::=	**out** *label*	*(R33)*

An out-segment is a point where a following path atom can connect. It is uniquely identified and is referred to in the semi-path definitions.

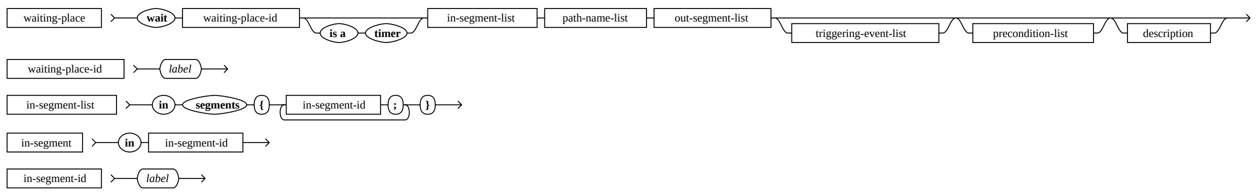
triggering-event-list ::=	**triggering events {** (event **;**)* **}**	*(R34)*

A start of path or waiting place may or may not have triggering events.

event ::=	*label* [description]	*(R35)*
precondition-list ::=	**preconditions {** (condition **;**)* **}**	*(R36)*
condition ::=	*label* [description]	*(R37)*

A start of path or waiting place may have preconditions. A condition is a label with an optional description.

3.2 Atom: waiting-place



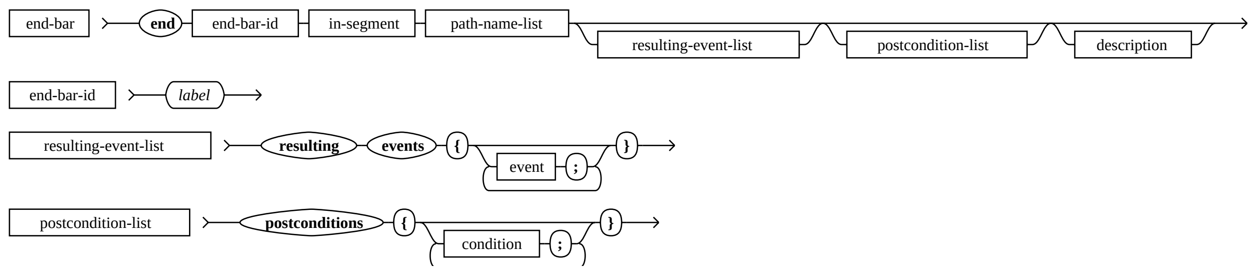
waiting-place ::= **wait** waiting-place-id [**is a timer**] in-segment-list path-name-list out-segment-list [triggering-event-list] [precondition-list] [description] *(R38)*

A waiting place is identified by a name and can be part of several paths at once. It can have several out segments and in segments. The triggering event list gives the set of things that restart the sequence of actions in a path. The precondition list gives the conditions that must be satisfied in order for the sequence to restart. A waiting place can be timer.

waiting-place-id ::= *label* *(R39)*
in-segment-list ::= **in segments {** (in-segment-id **;**)+ **}** *(R40)*
in-segment ::= **in** in-segment-id *(R41)*
in-segment-id ::= *label* *(R42)*

An *in* segment is a point where a preceeding path atom can connect. It is uniquely identified and is referred to in the semi-path definitions.

3.3 Atom: end-bar



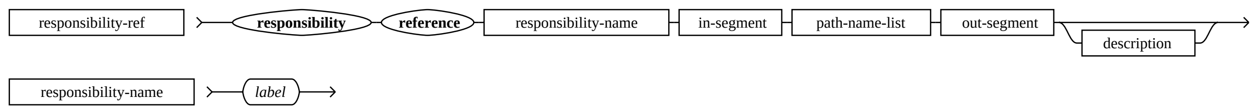
end-bar ::= **end** end-bar-id in-segment path-name-list [resulting-event-list] [postcondition-list] [description] *(R43)*

An end of path is identified by a name and can be part of several paths at once. It has one labelled *in* segment. The resulting event list gives the set of things that occur once the sequence of actions in a path are completed. The postcondition list gives conditions that must be satisfied once the sequence is completed.

end-bar-id ::= *label* *(R44)*
resulting-event-list ::= **resulting events {** (event **;**)* **}** *(R45)*
postcondition-list ::= **postconditions {** (condition **;**)* **}** *(R46)*

An end of path may or may not have resulting events, and it may or may not have postconditions.

3.4 Atom: responsibility-ref

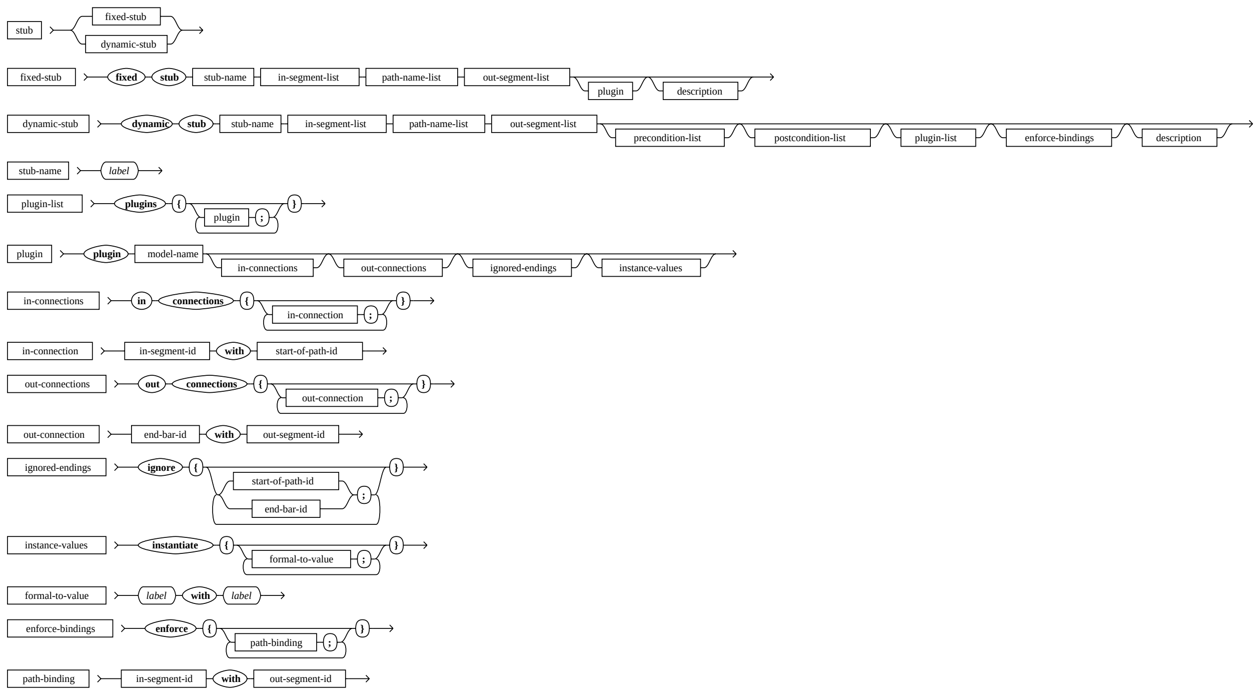


responsibility-ref ::= **responsibility reference** responsibility-name in-segment path-name-list out-segment [description] *(R47)*

A responsibility reference only merely places a responsibility along a path. It is not its specification. Its name refers to a specified responsibility. It has only one *out* segment and one *in* segment. It can be part of several paths.

responsibility-name ::= *label* *(R48)*

3.5 Atom: stub



stub ::= fixed-stub | dynamic-stub *(R49)*

fixed-stub ::= **fixed stub** stub-name in-segment-list path-name-list out-segment-list [plugin] [description] *(R50)*

dynamic-stub ::= **dynamic stub** stub-name in-segment-list path-name-list out-segment-list [precondition-list] [postcondition-list] [plugin-list] [enforce-bindings][description] *(R51)*

A stub is identified by name, it can have several *in* segments and *out* segments. Since a stub is an abstraction of a sub-map, it also has one plug-in when fixed, and a plug-in list when dynamic. The paths that go through a stub need to be bound to the paths of the plug-in in order to ensure continuity. This is done through explicit binding. The binding is declared in the definition of the stub rather than at the plug-in level since the plugin only makes sense in the context of the stub. For dynamic stubs, the runtime binding occurs only if the pre- and postconditions of the stub are satisfied by those of the plug-in.

stub-name ::=	*label*	*(R52)*
plugin-list ::=	**plugins {** (plugin **;**)* **}**	*(R53)*

A stub definition may or may not contain plugins.

plugin ::=	**plugin** model-name [in-connections] [out-connections] [ignored-endings] [instance-values]	*(R54)*

A plugin refers to a UCM model that can replace the stub. The binding of the two is defined by the *in* connections, the *out* connections, and the ignored endings.

in-connections ::=	**in connections {** (in-connection **;**)+ **}**	*(R55)*

There must be at least one *in* connection.

in-connection ::=	in-segment-id **with** start-of-path-id	*(R56)*

An *in* connection joins an *in* segment of the stub with a start of path from the plugin map.

out-connections ::=	**out connections {** (out-connection **;**)+ **}**	*(R57)*

There must be at least one *out* connection.

out-connection ::=	end-bar-id **with** out-segment-id	*(R58)*

An *out* connection joins an *out* segment of the stub to an end of path of the plugin map.

ignored-endings ::=	**ignore {** ((start-of-path-id | end-bar-id) **;**)* **}**	*(R59)*

Ignored endings specify the paths of the plugin that are not concerned or that should be ignored when the binding occurs. Any other paths from the plugin that have not been mentionned in the binding are considered local paths that may execute but do affect the map that holds the stub.

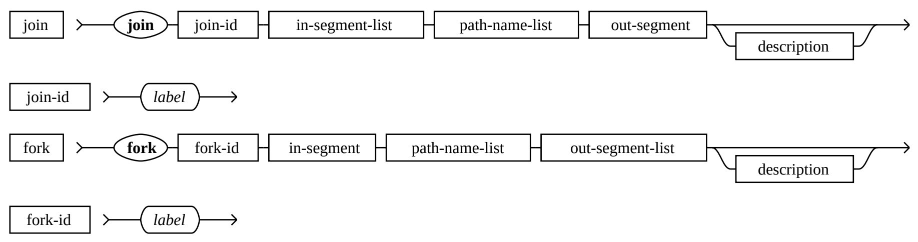
instance-values ::=	**instantiate {** (formal-to-value **;**)* **}**	*(R60)*
formal-to-value ::=	*label* **with** *label*	*(R61)*

Formal parameters within the plug-in model (such as formal components) can be instantiated, with a value, at run-time when the plug-in is selected.

enforce-bindings ::=	**enforce {** (path-binding **;**)* **}**	*(R62)*
path-binding ::=	in-segment-id **with** out-segment-id	*(R63)*

Bindings can also be enforced at stub definition time in order to preserve path continuity in the stub. Plug-ins that do not satisfy this constraint will not be selected.

3.6 Atoms: join and fork



join ::=	**join** join-id in-segment-list path-name-list out-segment [description]	*(R64)*

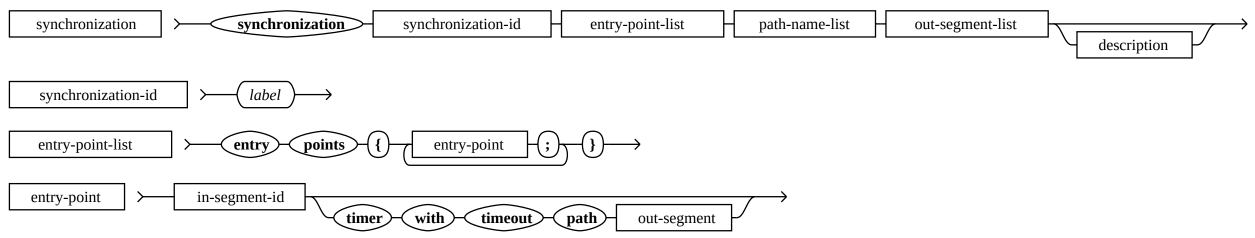
A join is identified by a name and has several *in* segments and only one *out* segment.

join-id ::=	*label*	*(R65)*
fork ::=	**fork** fork-id in-segment path-name-list out-segment-list [description]	*(R66)*

A fork is identified by a name and has several *out* segments and only one *in* segment.

fork-id ::=	*label*	*(R67)*

3.7 Atom: synchronization



synchronization ::=	**synchronization** synchronization-id entry-point-list path-name-list out-segment-list [description]	*(R68)*

A synchronization (sometimes called *and-join* or *and-fork*) is identified by a name and has several *out* segments. In the case of this atom, the *in* segments are replaces by entry points because the *in* segments can be enriched with a timeout capacity.

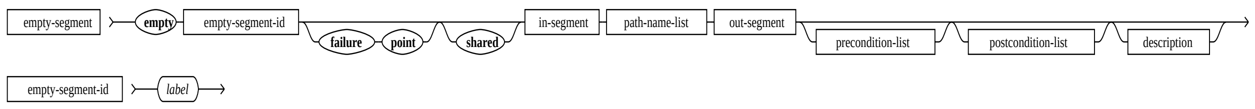
synchronization-id ::=	*label*	*(R69)*
entry-point-list ::=	**entry points {** (entry-point **;**)+ **}**	*(R70)*

A synchronization has at least one entry point.

entry-point ::=	in-segment-id [**timer with timeout path** out-segment]	*(R71)*

An entry point is an *in* segment that can have a timeout property. With this property comes an *out* segment declaration for the case where the timeout occurs.

3.8 Atom: empty-segment

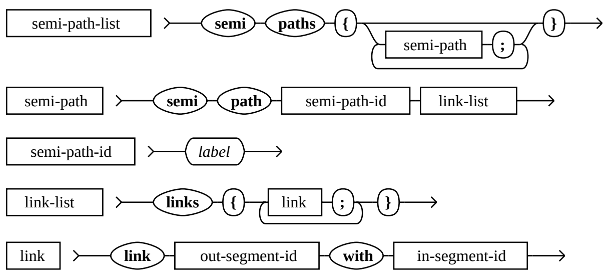


empty-segment ::= **empty** empty-segment-id [**failure point**] [**shared**] in-segment path-name-list out-segment [precondition-list] [postcondition-list] [description] *(R72)*

An empty segment is used to add attributes to a path such as the indication of a failure point or that the responsibilities preceeding and following it are shared. It has one *in* segment and one *out* segment. It might also contain preconditions for the next atom or postconditions of the previous atom. This feature is especially useful for conditions associated to stubs and goals (paths that cross components) when UCMs describe agent systems.

empty-segment-id ::= *label* *(R73)*

4. Semi-Path Construction



semi-path-list ::= **semi paths {** (semi-path **;**) * **}** *(R74)*

A model may or may not have semi-paths.

semi-path ::= **semi path** semi-path-id link-list *(R75)*

semi-path-id ::= *label* *(R76)*

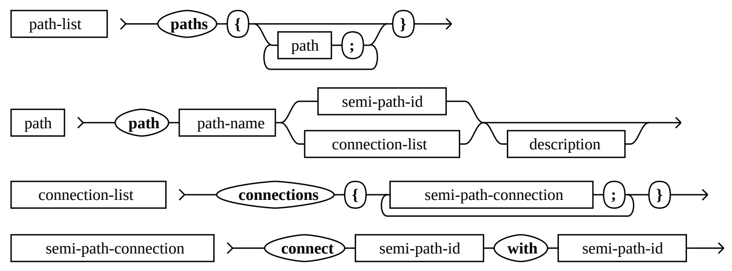
A semi-path is a set of contiguous atoms that form an unambiguous partial path. The rules used to delimit a semi-path are :

- an ambiguous atom is one that has more than one out segment or in segment.
- a semi-path starts with a start of path or with an ambiguous atom.
- a semi-path ends with an end of path or with an ambiguous atom.
- a semi-path cannot hold an ambiguous atom that is not at its extremities.
- the legal regular expression for a semi-path would be :
  "(start-of-path | ambiguous-atom) unambiguous-atom* (end-of-path | ambiguous-atom)"
- two contiguous semi-paths share the ambiguous atom (this atom acts as the link between the two).

link-list ::= **links {** (link **;**)+ **}** *(R77)*

link ::= **link** out-segment-id **with** in-segment-id *(R78)*

5. Path Construction



path-list ::= **paths {** (path **;**)* **}** *(R79)*

A model may or may not contain paths.

path ::= **path** path-name (semi-path-id | connection-list) [description] *(R80)*

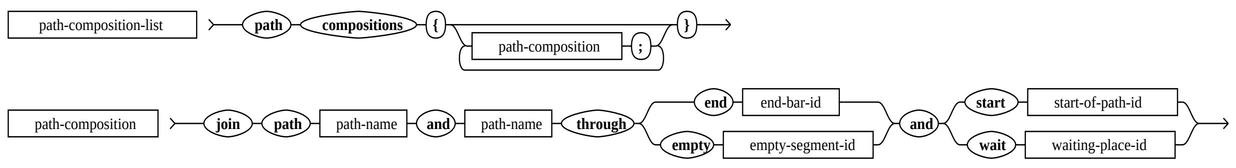
A path is identified by a name and can be composed of a connection list, which links semi-paths, or it can simply refer to a semi-path.

connection-list ::= **connections {** (semi-path-connection **;**)+ **}** *(R81)*

A connection list is made of at least one semi-path connection.

semi-path-connection ::= **connect** semi-path-id **with** semi-path-id *(R82)*

6. Path Composition



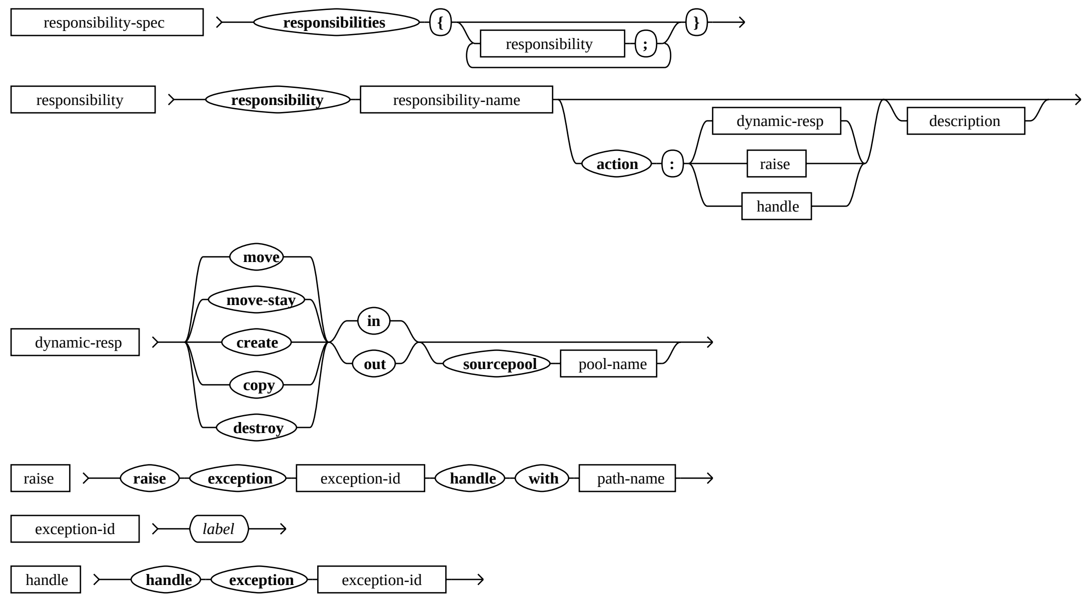
path-composition-list ::= **path compositions {** (path-composition **;**)* **}** *(R83)*

A model may or may not contain path compositions.

path-composition ::= **join path** path-name **and** path-name **through** (**end** end-bar-id | **empty** empty-segment-id) **and** (**start** start-of-path-id | **wait** waiting-place-id) *(R84)*

A path composition is the junction of two paths. The triggering path uses an end bar (synchronous interaction) or an empty segment (in-passing interaction) on the start point or a waiting place of the triggered path.

7. Responsibility Specification



responsibility-spec ::= **responsibilities {** (responsibility **;**)* **}** *(R85)*

A model may or may not contain responsibilities.

responsibility ::= **responsibility** responsibility-name [**action :** (dynamic-resp | raise | handle)] [description] *(R86)*

A responsibility specification uniquely identifies a responsibility which can then be refered to in the structure of the model, in the path definitions, or in both.

dynamic-resp ::= (**move** | **move-stay** | **create** | **copy** | **destroy**) (**in** | **out**) [**sourcepool** pool-name] *(R87)*

A dynamic responsibility performs an action on data/components/plug-ins, in or out of a path.

raise ::= **raise exception** exception-id **handle with** path-name *(R88)*

The raise exception responsibility indicates that an exception will be handled by another path.

exception-id ::= *label* *(R89)*
handle ::= **handle exception** exception-id *(R90)*

The handle responsibility indicates that it handles the exception of another path.