What must a commit function accomplish? In what order does a commit function have to execute? Do we have to call commit functions for every function called from the main event method? Why or why not? If we do have to call commit functions the same number the forward function was called, how do we accomplish this?

A commit function must accomplish three fundamental goals:

* Pop off all saved state from the state saving stack
* Deallocate memory whose deallocation was not performed in the forward function (e.g. when **delete** appears in the event method)
* Commit delayed I/O actions

There are also two important requirements for the commit method

* **The commit method must not rely on the current program state in any way.** This is because the commit method is called potentially long after the forward method has been called; many more methods could have executed and modified the simulation state. With the reverse method, we have a guarantee that when the reverse method is called, the program state is exactly the same as when the forward method finished executing. With the commit method we have no such guarantee.
* **The commit method must not modify the current program state in any way.** Unlike forward and reverse methods, the commit method must not modify the visible program state in any way. For example, memory is deallocated, but only memory that the rest of the program has already discarded and will not reference again.

Popping off the state saving stack

In the forward event, variables are pushed on the rollback stack in a certain order. In the reverse event, these variables are popped off in the reverse order. However, in the commit event, the variables must be popped from the bottom of the stack, in the original order they were pushed.

Forward: push(a), push(b), push(c)

Reverse: pop\_back(c), pop\_back(b), pop\_back(a)

Commit: pop\_front(a), pop\_front(b), pop\_front(c)

Unfortunately, the actual variables pushed on the stack during forward execution are not known statically. With branches and loops, the number (and types) of variables pushed on the stack may vary between executions. The reverse method knows what variables were pushed, since its execution retraces the steps of the forward method in reverse order. How do we generate the commit method in such a way that it correctly pops off all the data? The commit method has no guarantees about the current program state, so it may not execute similarly to the reverse method, even if the order of the stack is not a problem.

Solution 1 (broken)

Save the total number of variabes pushed onto the stack. Then, in the commit function, pop that number of variables

Problem with solution 1

Our C++ rollback stack is type-safe; we use templates to push different types on the same stack. When popping off items, we must provide the type of the value that was pushed. Hence, just knowing the number of items that must be popped off the stack is not sufficient.

Solution 2

**Assume that only scalars are pushed on the rollback stack.** This assumption would allow us to pop of values in a manner that is not type-safe, because we wouldn’t need to call any destructors.

In this case, we can simply push void\* pointers on the rollback stack. Then, in the commit function, we call **operator delete** on each pointer to dellocate the memory. If we also have the total number of items pushed on the rollback stack during forward execution, we can easily pop them off.

Problem with solution 2

Using dynamic memory allocation for each little heap value is extremely, extremely expensive. We also have the restriction that only scalars are pushed on the stack.

Solution 3

We start with the same assumption as solution 2: only scalars are pushed on the rollback stack. We specialize the rollback stacks for the four different scalar sizes: we have a 1-byte stack, 2-byte stack, 4-byte stack, and an 8-byte stack.

In this case, we avoid dynamic memory allocation. However, we have to store the number of elements pushed onto each stack. Thus, we would have to store 4 integers giving us the stack heights, rather than just one.

Implementation issue 1: How do we implement this in a cross-platform manner? Sizes of primitive types are system-dependent. Right now I have a proof-of-concept implementation of this that hardcodes the size of all the native types. However, it asserts if the size of the native type doesn’t match the expected size, so when we run on different systems we’ll detect the difference in size.

Ok, so we’ve proposed some solutions to deal with popping off the state saving stack. Now we should decide how to handle implementing the other two roles of a commit function – delaying deallocation and delaying output.

Implementing Delayed Deallocation

In the commit method, we have to call **operator delete** on a number of pointers. If we have four stacks for primitive values, we can add a fifth stack for pointers that must be deallocated. In the commit function, we simply traverse this stack and deallocate every pointer. Also, in the forward function we must save the total number of pointers pushed on the deallocation stack, so that the commit function knows how many pointers to deallocate.

Implementing Delayed Output

Commands such as **printf**, **fprintf**, **close**, and C++’s **cout**, **cerr** need to be delayed to the commit method. Due to the nature of I/O API, we cannot detect all output that can occur from within a method. We will choose a fixed set of standard I/O library calls and handle those explicitly. Handling **printf** and **fprintf** could be done with replacing those calls with **snprintf** or **asnprintf** to generate the string that would be output. Then, the output string along with the file handle can be used in the commit method to send the output to the appropriate file handle. Similarly, **cout** and **cerr** can be replaced with **std::stringstream**.

Each delayed output call fundamentally requires saving two things – the file handle and the action to be performed on it. All the execution that may contain side effects still occurs during the forward method; we simply delay the output action itself.

Note that output, unlike popping off the stack or delaying memory allocation, must be committed *in the same order as the original execution*. This is not particularly difficult, because once we know the number of I/O items that must be commited, we can read them off the stack and commit them in the order they’re encountered. It’s simply worth mentioning that this is the only contraint on order of actions in the commit function.

Library implementation of a commit function

If we choose to implement a commit function the way described above, most of the implementation of the function can go in a Backstroke library call. By the time the forward function finishes executing, the rollback stacks, the deallocation stack, and the I/O stack have all been populated. The commit function must simply traverse these stacks and perform the required actions; this process is not specific to the function being reversed. Hence, the commit function may simply call “\_\_backstroke\_commit” or a similar library function.

It’s worth noting that there may be ways to specialize the generation of the commit function in a way that makes the commit function more efficient than the generic one outlined here. A simple case of this is a perfectly reversible forward function that pushes nothing on any stack – such a function can have an empty commit method. For such cases, it would still be beneficial to leave the Backstroke-generated commit method that is specific for a single event, even if in most cases the generated commit method is a single function call.

Prototype

I’ve prototyped a commit method implementation implemented as described above, excluding the I/O delaying. Here’s a code sample:

//Called at the very beginning of the forward event. Backstroke will automatically insert this call

void \_\_initialize\_forward\_event()

{

//Make sure \_\_initialize is not being called twice in a row

assert(stack8\_marker == -1 && stack16\_marker == -1 &&

stack32\_marker == -1 && stack64\_marker == -1 && deallocation\_stack\_marker == -1);

//Save all the stack sizes

stack8\_marker = rollback\_stack\_8.size();

stack16\_marker = rollback\_stack\_16.size();

stack32\_marker = rollback\_stack\_32.size();

stack64\_marker = rollback\_stack\_64.size();

deallocation\_stack\_marker = deallocation\_stack.size();

}

//Called after the forward event has finished. Backstroke will automatically insert this call

void \_\_finalize\_forward\_event()

{

//Make sure \_\_initialize was called first

assert(stack8\_marker != -1 && stack16\_marker != -1 && stack32\_marker != -1 &&

stack64\_marker != -1 && deallocation\_stack\_marker != -1);

//Add a processing record the number of events pushed on each stack

EventProcessingRecord record;

record.stack8\_pushes = rollback\_stack\_8.size() - stack8\_marker;

record.stack16\_pushes = rollback\_stack\_16.size() - stack16\_marker;

record.stack32\_pushes = rollback\_stack\_32.size() - stack32\_marker;

record.stack64\_pushes = rollback\_stack\_64.size() - stack64\_marker;

record.deallocation\_stack\_pushes = deallocation\_stack.size() - deallocation\_stack\_marker;

//Make sure the stacks only grow during forward execution

assert(record.stack8\_pushes >= 0 && record.stack16\_pushes >= 0 &&

record.stack32\_pushes >=0 && record.stack64\_pushes >= 0 &&

record.deallocation\_stack\_pushes >= 0);

event\_processing\_stack.push\_back(record);

stack8\_marker = -1;

stack16\_marker = -1;

stack32\_marker = -1;

stack64\_marker = -1;

deallocation\_stack\_marker = -1;

}

void \_\_commit()

{

assert(!event\_processing\_stack.empty());

const EventProcessingRecord& r = event\_processing\_stack.front();

//Pop the rollback stacks

rollback\_stack\_8.erase(rollback\_stack\_8.begin(), rollback\_stack\_8.begin() + r.stack8\_pushes);

rollback\_stack\_16.erase(rollback\_stack\_16.begin(), rollback\_stack\_16.begin() + r.stack16\_pushes);

rollback\_stack\_32.erase(rollback\_stack\_32.begin(), rollback\_stack\_32.begin() + r.stack32\_pushes);

rollback\_stack\_64.erase(rollback\_stack\_64.begin(), rollback\_stack\_64.begin() + r.stack64\_pushes);

//Deallocate memory

for (int i = 0; i < r.deallocation\_stack\_pushes; ++i)

{

operator delete deallocation\_stack.front();

deallocation\_stack.pop\_front();

}

//Remove the event processing record

event\_processing\_stack.pop\_front();

}