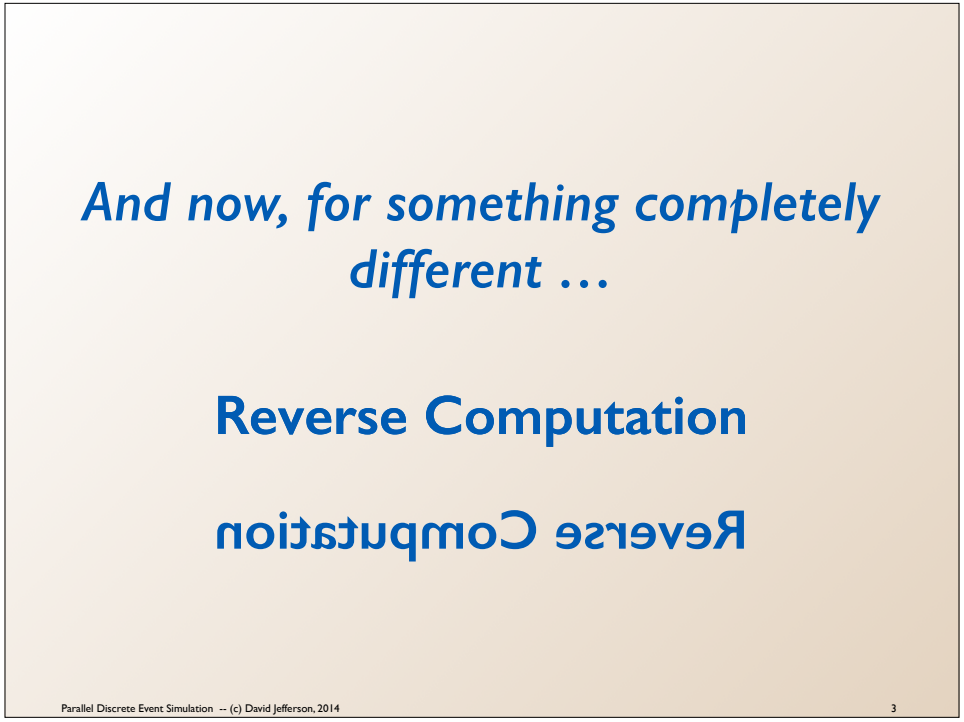
Parallel Discrete Event Simulation Course #12

David Jefferson
Lawrence Livermore National Laboratory
2014

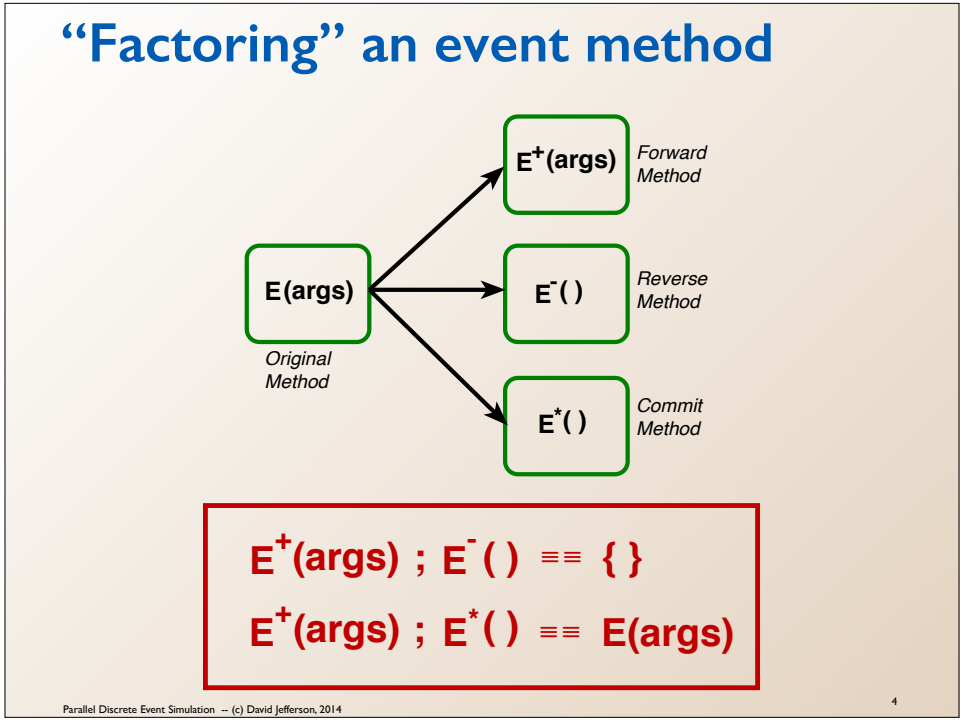
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Release Number: LLNL-PRES-653678

Reprise



It's different because we will be talking about sequential computation, programming languages, source-to source transformations, etc., without much talk about synchronization.



The fundamental idea is to take all of the event methods **`E(args)`** in a parallel discrete event simulation and "factor" each of them into three parts: **`E+(args)`**, **`E-()`**, and **`E*()`**.

`E+(args)` is executed in place of E(args) in the simulation and is instrumented to save all information destroyed by the forward execution of **`E(args)`** so as to preserve the option after **`E(args)`** completes of restoring the initial state of an object before it executed.

`E-()` uses the information stored by **`E+()`** and also the object's state information to exactly reconstruct the state before **`E+()`** executed. It in effect reverses all of the side effect of **`E+()`** and exactly accomplishes rollback of the event.

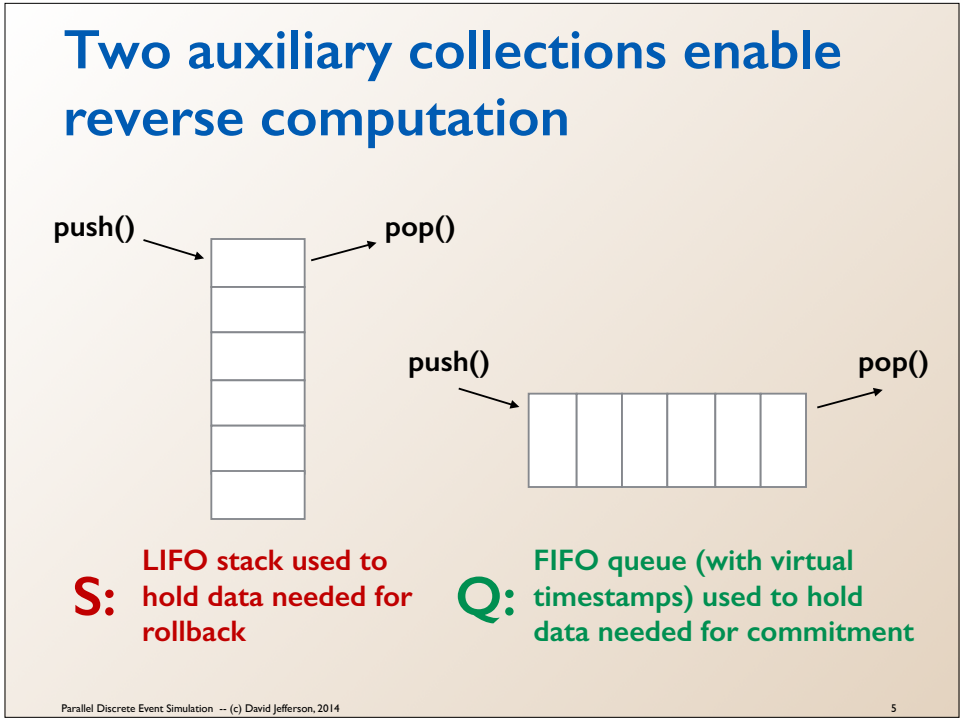
`E*()` is executed at the time event **`E(args)`** is committed, and deals with actions specified in **`E+(args)`** that really cannot be rolled back, such as output, or the freeing of dynamically allocated storage.

The two equations boxed in red are properties that the three methods must satisfy.

The first one says the **`E-()`** really does reverse all of the side effects of **`E+(args)`**, *and does nothing else*, so that executing **`E+(args)`** followed by **`E-()`** is a no-op.

The second one says that executing **`E+(args)`** followed by **`E*()`** is equivalent to executing the original method E(args).

Since every time **`E+(args)`** is executed it will either be rolled back or committed, then either **`E-()`** or **`E*()`** will be executed after it and the net effect will either be a no-op (in the case of a rollback) or it will be as if **`E(args)`** executed (in the case of commitment).



In factoring an event method we generally use two auxiliary data structures. One, which we call S in these slides, is a LIFO stack that is used to hold data needed for rollback. The LIFO structure is natural, because the data that we need for executing $E^-()$ is need in the reverse order of the order in which it was put in during forward execution of $E^+()$.

The other we are calling Q, and it is a FIFO queue of data to be saved for $E^*()$ at commitment time. Since the things done at commitment time (e.g. output) have to be done in the same order as specified during execution of $E^+()$, it is natural for a FIFO queue to be employed.

The "**push**", "**pop**" and later "**top**" and "**front**" terminology are from the C++ STL.

Minimize saved state

- Goal: ***minimize the amount of data saved and restored to conserve time and space***
- More generally, minimize all overheads introduced in `E+(args)` assuming that `E+()` is called many more times than `E-()`
- "Perfect reversibility": a segment of code is ***perfectly reversible*** of it can reversed with ***no saved state at all.***

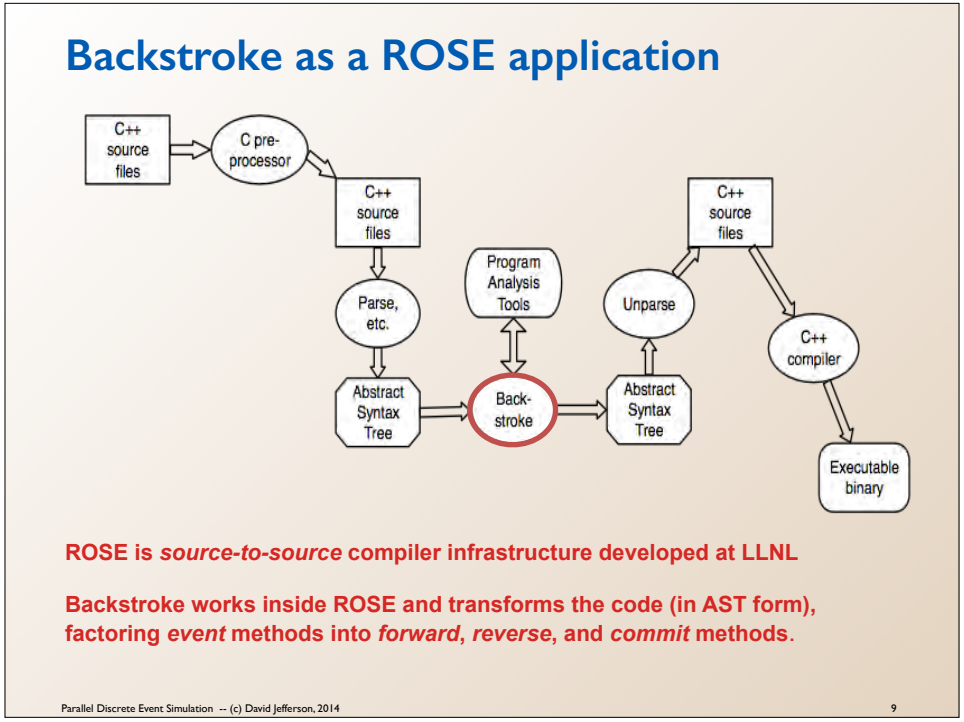
Perfect or near perfect reversibility is very useful when achievable. When data has to be stored on the Stack it involves storage allocation, data copying in memory, and calls to both **`push()`** and **`pop()`** for each data item stored — which is quite a bit of overhead. Thus we *really* want to minimize the data that has to be stored to enable rollback, even at the expense of a large investment in program analysis at compile time.

Need for *automated* generation of forward, reverse and commit methods

- Requiring programmers to write `E+(args)`, `E-()`, and `E*()` in addition to `E(args)` is a prohibitive software engineering burden.
  - It essentially triples the work
  - It is extremely taxing mentally
  - It is extremely difficult to debug and maintain.
  - Turns ordinary bugs into Heisenbugs
- For reverse computation to be feasible it is essential that the programmer have to write no more code than `E(args)`
- As a practical matter `E+(args)`, `E-()`, and `E*()` must be *automatically generated* !

LLNL has a project called Backstroke that is intended to automatically generate reverse code for almost any code written in C++.

Backstroke



This is a diagram of the workflow in ROSE, a general source-to-source program transformation system and compiler. Backstroke is the component in **red** that uses ROSE's powerful program analysis tools and adds forward, reverse and commit routines for all event methods in a ROSS-compatible simulation.

It is not absolutely essential to construct a reverse code generator this way. You could go directly from the revised Abstract Syntax Tree to executable binary. (The LORAIN project at RPI, based on LLVM, is structured that way.) But by going back through source code, the programmer can see what the forward, reverse, and commit methods look like, and perhaps learn how to improve their performance.

Backstroke applies to most of C++

- **We intend to support *almost the entire C++ language*, including**
  - assignment, initialization
  - sequential control constructs `;`, `if`, `switch`, `for`, `while`, `return`, `continue`, `break`, `goto`
  - scalars, structs, arrays, classes and class types
  - methods, functions, inheritance, virtual functions, recursion
  - casts
  - constructors, copy constructors, destructors
  - many STL container classes
  - dynamic storage allocation, deallocation
  - templates
- **With restricted support of**
  - arbitrary pointer structures
- **But excluding**
  - exceptions, throws
  - function pointers
  - threads

Some parts of the C++ language are straightforward to support. Some are quite tricky. Some may benefit greatly from programmer advice. and some are so difficult to support that it will never be worth the effort to try. If generating reverse code for legacy code we may need hand work to rewrite those parts that use features of the language that are not conducive to reversibility. For new code we need to advise programmers to avoid certain language constructs that will preclude automatic generation of reverse code.

Generating reverse code

Some basic program fragments and reversing templates

	E()	E⁺()	E⁻()
Additive integer assignment	`i = i + m;`	`i = i + m;`	`i = i - m;`
Floating point	`x = x + y;`	`S.push(x);` `x = x + y;`	`x = S.top();` `S.pop();`

General assignment	`x = f(x,y);`	`S.push(x);`
		`x = f⁺(x,y);`
			`f⁻();`
			`x = S.top();`
			`S.pop();`

Sequential composition	`P ; R ;`	`P⁺ ; R⁺ ;`	`R⁻ ; P⁻ ;`

Conditional w/ side-effect free test	`if (test) { P; }`
	`else { R; }`
		`if (test)`
		`{ P⁺; S.push(1); }`
		`else`
		`{ R⁺; S.push(0); }`
			`if (S.top())`
			`{ S.pop(); P⁻; }`
			`else`
			`{ S.pop(); R⁻; }`

WHILE-loop with side-effect free test	`while (test) {`
	`P;`
	`}`
		`i = 0;`
		`while (test) {`
		`P⁺;`
		`i=i+1;`
		`}`
		`S.push(i);`
			`i = S.top();`
			`S.pop();`
			`while ( i>0 ) {`
			`i=i-1;`
			`P⁻;`
			`}`

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 12

Note: the code in the General Assignment row and the **`E⁻()`** column is incorrect. Fixed on a later slide!

S is the global stack onto which all saved data is pushed that is required for rolling back forward execution.

For the tests in both the conditional and the while loop, we assume there are no side effects. If there are, then the reverse code can to be easily adjusted. Note that integer increments / decrements and sequential composition require no data to be stored on the stack S.

Throughout these slides we use **`S.pop()`** as defined for Stacks in the C++ STL, so that **`S.pop()`** does *not* return the value on the top of the stack, but just deletes it.

Some basic program fragments and reversing templates

	E()	E^+()	E^-()
Additive integer assignment	`i = i + m;`	`i = i + m;`	`i = i - m;`
Floating point	`x = x + y;`	`S.push(x);` `x = x + y;`	`x = S.top();` `S.pop();`

General assignment	`x = f(x,y);`	`S.push(x);`
		`x = f+(x,y);`
			`f-();`
			`x = S.top();`
			`S.pop();`

Sequential composition	`P ; R ;`	`P+ ; R+ ;`	`R- ; P- ;`

Conditional w/ side-effect free test	`if (test) { P; }`
	`else { R; }`
		`if (test)`
		`{ P+; S.push(1); }`
		`else`
		`{ R+; S.push(0); }`
			`if (S.top())`
			`{ S.pop(); P-; }`
			`else`
			`{ S.pop(); R-; }`

WHILE-loop with side-effect free test	`while (test) {`
	`P;`
	`}`
		`i = 0;`
		`while (test) {`
		`P+;`
		`i=i+1;`
		`}`
		`S.push(i);`
			`i = S.top();`
			`S.pop();`
			`while ( i>0 ) {`
			`i=i-1;`
			`P-;`
			`}`

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 13

`S` is the global stack onto which all saved data is pushed that is required for rolling back forward execution.

For the tests in both the conditional and the **`while`**-loop, we assume there are no side effects. If there are, then the reverse code can to be easily adjusted.

Note that integer addition / subtraction and sequential composition require no data to be stored on the stack **`S`**.

Some basic program fragments and reversing templates

	E()	E^+()	E^-()
Additive integer assignment	`i = i + m;`	`i = i + m;`	`i = i - m;`
Floating point	`x = x + y;`	`S.push(x);` `x = x + y;`	`x = S.top();` `S.pop();`

General assignment	`x = f(x,y);`	`S.push(x);`
		`x = f+(x,y);`
			`x = S.top();`
			`S.pop();`
			`f-();`

Sequential composition	`P ; R ;`	`P+ ; R+ ;`	`R- ; P- ;`

Conditional w/ side-effect free test	`if (test) { P; }`
	`else { R; }`
		`if (test)`
		`{ P+; S.push(1); }`
		`else`
		`{ R+; S.push(0); }`
			`if (S.top())`
			`{ S.pop(); P-; }`
			`else`
			`{ S.pop(); R-; }`

WHILE-loop with side-effect free test	`while (test) {`
	`P;`
	`}`
		`i = 0;`
		`while (test) {`
		`P+;`
		`i=i+1;`
		`}`
		`S.push(i);`
			`i = S.top();`
			`S.pop();`
			`while ( i>0 ) {`
			`i=i-1;`
			`P-;`
			`}`

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 14

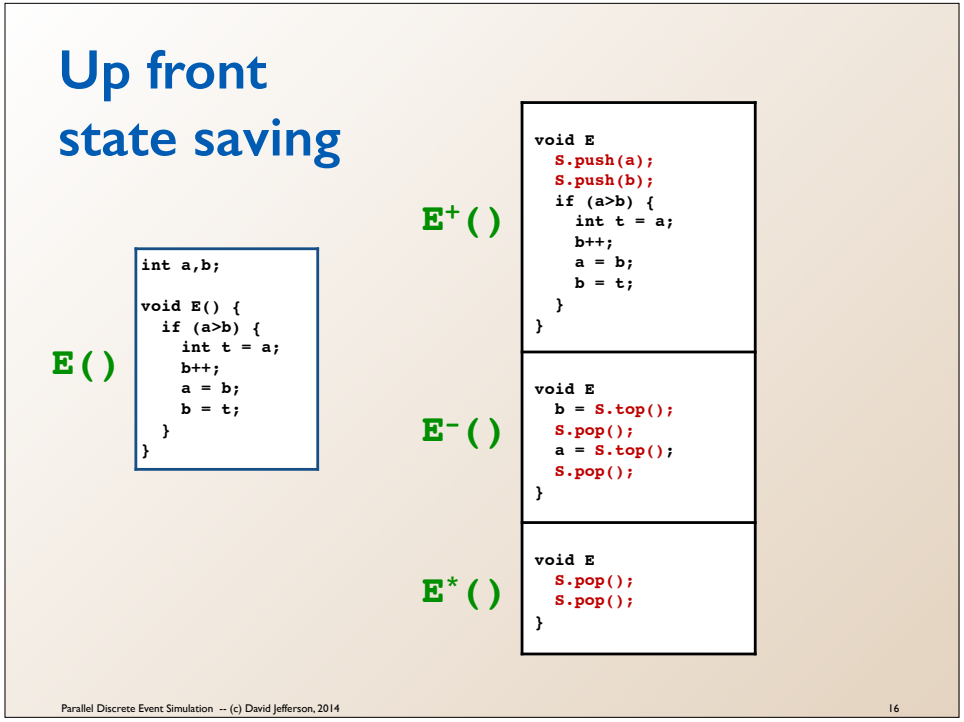
S is the global stack onto which all saved data is pushed that is required for rolling back forward execution.

For the tests in both the conditional and the while loop, we assume there are no side effects. If there are, then the reverse code can to be easily adjusted.

Note that integer increments / decrements and sequential composition require no data to be stored on the stack **S**.

General approaches to reverse code: Up front state saving

- **Forward:**
  - Determine statically what variables might change during execution of the event
  - Push values of all writable state variables onto the stack at beginning of `E+(args)`
- **Reverse:**
  - Pop the stack and restore all state variables in the body of `E-()`
- **Strengths:**
  - restores only initial state, not intermediate states
  - independent of the length of time the event runs
  - stores a variable only once, regardless of how many times it is modified
  - does not require control flow analysis (though it helps)
  - works with some language constructs nearly impossible to handle with other approaches
    - threads
    - exceptions
- **Weaknesses:**
  - Time and space overhead proportional to the ***size of the object state***, even if only a small fraction is changed in one event
  - must save *all data that might be modified* if you can't statically demonstrate it will not be modified, including
    - whole structs, arrays, and collections even if only one element is touched but you don't know which one
    - any data on the heap reachable by pointer chains that *might* be modified



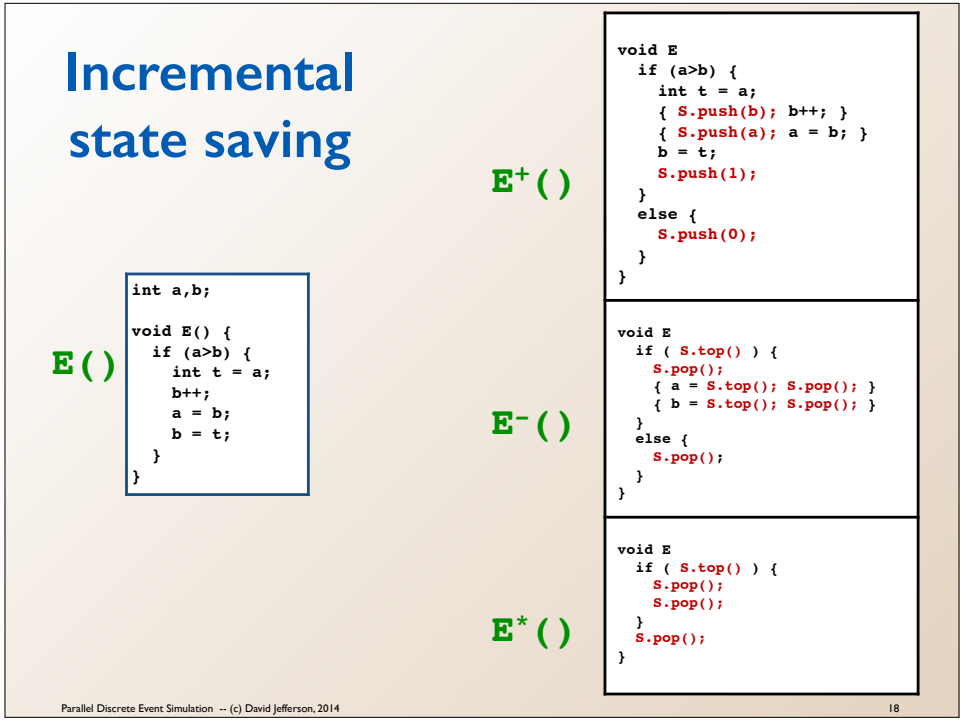
With up front state saving we identify all of the state variables that *might* change and push their values on the stack at the beginning of the forward routine. We do not have to record which branch of the conditional was taken, and if there were a loop in the method body we would not need to record how many times the body was executed. In this case both state variables do change. But if there were more variables in the state and we could not determine statically that some of them would not change during execution of **`E()`** we would just introduce code in **`E`**$^{+}$**`()`** and **`E`**$^{-}$**`()`** to save and restore them all. In **`E`**$^{-}$**`()`** we simply restore the values of **`a`** and **`b`** and pop the stack.. We do not have to pay any attention to the algorithm used in **`E()`**.

The commit method **`E`*****`()`** must also pop any values off the stack that were pushed there by the forward method because the commit method is called if and only if the reverse method **`E`**$^{-}$**`()`** is *not* called.

Note also that variable **`t`** here is only a temp. It is not a state variable in the simulation, and hence it does not have to be restored during rollback.

General approaches to reverse code: Incremental state saving

- **Forward:**
  - Push `<variable,oldvalue>`-pairs onto the entropy stack every time a variable changes during `E+(args)`.
    - If `<variable,*>` already appears in the stack, no need to save a second time
    - … but it may cost to much too check that each time!
- **Reverse**
  - Pop `<variable,oldvalue>`-pairs off of the entropy stack one by one, and restore variable values in reverse order in which they were saved.
    - A variable may be "restored" multiple times if duplicates are not eliminated
- **Strengths:**
  - Works well for objects with large states as long as only a small part of the state is modified in an event, and even if we cannot determine statically know which variables will be modified
  - Works well with arrays and collections when only a small part is modified in an event
  - Works well when variables are modified indirectly through pointers
- **Weaknesses:**
  - Time and space overhead is proportional to the *time* the event method executes
  - Aliasing inhibits the ability to detect that a variable have already been saved unless the change is stored by address
  - Provides the capability of restoring any intermediate state, not just the initial state — more than necessary.

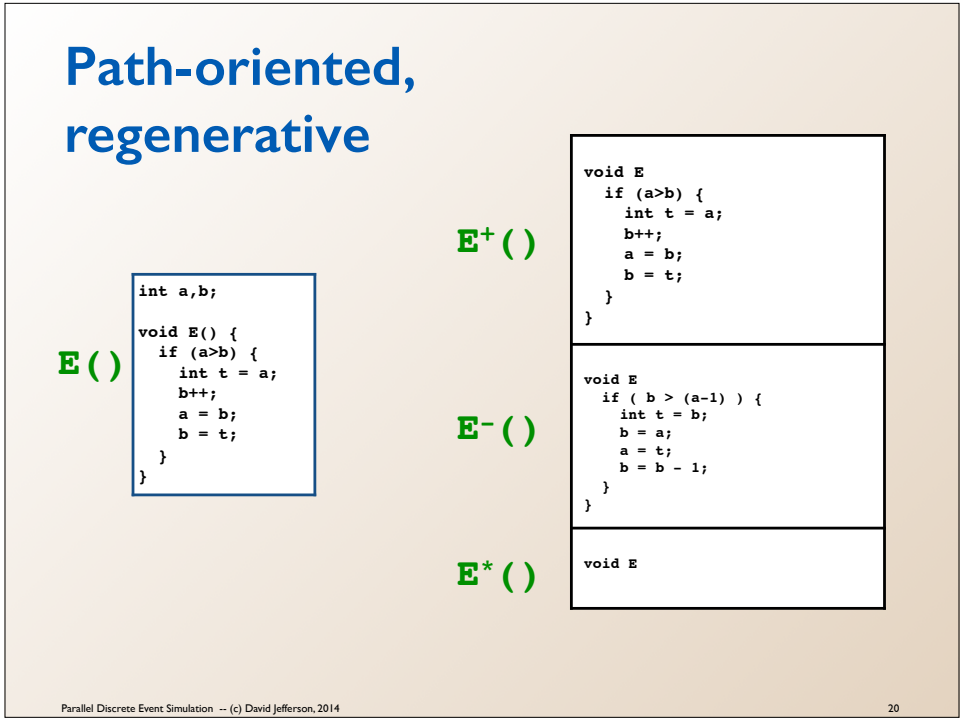


With Incremental State Saving we instrument the forward routine **E+()** to save the value of a state variable the very first time it is overwritten or, if we cannot determine that statically, then we save the value every time it is overwritten that *might possibly be the first time*. In the forward method **E+()** we save its value only the first time if we can because of course in the reverse method we only need to restore it to its *initial* value. In this example the variable b is overwritten twice, but we push its value onto the stack only the first time. Of course we also have to push a boolean indicating which branch of the conditional was taken. The reverse method, **E-()**, only restores the variable to their original values and pops the stack.

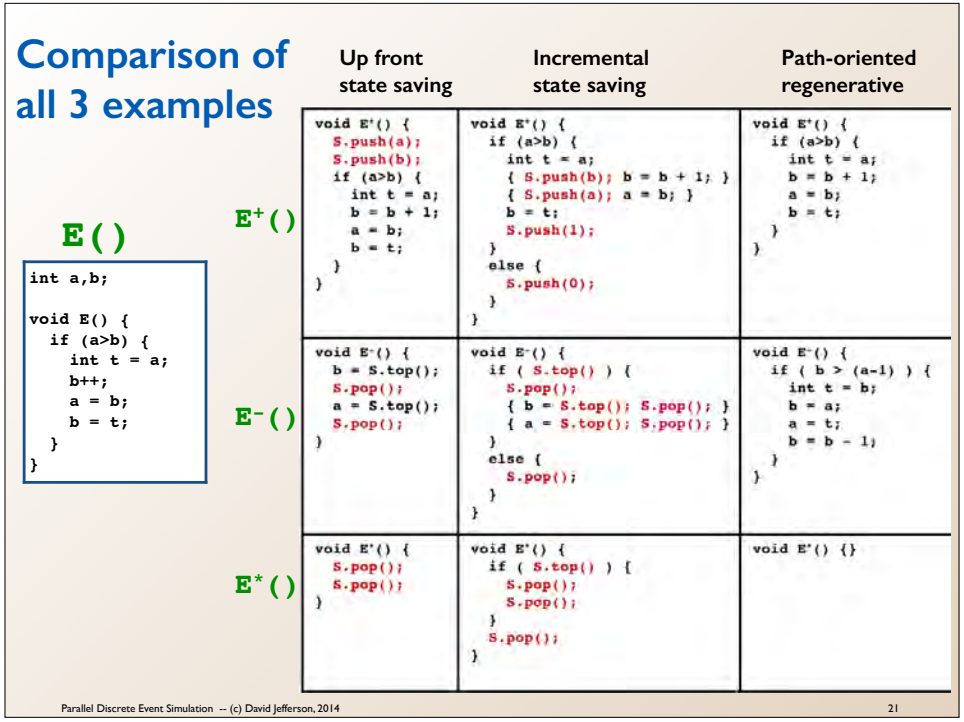
The commit method **E*()** must also pop any values off the stack that were pushed there by the forward method because the commit method is called if and only if the reverse method **E-()** is *not* called.

General approaches to reverse code: Path-oriented, regenerative methods

- **Forward:**
  - Deep control and data flow analysis of the code to determine what parts of the initial state can be reconstructed from calculations based on the final state.
  - Save only initial data that cannot be reconstructed from data in the final state (deciding what to save after reverse code has been generated).
- **Reverse**
  - Each path through the code considered separately.
  - Reconstruct as much initial data along each path as possible from final state data, and insert S.push() calls along each path in the forward method for data that cannot be reconstructed along that path, and corresponding S.pop() calls in the reverse routine.
- **Strengths:**
  - In most cases should allow faster forward and reverse code to be generated, with near minimal data storage.
- **Weaknesses:**
  - Number of paths through code is exponential in the number of branches
  - Special methods required for loops



In this case the path-oriented and regenerative style of reverse code generation manages to produce *perfectly reversible code*, with nothing pushed onto or popped from the stack. Not that the forward `E+()` routine in this case is identical to `E()` and the commit routine `E*()` is a no-op. We did not even have to introduce any additional variables. Perfect reversibility is not usually achievable for a whole event method, but it often is for at least some regions of the code and/or some variables. It is the ideal of reverse computation.



This slide just summarizes the last three examples. The original code for `E()` is in a box on the left, and three different ways of factoring it into $E^{+}()$, $E^{-}()$, and $E^{*}()$ are recorded in the next three columns of the table.

The comparison shows that in this particular example, if the condition `(a>b)` is true, then there is more overhead with incremental state saving then there is with up front state saving, but if `(a>b)` is false then the reverse is true. This is not a general statement, however.

Also in this case the path-oriented regenerative methods generate perfect reverse code that does not need to save any data on the stack at all. In this case the code it produces is clearly superior to either of the other methods.

Path-oriented, regenerative inversion

E()

```
int a, b, c;

void E() {
  if (a > 0)
    { b = a + 10;
      a = 3;
    }
  if (c == 0)
    c = 5;
  else
    c = 7;
}
```

E⁺()

```
void E+() {
  int path = 0;
  if (a > 0) {
    S.push(b);
    b = a + 10;
    a = 3;
  }
  else
    path |= 2;
  if (c == 0)
    c = 5;
  else {
    path |= 1;
    S.push(c);
    c = 7;
  }
  S.push(path);
}
```

E⁻()

```
void E-() {
  int path = S.top();
  S.pop();
  if ( path & 1 ) {
    c = S.top();
    S.pop();
  }
  else
    c = 0;
  if ( (path & 2 ) == 0) {
    a = b – 10;
    b = S.top();
    S.pop();
  }
}
```

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 22

This is another example application of Backstroke's path-oriented regenerative inversion algorithm. The variables **a**, **b**, and **c** are state variables. The forward code is instrumented in **red** to keep track of the dynamic path taken, and the reverse code uses path information to restore variable values. Note that the variable **path** that is introduced in the forward and reverse methods should be viewed as a bit mask that records for each conditional which branch on the conditional was taken. In the forward routine the low order bit of **path** records is **0** if the **then**-branch of the second conditional is taken, and is a **1** if it is not. The second bit records the same thing for the first conditional. In the reverse method the corresponding conditionals are reversed in order, so the low order bit indicates which branch to take in the first conditional, and the second bit indicates which branch to take with the second conditional.

End Reprise

Function calls

Reverse functions:

	E()	E⁺()	E⁻()

General assignment	`void E() {`
	`  P;`
	`  x = f(x,y);`
	`  R;`
	`}`
		`void E⁺() {`
		`  P⁺;`
		`  S.push(x);`
		`  x = f⁺(x,y);`
		`  R⁺;`
		`}`
			`void E⁻() {`
			`  R⁻;`
			`  x = S.top();`
			`  S.pop();`
			`  f⁻();`
			`  P⁻;`
			`}`

- **We can handle function in some cases by inlining, but that is often not practical.**
- **A function call splits the body of `E` into two *regions*: before `f()` and after `f()`.**
- **It requires us to be able to restore an intermediate state, right where the function was called, not just the initial state.**
  - In this case that is the state just after the execution of **P⁺** in **E⁺()**
  - This is a contrast between region-based and incremental inversion methods.
  - ***Up front state saving*** now must be interpreted as up front of the *region*, not just up front of the entire event method
  - ***Incremental state saving***, however, makes this easy.

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 25

S is the global stack onto which all saved data is pushed that is required for rolling back forward execution.

For the tests in both the conditional and the while loop, we assume there are no side effects. If there are, then the reverse code can to be easily adjusted.

Note that integer addition / subtraction and sequential composition require no data to be stored on the stack **S**.

Saving and restoring class and struct types

Saving and restoring class type objects

- In saving and restoring class-type data, copies have to be made and destroyed. C++ has a number of constructs for this, used in the code on the right:
  - Line 1 invokes the (default) *constructor* for type `CT`
  - Line 2 invokes the *copy constructor* for type `CT`
  - Line 3 invokes the `operator =` function for type CT
  - Line 4 invokes the *destructor* for type `CT`
- Their implementations must all work together when used in saving and restoring class type variables.
  - They must use *full deep copies* and *restores* with *no other side effects*
  - If pointer types are involved then the copies have to be fully *cycle- and aliasing-aware.*
- Backstroke or other automatic reverse code generator must either
  - trust that the implementations of these functions have these properties, or
  - accept a programmer declaration (via pragma) that they do, or
  - *prove* that that do, or
  - auto-generate appropriate versions of these four functions for every class type that has to be saved and restored.

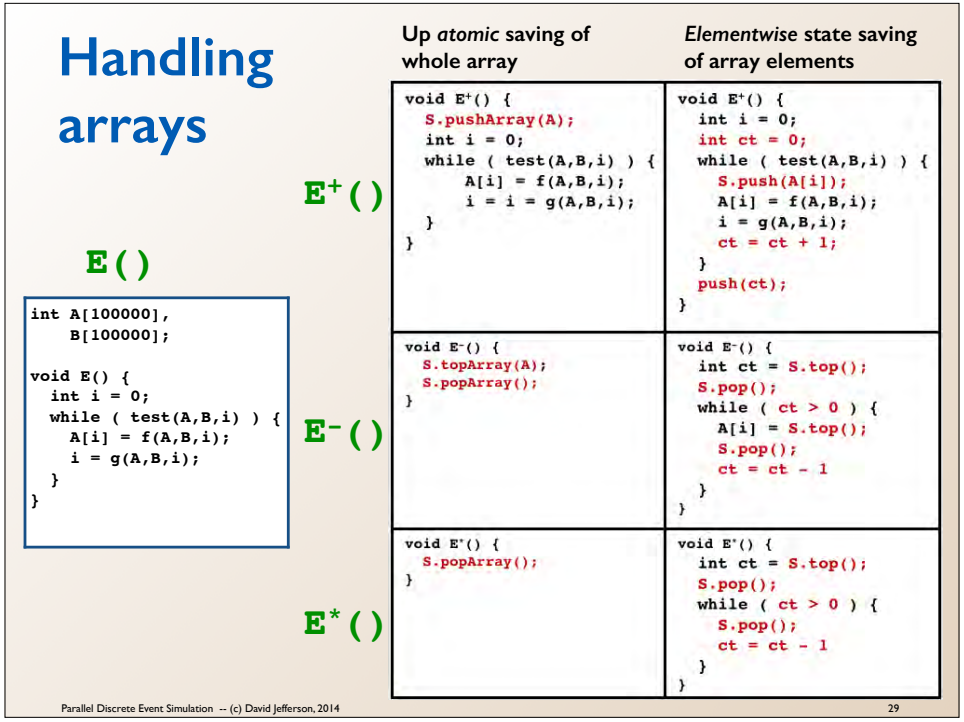
```
CT c;                   // 1

S.push(c);              // 2
c = S.top();            // 3
S.pop();                // 4
```

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 27

The semantics of C++ constructors, copy constructors, destructors, and assignment operators all play a fundamental role in the way reverse computation is implemented when class-type values are involved. We cannot assume that the destructor is the reverse of the constructor, and we cannot assume that they all play well together and have the properties needed in the general case for reverse computation, namely that full, deep copies are made that are aliasing- and cycle-aware and that they have no other side effects except those required for perfect copies.

In C++ it is also useful in some patterns to create uncopyable or unassignable types, and those simply should not be used as the values of state variables in a simulation.

Arrays in loops

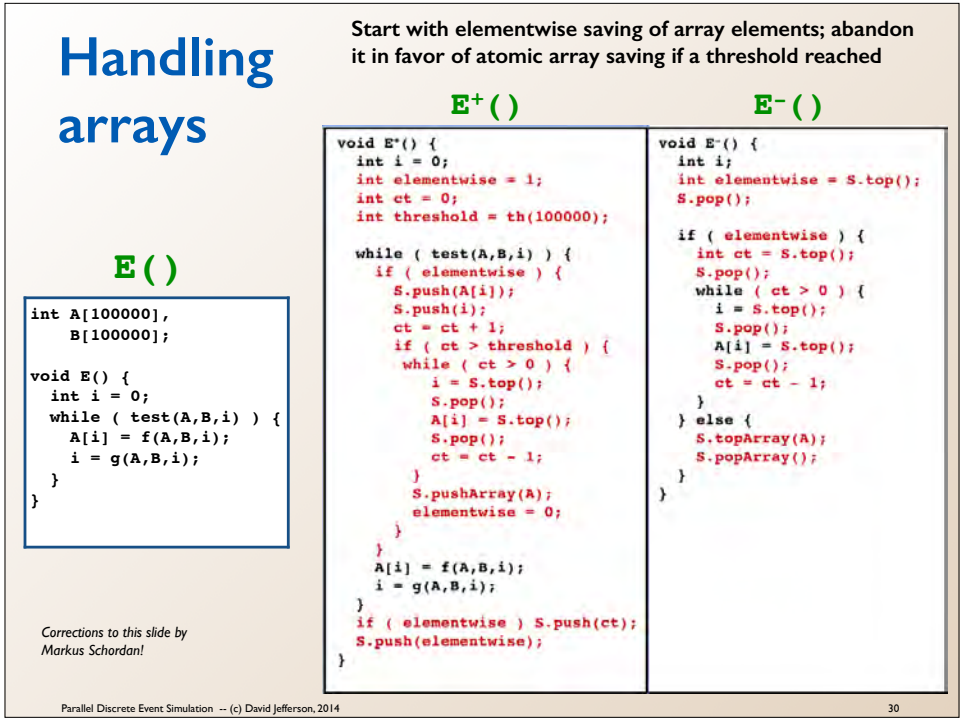


In this example the **`A`** and **`B`** arrays are state variables, but **`i`** is not. And the functions **`test`**, **`f`**, and **`g`** are side-effect free.

For the purposes of exposition on this slide we have assumed the existence of methods **`S.pushArray(A)`**, **`topArray(A)`**, and **`S.popArray()`** that do for array arguments the same things as **`S.push(n)`**, **`S.top()`**, and **`S.pop()`** do for integer arguments.

This exemplifies the tradeoffs that come with handling arrays and other collections. If the **`while`**-loop is executed only a few times, then it is much faster to do elementwise saving of the few elements of the array that are overwritten and restore them one at a time in case of rollback than to save and restore the entire 100,000-element array. But if the loop is executed many times, and many elements of the array are overwritten, then it is faster to simply save the entire array as an atomic data structure and restore the same way on rollback.

But we may not be able to determine statically which of those is the case, which leaves the reverse code generator in a quandary. Which kind of reverse code should it generate? One approach is to allow the programmer to offer advice in the form of a pragma or specially formatted comment indicating which choice to use. Another approach is illustrated on the next slide.



Multiple corrections on this slide thanks to **Markus Schordan.**

In this example the **A** and **B** arrays are state variables, but **i** is not. The functions **th**, **test**, **f**, and **g** are all side-effect free. The function **S.pushArray(A)** is intended to push the entire array **A** onto the stack, even though this is not strictly correct C++; likewise **S.topArray(A)** is intended to copy the entire array value from the top of the stack into **A**. **S.popArray()** just pops off the entire array.

Here we do something sophisticated. We do not decide decide statically whether to use elementwise or atomic saving of array **A**. *Instead, we make some measurements at runtime and decide then.* Whether this method will prove practical or not (i.e. whether we can automatically generate code like this) is an open research question, but this example illustrates the depth and complexity of the options we have in generating good forward and reverse code.

In the forward method **E+()** we don't know how many times the loop will be executed, and so we start out assuming that we will be doing incremental saving of array elements as they are modified, keeping count of the number of loop executions and also pushing array elements onto stack **S** as they are modified. But after a certain threshold number of elements have been pushed onto the stack, it becomes probable the loop will cycle many times and that a large fraction of all of the array elements will be modified. At that point we abandon elementwise and pop everything off of the stack that we pushed onto it. We then start over, pushing the entire array **A** onto the stack all at once. If, at the end of the loop, we did not abandon elementwise saving of array **A**, then we push the loop count **ct** on the stack to prepare for elementwise restoration. Either way, the last thing we push on the stack is the boolean **elementwise** to tell the reverse method **E-()** which mechanism to use to restore **A**.

Storage allocation and deallocation

Allocation

Dynamic storage allocation: Naïve attempt

E()	E^+()	E^-()	E*()

T* tptr;
 	
void E() {
  P;
  tptr = new T;
  R;
}
	void E^+() {
	  P^+;
	  tptr = new T;
	  R^+;
	}
		void E^-() {
		  R^-;
		  delete(tptr);
		  P^-;
		}
			void E^*() {
			  P^*;
			  R^*;
			}

Note: this information on this slide is WRONG, as revealed in subsequent slides.

The temptation here is to believe that **`delete()`** is the proper reverse for **`new`**. It isn't.

Dynamic storage allocation: Naïve attempt

	E()	E⁺()	E⁻()	E*()

Wrong	`T* tptr;`
	
	`void E() {`
	`  P;`
	`  tptr = new T;`
	`  R;`
	`}`
		`void E⁺() {`
		`  P⁺;`
		`  tptr = new T;`
		`  R⁺;`
		`}`
			`void E⁻() {`
			`  R⁻;`
			`  delete(tptr);`
			`  P⁻;`
			`}`
				`void E*() {`
				`  P*;`
				`  R*;`
				`}`

Problem:

- **`new`** invokes the *constructor* for **`T`**
- **`delete`** invokes *destructor* for **`T`**
- … but the destructor may not be the perfect reverse of the constructor

Note: this information on this slide is WRONG, as revealed in subsequent slides.

Dynamic storage allocation: Correct version

	E()	E⁺()	E⁻()	E*()

Wrong	`T* tptr;`
	
	`void E() {`
	`  P;`
	`  tptr = new T;`
	`  R;`
	`}`
		`void E⁺() {`
		`  P⁺;`
		`  tptr = new T;`
		`  R⁺;`
		`}`
			`void E⁻() {`
			`  R⁻;`
			`  delete(tptr);`
			`  P⁻;`
			`}`
				`void E*() {`
				`  P*;`
				`  R*;`
				`}`

Right	`T* tptr;`
	
	`void E() {`
	`  P;`
	`  tptr = new T;`
	`  R;`
	`}`
		`void E⁺() {`
		`  P⁺;`
		`  tptr = new T;`
		`  R⁺;`
		`}`
			`void E⁻() {`
			`  R⁻;`
			`  tptr->T_constructor⁻();`
			`  operator delete(tptr);`
			`  P⁻;`
			`}`
				`void E*() {`
				`  P*;`
				`  R*;`
				`}`

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 35

Calling **`new T`** in the forward method is OK. But we must realize that this invokes a constructor for type **`T`** in C++, (which in turn invokes a whole hierarchy of other constructors for members in tyre and parent types, all called in canonical order). The problem with treating **`delete()`** as its reverse is that **`delete()`** calls the *destructor* for the type of data being deleted, *but the destructor is not generally the reverse of the constructor!* In general, programmer-defined constructors can have arbitrary side-effects on other variables, which may or may not be exactly reversed by the matching destructor. What we need is for each constructor to have a *reverse constructor* for T, which we denote here by **`T_constructor⁻()`**. We call that first, and the use the obscure C++ construct **`operator delete()`** to finally free the storage allocated by **`new`**. In C++ **`operator delete()`** does not call a destructor.

Deallocation

Dynamic storage deallocation: Naïve attempt

E()	E⁺()	E⁻()	E*()

T* tptr;
 	
void E() {
  P;
  delete(tptr);
  R;
}
	void E⁺() {
	  P⁺;
	  Q.push(tptr);
	  R⁺;
	}
		void E⁻() {
		  R⁻();
		  Q.push⁻();
		  P⁻();
		}
			void E*() {
			  P*;
			  delete(Q.front());
			  Q.pop();
			  R*;
			}

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 37

Note: this information on this slide is **WRONG**, as revealed in subsequent slides.

Here we recognize that the **`delete()`** primitive is fundamentally irreversible, so we delay it to the commute routine **`E*()`**.

Note that in **`E⁻()`** we use **`push⁻()`** as the reverse of push. In the C++ STL there does not happen to be a primitive to remove an entry added to the back of a Queue, so we presume that the code for Queue has been run through Backstroke to create **`push⁻()`** (or that someone wrote it by hand).

Dynamic storage deallocation: Naïve attempt

	E()	E^+()	E^-()	E*()

Wrong	T* tptr;
	
	void E() {
	P;
	delete(tptr);
	R;
	}
		void E^+() {
		P^+;
		Q.push(tptr);
		R^+;
		}
			void E^-() {
			R^-;
			Q.pop();
			P^-;
			}
				void E^*() {
				P^*;
				delete(Q.front());
				Q.pop();
				R^*;
				}

Problem:

- `delete` invokes the destructor, and that should be called in E^+ because R may depend on it
- Thus, the reverse destructor also has to be called in the E^-

Note: the information on this slide is WRONG, as revealed in subsequent slides.

Dynamic storage deallocation

	E()	E⁺()	E⁻()	E*()

Wrong	T* tptr;
	
	void E() {
	  P;
	  delete(tptr);
	  R;
	}
		void E⁺() {
		  P⁺;
		  Q.push(tptr);
		  R⁺;
		}
			void E⁻() {
			  R⁻;
			  Q.push⁻();
			  P⁻;
			}
				void E*() {
				  P*;
				  delete(Q.front());
				  Q.pop();
				  R*;
				}

Right	T* tptr;
	
	void E() {
	  P;
	  delete(tptr);
	  R;
	}
		void E⁺() {
		  P⁺;
		  tptr->T_destructor⁺();
		  Q.push(tptr);
		  R⁺;
		}
			void E⁻() {
			  R⁻;
			  Q.push⁻();
			  tptr->T_destructor⁻();
			  P⁻;
			}
				void E*() {
				  P*;
				  operator
				  delete(Q.front(tptr));
				  Q.pop();
				  R*;
				}

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 39

As with new, a similar problem arises in reversing **`delete()`** in that **`delete()`** invokes the destructor for the type of object being destroyed. We need to invoke that destructor in the forward routine, because the side effects of the destructor on other variables can be felt in subsequent statements (**`R`**). However, we need to run not the original destructor, **`tptr->T_destruct()`**, but its forward instrumented version, **`tptr->T_destruct+()`**. And we need to be able to reverse the effects of **`tptr->T_destruct+()`** in the **`E⁻()`** by calling **`tptr->T_destruct-()`**. But while we have to reverse the effects of the destructor in **`E⁻()`**, we cannot really free the storage associated with the object in the **`E⁻() routine`**, because if we do the storage could be re-allocated and overwritten, and the overwriting would make it impossible for us to reverse the action in **`E⁻()`**. So we delay the actual freeing of the storage until the commit routine, and then we do it using the **`operator delete()`** construct.

Output

Output

E()	E⁺()	E⁻()	E*()

File file;
 	
void E() {
  P;
  file.output(expr);
  R;
}
	void E⁺() {
	  P⁺;
	  Q.push(file);
	  value = expr;
	  Q.push(value);
	  R⁺;
	}
		void E⁻() {
		  R⁻;
		  Q.push⁻();
		  Q.push⁻();
		  P⁻;
		}
			void E*() {
			  P*;
			  File f = Q.front();
			  Q.pop();
			  f.output(Q.front());
			  Q.pop();
			  R*;
			}

- Both `file` and the value of `expr` must be pushed into the stack *by value*
- If `expr` evaluates to an array, the entire array must be pushed onto the stack
- If `expr` evaluates to a struct, or class type, fully deep copies are required using an appropriate, perhaps non-default copy constructor.

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 41

Note: **`expr`** is a side-effect free expression. When we push data onto the queue **`Q`** for processing at commit time, we must push *values,* not expressions to be evaluated.

End