# SLX's User-Callable Checkpoint/Restore (Rollback) Feature

**Background**

From its earliest days, SLX's debugger included a checkpoint/restore feature. The debugger provides menu-driven interfaces for saving the current state of a model, i.e., issuing checkpoints, and for subsequently restoring a model to a previously saved state. A model's *complete* state is saved. For example, for each file that is open when a checkpoint is issued, the file's read/write pointer is saved. When a restore is performed, any files that were open when the corresponding checkpoint was issued are restored to their proper state. Files that are open when the restore is issued, but were not open when the corresponding checkpoint was issued are closed.

Checkpoint/restore is extremely useful for finding obscure bugs. One can search for such bugs using a divide-and-conquer approach. For example, suppose you know that a deadlock condition exists at simulated time 1000 in a model, but you don't know how and when the deadlock first arose. You can rerun the model to time 500 and look for the deadlock. If the deadlock does not yet exist, you can issue a checkpoint and run to time 750. If the deadlock exists at time 750, you can restore the model's state to time 500 and proceed more carefully. The important points are (1) you don't have to rerun the entire model, and (2) you can narrow down the search interval by roughly one half with each iteration.

In response to requests from several SLX users, in early 2003 we investigated adding a user-callable interface to SLX's checkpoint/restore. We anticipated three potential uses. First, it could be used to capture a model's state after warmup and use this state as a starting point for conducting experiments. For example, a factory model that starts empty and idle has to undergo a substantial warmup to reach normal operating conditions. Second, several users had described modeling situations in which at a given point in simulated time, they needed to "simulate ahead" one or more times to select system algorithms or algorithm parameters, then restore back to the original point in time and resume "normal" simulation, using the selected algorithms/parameters. The third use we anticipated was rollback for distributed simulation.

It was pretty clear at the outset that the limited demands imposed by the first two types of applications could be met easily. As of this writing, it still remains to be demonstrated that SLX's user-callable, general-purpose checkpoint/restore can meet the much heavier demands imposed by frequent rollback in a distributed simulation. Every effort was made to maximize performance. For example, when possible, checkpoints are recorded in virtual memory, rather than being written to files. In addition, SLX's underlying memory management routines were revised to decrease the overhead required to traverse lists of dynamic memory in use at the time of issuance of a checkpoint.

**The User-Callable Checkpoint/Restore Interface**

SLX provides three built-in functions for managing checkpoint/restore:

| | |
|---|---|
| SLXCheckpoint | - issues checkpoints |
| SLXRestore | - restores model state to a saved checkpoint |
| SLXReleaseCheckpoint | - releases memory consumed by a checkpoint |

**SLXCheckpoint**

SLXCheckpoint is called as follows:

```
Class exemption             // a collection of data exempt from checkpoint/restore
    {
    …                           // checkpoint-exempt data
    };

exemption   Exemptions; // an instance of the exemption class
int         idno;

idno = SLXCheckpoint(&Exemptions);
```

SLXCheckpoint returns an integer checkpoint identifier that can be used in subsequent SLXRestore or SLXReleaseCheckpoint calls.

The single, mandatory, non-NULL argument required by SLXCheckpoint is a pointer to an object that is exempt from checkpoint/restore. The need for exemptions is readily apparent. If there were no exemptions, the first time a model issued an SLXRestore call, the model would go into an infinite loop, restoring model state to exactly that previously stored in a checkpoint and then once again running to the SLXRestore call. Logic based on exempt variables is needed to break such loops.

SLX prohibits the use of sets, pointers, variable-dimensioned arrays, and nested objects within the exempt data class. Without these restrictions, implementation of user-callable checkpoint/ restore would be much more difficult, if not impossible. Consider, for example, the use of pointers. If pointers were allowed to be exempt from checkpoint/restore, handling use counts of any objects to which they pointed would become impossibly difficult. When such objects were restored, their use counts would be rolled back to their checkpointed state, ignoring uses that resulted from assignments to exempt pointers made between the time the checkpoint and restore were issued.

**SLXRestore**

SLXRestore is called as follows:

```
SLXRestore(idno);
```

"idno" is an integer ID returned by a prior call to SLXCheckpoint.

**SLXReleaseCheckpoint**

In many applications, multiple, if not many, checkpoints are required. Since a checkpoint saves the entire state of a model, memory requirements can accumulate at a prodigious rate. The only circumstance under which SLX automatically discards a checkpoint occurs when an SLXRestore call restores an earlier checkpoint. Checkpoints later that the one restored *must* be discarded, because in the general case, after restoration, interactive model input can take the model down a path that differs from the original path taken. Accordingly, all checkpoints other than those invalidated by restoration of an earlier checkpoint are retained until they are explicitly released by calling SLXReleaseCheckpoint, which is called as follows:

```
SLXReleaseCheckpoint(idno);
```

"idno" is an integer ID returned by a prior call to SLXCheckpoint.

**A Simple Example**

**Checkpnt.slx**

Checkpnt.slx is a modified version of a simple queueing model (Tom Schriber's legendary barbershop model). Its main program forks off a puck that issues SLXCheckpoint and SLXRestore calls. The model calls SLXCheckpoint at time 100 and 200. A class named ExemptData is used to store the ID of the current checkpoint and a count of the number of times code "downstream" from SLXCheckpoint calls is executed. At time 300, if the count exceeds 10, the model exits. If the count is less than 5, an SLXRestore call is issued, restoring the model's state to checkpoint 1, issued at time 100. If the count is greater than 5, but less than 10, the model's state is restored to that saved in the checkpoint issued at time 200. Since the checkpoint issued at time 100 is the first checkpoint issued, we can always refer to it as checkpoint 1. Multiple checkpoints are issued at time 200. Let's briefly summarize the checkpoints issued in this model.

| Checkpoint ID | Time Issued | Notes |
|---|---|---|
| 1 | 100 | The first two restores restore the state saved in checkpoint 1. |
| 2 | 200 | The 1st restore (to time 100) causes this checkpoint to be released automatically. |
| 3 | 200 | The 2nd restore (to time 100) causes this checkpoint to be released automatically. |
| 4 | 200 | When the 3rd restore is issued, XD.count = 6, because it's been incremented 3*2 times. Thus, the second of the two restores is issued, using the current checkpoint ID (4).<br><br>The 4th through 7th restores also restore the state of checkpoint 4, as XD.count goes from 7 to 10. When XD.count reaches 11, the model terminates. |

**A More Subtle Example**

In this section, we'll examine four variations of a model developed to vigorously exercise checkpoint/restore. Each model has a main program that creates a new active object at time 1…1000. Each object records a checkpoint, advances the clock by 1, and issues a restore. The object repeats the time advance, restore sequence 100 times before terminating. In all, 1,000 checkpoints and 100,000 restores are issued.

**Slowdown.slx**

The first model, slowdown.slx, as its name suggests, exhibits some performance problems. It runs slower than one would hope, and it consumes a lot of virtual memory (24MB) in a hurry. Its appetite for memory is explained by the fact that checkpoints are never released. Therefore, the model quickly accumulates 1000 copies of the model's state. Performance suffers for the following reason. The checkpoint is issued in-line as an initial value assignment to the variable named check, local to the actions property of the class named process. SLX1, under which this program was developed, did not handle local variable initialization in the same way C/C++ does. In SLX1, all initialization of local variables was "promoted" to the "top" of the class or procedure that contains the variables. Thus "check" is initialized as part of the creation of a "process" object. Therefore, the checkpoints in slowdown.slx are issued between the activate and new verbs in "activate new process". When a restore is performed, all the actions associated with the "activate" verb must be repeated. This is not at all obvious in looking at the code. You might wonder why SLX1 was so dumb about handling local variable initialization. The answer lies in the fact that SLX imposes far more stringent rules than C/C++ does. In SLX, every variable has a known initial value. In C/C++, variables not explicitly initialized have undefined initial values. SLX must also deal with potential suspension and resumption of pucks. In short, because SLX has a much harder job to do, we took an implementation shortcut. SLX3 has overcome these limitations. It allows C/C++-compatible initialization of variables.

**Faster.slx**

Faster.slx improves the performance of slowdown.slx by separating the declaration of "check" and the issuance of the checkpoint. This eliminates quite a bit of scheduling overhead, but does nothing to reduce the rapid accumulation of virtual memory.

**Fastest.slx**

Fastest.slx removes additional overhead by moving the instantiation of the x (the "exempt" object) from within class process to the module level. The end result is that x is allocated exactly once, rather than 1000 times. In theory, this ought to further reduce execution time, but in our experiments, this improvement only became visible when the total number of objects created was substantially increased. To summarize, the improvement (actually slightly negative) was less than expected.

**Smallest.slx**

Smallest.slx drastically reduces memory requirements by releasing checkpoints that will never be used again. A great deal of memory is saved at the expense of a bit of CPU time. (It takes time to release the checkpoints.)

**Summary**

The user-callable interface for SLX's checkpoint/restore feature opens up some exciting new opportunities. As of this writing, it remains to be seen exactly what our users will be able to accomplish with this feature. As the examples we've presented illustrate, user-callable checkpoint/restore must be used with great care. Carelessly constructed models can consume gargantuan amounts of memory and CPU time, and of course, improperly constructed termination conditions can easily lead to infinite checkpoint…restore loops.

To quote the GPSS/H Reference Manual, "It's usually the case in life that great power is accompanied by great responsibility." Have fun, but be careful.