How do we treat calls to **new** in the event function? What corresponding code must go in the reverse and commit functions?

The **new** keyword allocates memory; hence at first look its natural inverse is **delete**. However, the **new** keyword actually performs two actions: allocation and construction. While **delete** is the inverse for the allocation functionality of **new**, it is not necessarily an inverse of the construction of a new object. Consider the following example:

struct A

{

static int activeCount;

A() { activeCount++; }

~A()

{

--activeCount;

if (activeCount == 0)

{

//Change the simulation state here

}

}

};

void eventMethod(A\* pa)

{

...

if (...)  
 pa = new A();

...

}

Now, if the reverse event calls **delete pa**, the destructor of A will be called, which may potentially modify the simulation state. Instead of destruction and deallocation, in the reverse function we want *reverse construction* and deallocation.

So, the reverse code may handle a **new** like this:

//First, call the inverse of the constructor

pa->reverse\_constructor();

//Deallocate without calling the destructor

operator delete(pa);

Complications with generating the reverse constructor

Generating a reverse constructor requires reversing all the constructors that were called when the object was constructed. When constructing an object, all the constructors of its ancestors and members are called. This process is recursive, calling constructors for members of members, etc. Let’s call this the construction tree.

Unlike destructors, constructors can be overloaded in C++. Hence, in every point of the construction tree, there may be multiple valid constructors that may be called. Luckily, we can statically determine the tree of destructors that will be called.