# On State Machine Inheritance in UML

**Bran Selić** (version Nov. 17, 2015)

## The Requirement

It is common in UML to use state machines to describe the classifier behavior of complex active classes. Since UML is an object-oriented language that supports specialization of classes using an inheritance-based mechanism, it is natural to expect to be able to specialize UML classes whose behaviors happen to be specified by UML state machines.

This requirement is illustrated using the example state machine shown in Figure 1, which represents the classifier behavior of some (active) UML class (not shown explicitly in the diagram)[[1]](#footnote-1). Being a classifier behavior means that this state machine is a member of the namespace of that class. Because the classifier behavior of a UML class qualifies as an “inheritable member” of that class[[2]](#footnote-2), it will be inherited by any subclass of the owning UML class[[3]](#footnote-3).



1. An “abstract” state machine behavior

Assume that, in this particular case, the class that owns this state machine is intended as an abstract class and that its classifier behavior state machine describes a general high-level (i.e., “abstract”) behavior that is intended to be shared by all subclasses of that class. Starting with this basic behavior, each subclass will need to refine this high-level behavior depending on its particular purpose. For example, each subclass may specify different behavior for its instances when they are in the **Operational** state.

One elegant way to achieve this would be for a subclass to “inherit” the state machine of its parent and then to refine some of its states and transitions using UML’s redefinition mechanism while also adding new ones to suit the specific needs of the subclass. A prototypical example of such refinement for the case of the state machine in Figure 1 is depicted in Figure 2.



1. A desired refinement (**SM1’**) of the state machine **SM1** specified in Figure 1

In this case, we have refined the original simple **Operational** state from the general state machine, by redefining with an application-specific *composite* state, **Operational’**, with an internal state machine of its own. Note that, to retain behavioral compatibility, we have retained the two original transitions (“**go**” and “**error**”) associated with the general state machine and the states **Ready** and **Broken**. In addition to this redefinition, for this particular refinement we have included one additional state (**Recovering**) along with its associated transitions. (For clarity, the inherited elements are rendered using lighter coloured lines and text, while the new subclass-specific elements are drawn in red. However, as explained below, this diagram is not a fully accurate rendering of the actual extending state machine.)

## State Machine Specialization

To achieve this type of inheritance, a special mechanism was provided in UML 2.0. It is based on the notion of “extending” one state machine with another. By definition, an *extending* (i.e., specializing) state machine has *all* the elements of the *extended* (i.e., general) state machine, except that (a) it may add new elements and (b) redefine certain types of existing ones[[4]](#footnote-4). Of course, these are the very same principles that apply to standard inheritance in most conventional object-oriented languages.

To support the kind of redefinition exemplified by the state **Operational’** in Figure 2, UML allows the following three kinds of state machine members to be redefined:

* Regions
* States
* Transitions

This is achieved by making them subclasses of the abstract **UML::RedefinableElement** class. Note that, although it is required that a redefining element must be “consistent with” the element that is being redefined, UML is quite flexible in how such compatibility is defined. In section 9.2.3.3 of [1] it is stated:

“The detailed semantics of redefinition vary for each specialization of RedefinableElement. There are various kinds of compatibility between a redefined element and its redefining element…”

This suggests that the intent was that **UML::RedefinableElement** should be specialized for specific cases. Unfortunately, this was not done for state machines, making it difficult to associate case-specific semantics for this particular case. Furthermore, no explanation was provided for what those semantics should be[[5]](#footnote-5). As it turns out, for this mechanism to work as intended, an idiosyncratic interpretation of redefinition is necessary.

To see why, consider the transition triggered by the **recover** event in Figure 2, which originates on state **Broken** and terminates on state **Recovering**. While state **Recovering** is defined in the extending state machine (**SM1’**), state **Broken** is defined in the extended state machine (**SM1**). However, according to the UML abstract syntax rules (i.e., the UML metamodel), a state machine transition cannot be connected between vertices that *belong to two different state machines*, as would be the case here. To avoid this problem, it is therefore necessary to add a *new source state* for the transition (call it **Broken’**). This new state has to redefine the original state **Broken** in order to make the proper semantic connection (transitions cannot be left “dangling” but must be anchored at both ends). Furthermore, again due to constraints imposed by the abstract syntax of UML, both the **Broken’** state and the new transition must be owned by a region that is part of the same state machine, which requires that a new region must be defined that redefines the region in of the original **Broken** state. Finally, this new region will be owned by the extending state machine **SM1’**. The necessary extensions are shown in Figure 3, using blue lines.



1. The correct extensions, additions, and redefinitions for state machine **SM1’**

The result is that, whenever a new member (e.g., region, vertex, or transition) is added, all the containers in the transitive closure of container elements in which that element is defined have to be redefined (except, of course, for the top-level state machine container, which is “extended”). Furthermore, if a transition is added, its source and target vertices need to be either redefined or added (if they are new elements), so that they have the proper **incoming** and **outgoing** transition collection features.

But, even this is not enough. Note that the redefinition mechanism for regions and states shown in Figure 3 assumes that the redefining element “inherits” all of the members of the redefined element[[6]](#footnote-6). Thus, state **Operational’** must inherit the **incoming** and **outgoing** transition features of its redefined element and, in addition, merge its own incoming and outgoing transition elements into these.

One final issue remains to make this approach workable: Since transitions do not always originate or terminate on pure states, but can also do so from other kinds of vertices, it is necessary to make **UML::Vertex** a redefinable element, instead of just **UML::State**.

## Discussion

The issues with the present mechanism identified above can be resolved in a number of ways, but they all require some changes to the UML abstract syntax in the area of state machine redefinition. At the very least, the redefinition of **UML::State** has to be replaced by making **UML::Vertex** a redefinable element. In addition, from the viewpoint of conceptual clarity, it would probably be a good idea to specialize **UML::RedefinableElement** to account for this particular form and, possibly, to introduce a new concept to handle the notion of state machine extension (e.g., as a specialization of **UML::Generalization**). Naturally, these changes should be accompanied by appropriate additions to the text of the spec.

Finally, the current spec mentions (informally) a restriction on transition redefinition, which requires that a redefining transition must retain the same source state as the redefined transition, which, clearly, is not feasible since a transition cannot span elements that belong to two different state machines. This text should be removed.

## References

1. Object Management Group, The, *OMG Unified Modeling Language (OMG UML),* Version 2.5, formal/2015-03-01, 2015.

1. Although UML does not provide an explicit notation for regions (only for region separators), for the purposes of this discussion, we have chosen to represent them explicitly using a dashed box with rounded corners, which encloses the region’s states and vertices. [↑](#footnote-ref-1)
2. This is defined by OCL operation Classifier::inheritableMembers() (section 9.9.4.7 in [1]). [↑](#footnote-ref-2)
3. Although this example focuses on the case of the classifier behavior of a class (which happens to be the most interesting one from a practical viewpoint), it is possible to specialize any state machine that is a non-private member of a class. [↑](#footnote-ref-3)
4. See section 14.3.3.1 (“State Machine Extension”) of [1]. [↑](#footnote-ref-4)
5. In fairness to the authors, this capability was added rather hastily towards the end of the drafting process, as the deadline for submission was looming. As a result, there was no time to properly review this feature. [↑](#footnote-ref-5)
6. Clearly, none of the members of a state machine element should have private visibility, since that could lead to violations of the abstract syntax. [↑](#footnote-ref-6)