Stitched Together
Transitioning CMS to a Hierarchical Threaded Framework

Elizabeth Sexton-Kennedy
Christopher Jones *FNAL*
On behalf of CMS Offline

Outline

Goals

Design

Thread Safety

Tools

Goals

Better scaling of system resources as core count increases

Puts less burdens on existing grid sites since one batch slot uses more cores

Potential to use sites with lower available resources

More sharing between cores

Share infrequently updated memory

- conditions
- I/O buffers

Share file handles

Share network connections

Minimize changes to existing framework and user facing interfaces

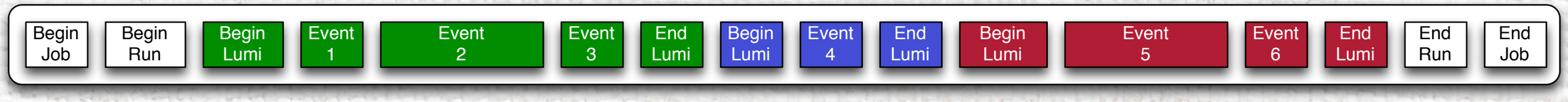
Design

CMS


Button
Center back
Straight grain of
Fold line
Center front line
Seam line
Simplicity 4700
ONE SIZE DOLL
COAT AND SLEEVE FRONT VIEW 2
½" Hem allowed
½" Hem allowed
BLOUSE
Center back
Simplicity 4700
ONE SIZE DOLL
SCARF VIEW 1
Straight grain of fabric

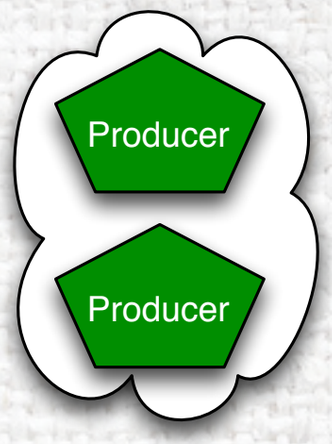
Legacy Design

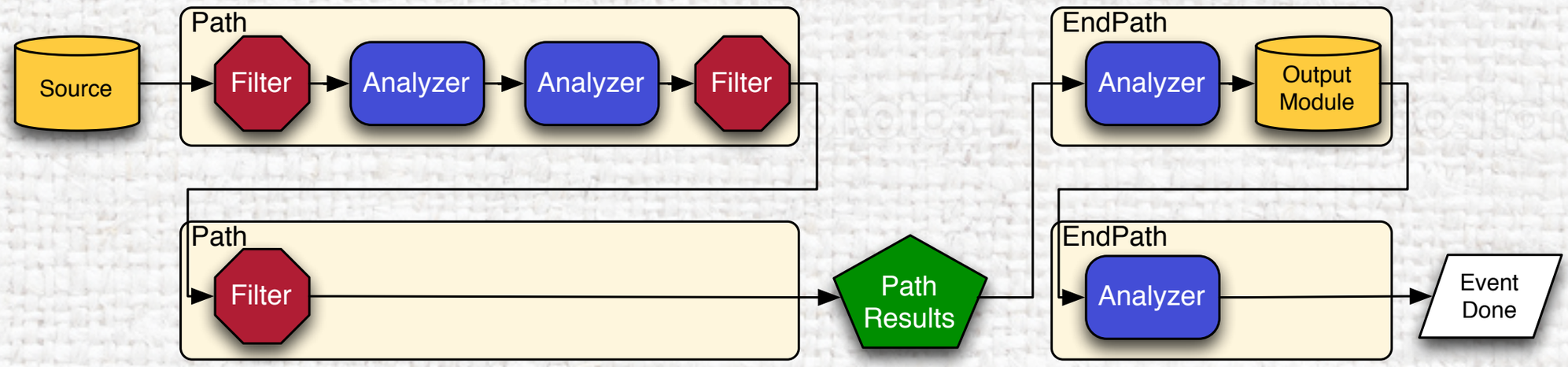
State Transitions



Event Processing

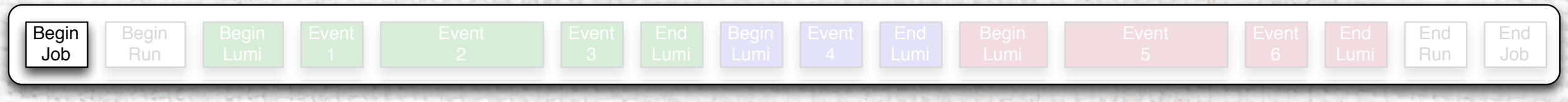
Algorithms are encapsulated into modules





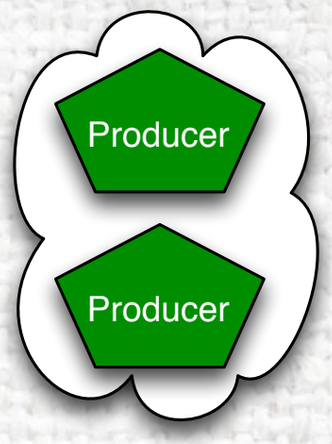
Legacy Design

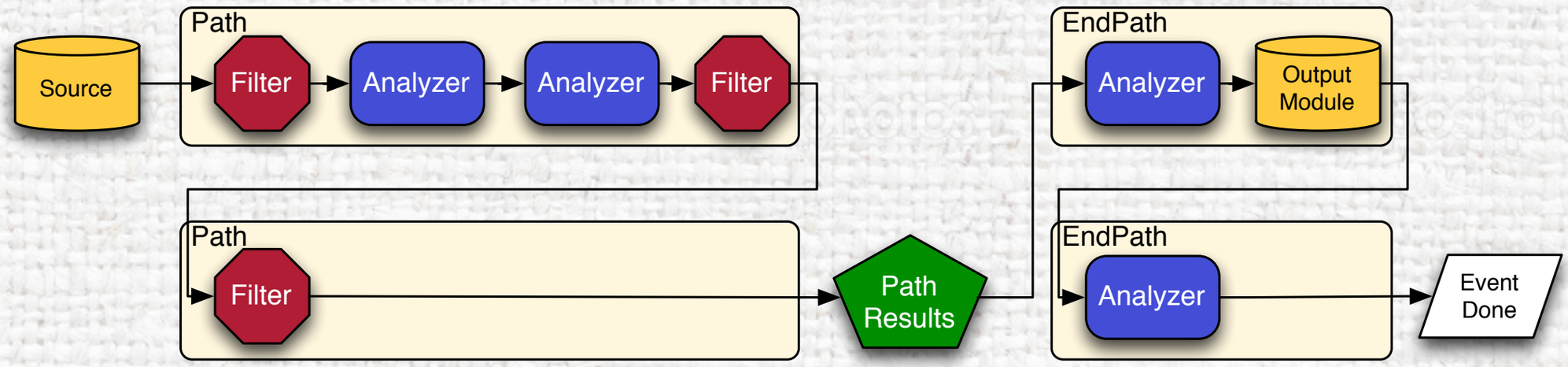
State Transitions



Event Processing

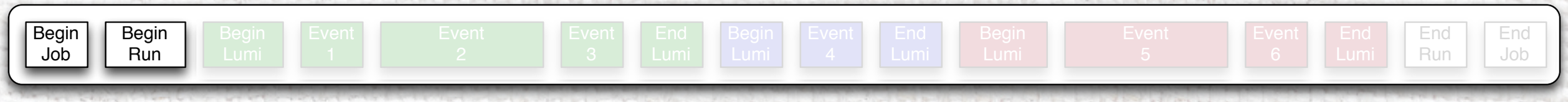
Algorithms are encapsulated into modules





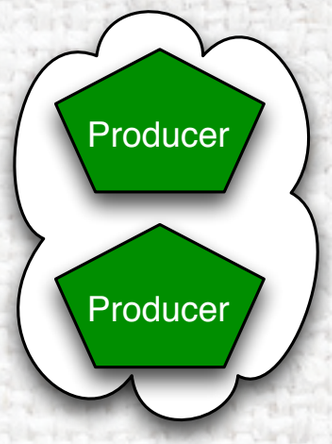
Legacy Design

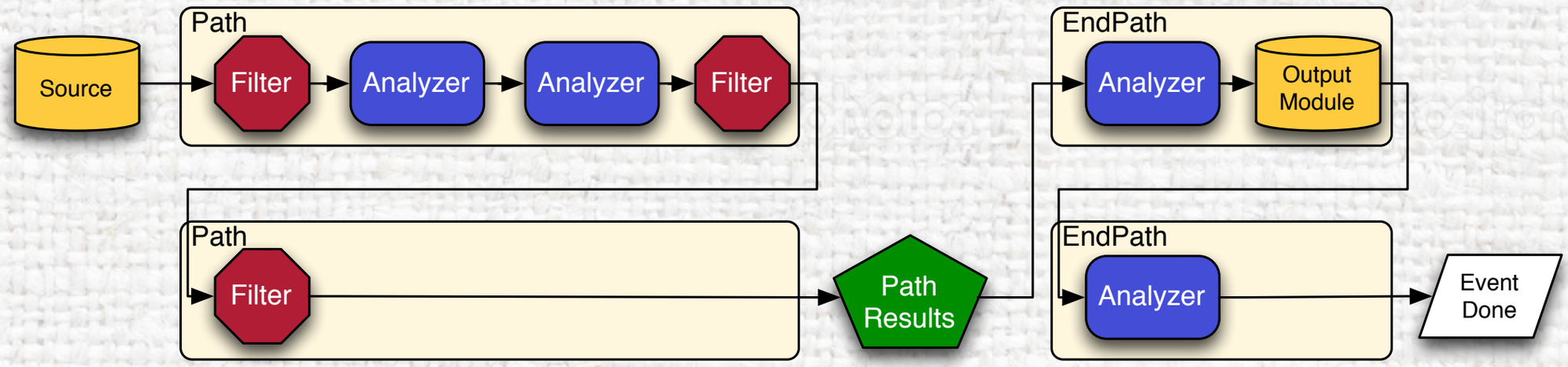
State Transitions



Event Processing

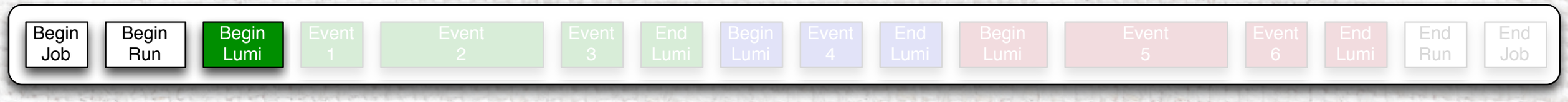
Algorithms are encapsulated into modules





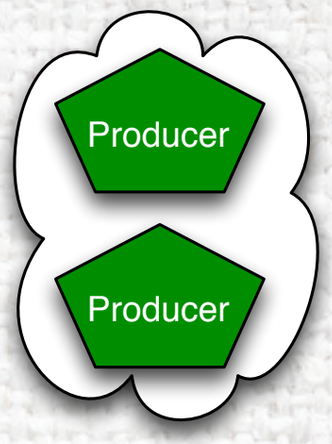
Legacy Design

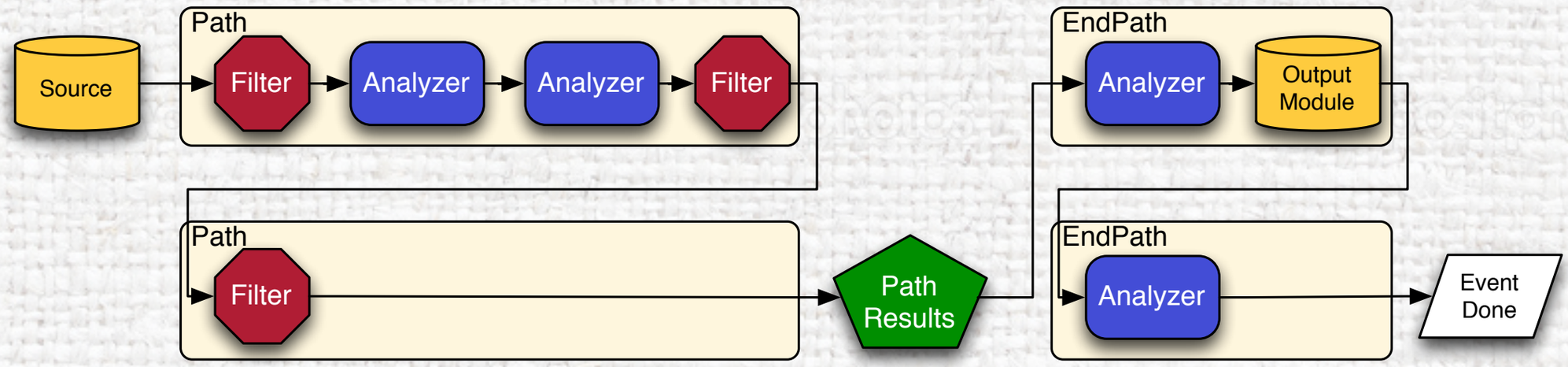
State Transitions



Event Processing

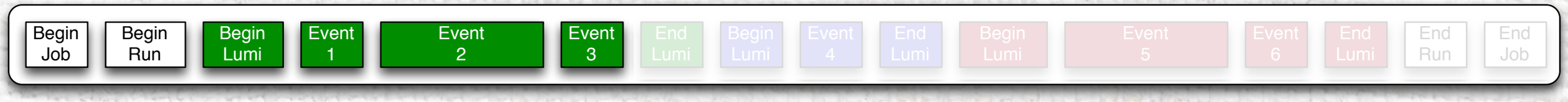
Algorithms are encapsulated into modules





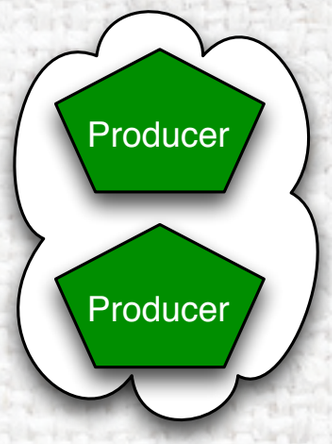
Legacy Design

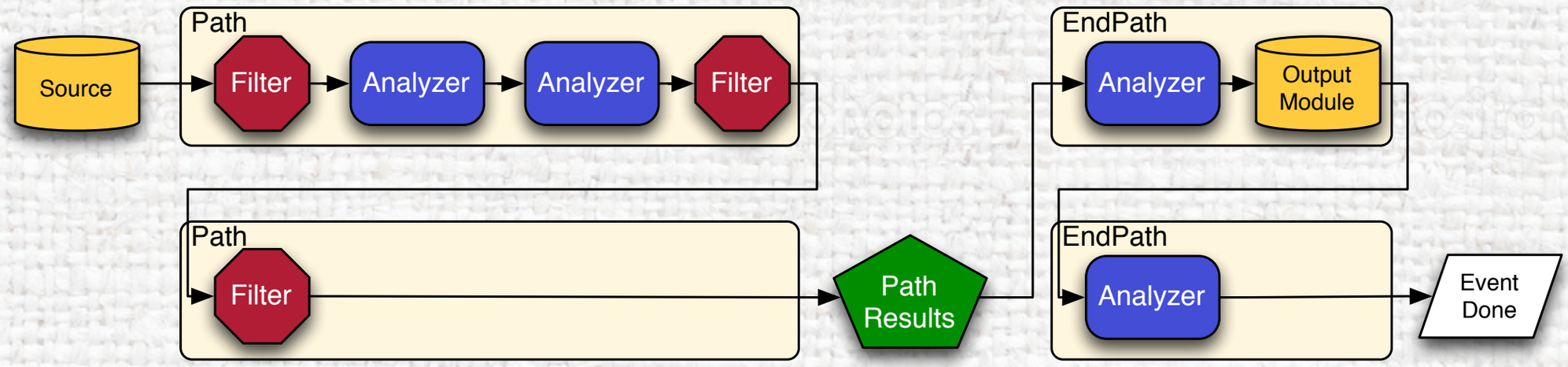
State Transitions



Event Processing

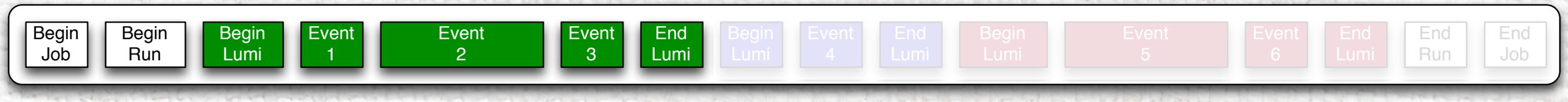
Algorithms are encapsulated into modules





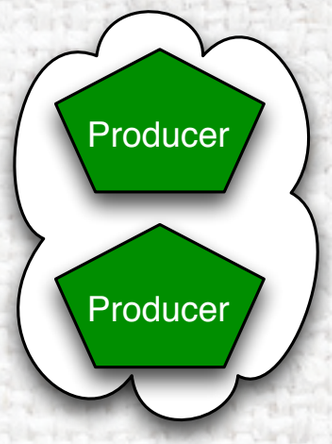
Legacy Design

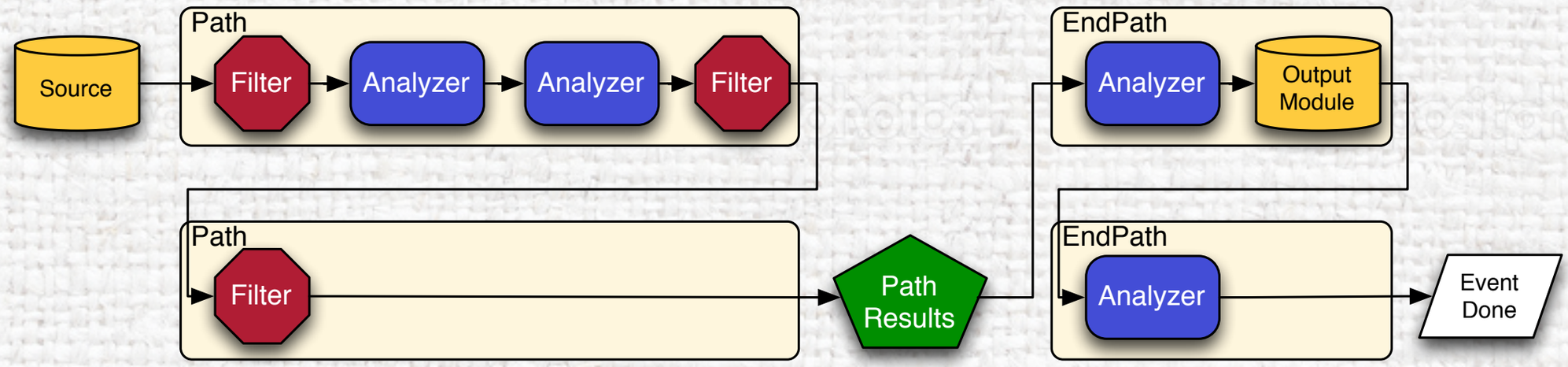
State Transitions



Event Processing

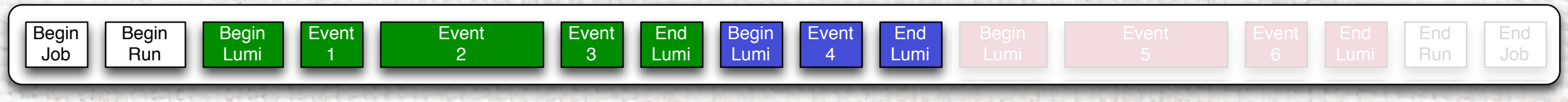
Algorithms are encapsulated into modules





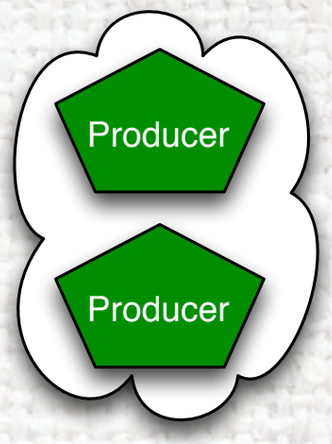
Legacy Design

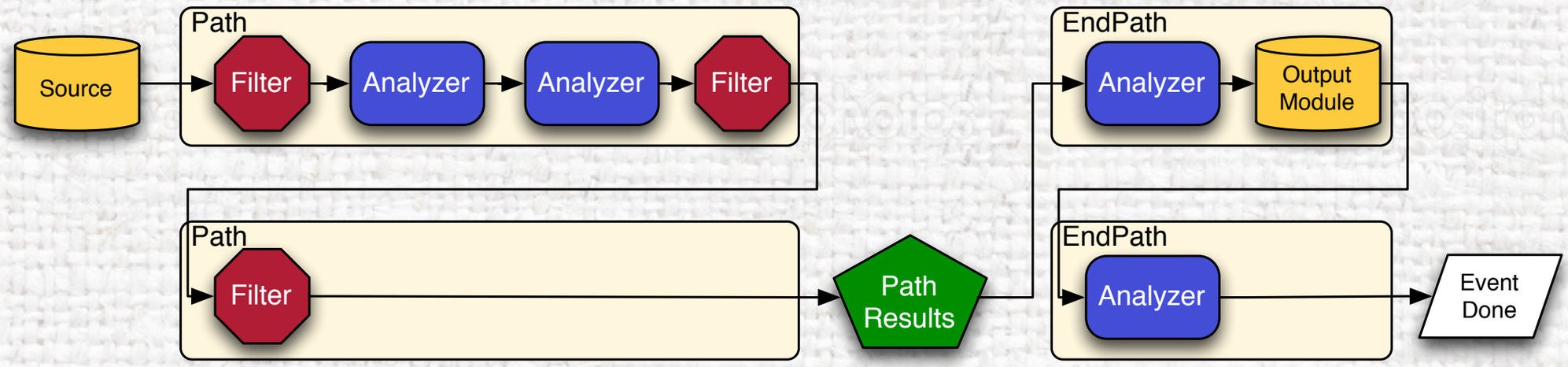
State Transitions



Event Processing

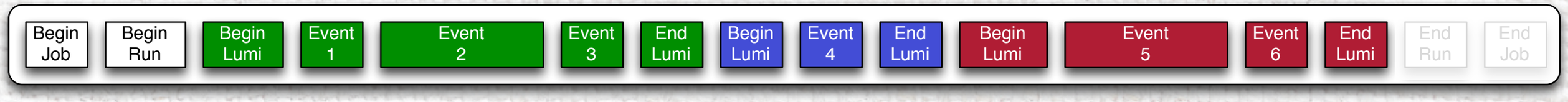
Algorithms are encapsulated into modules





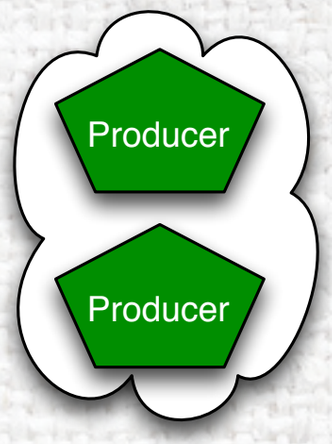
Legacy Design

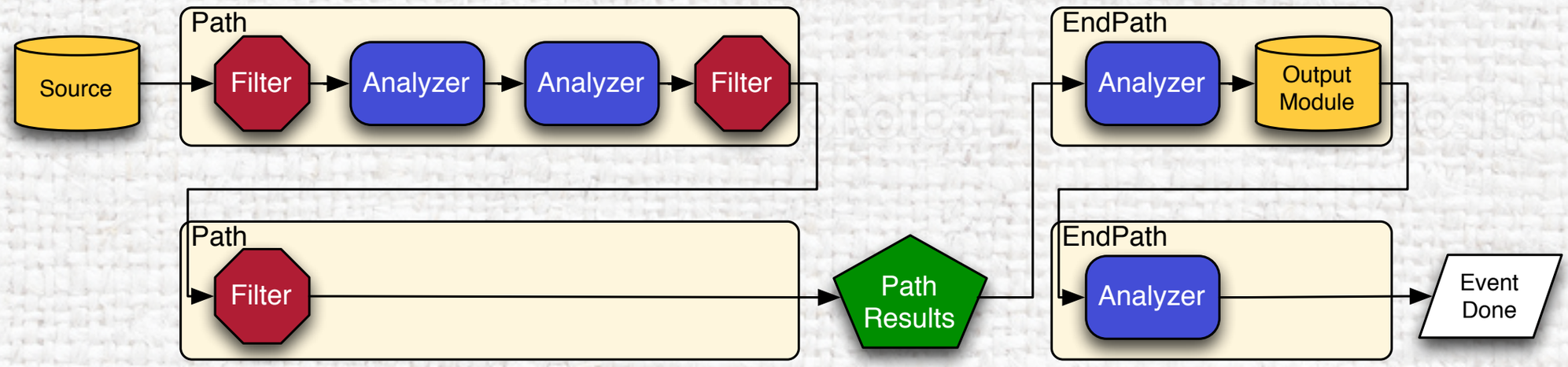
State Transitions



Event Processing

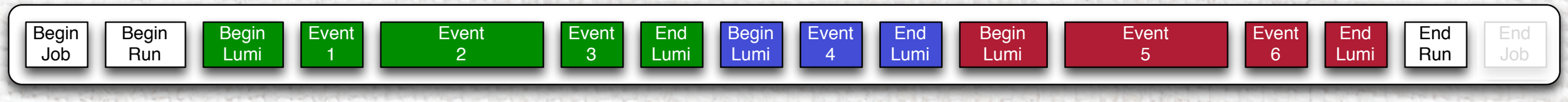
Algorithms are encapsulated into modules





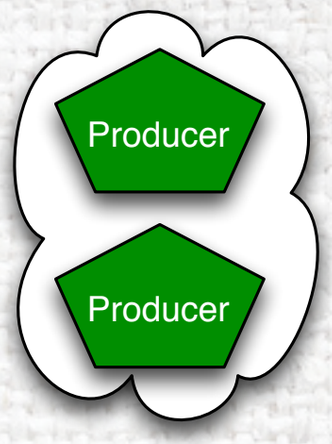
Legacy Design

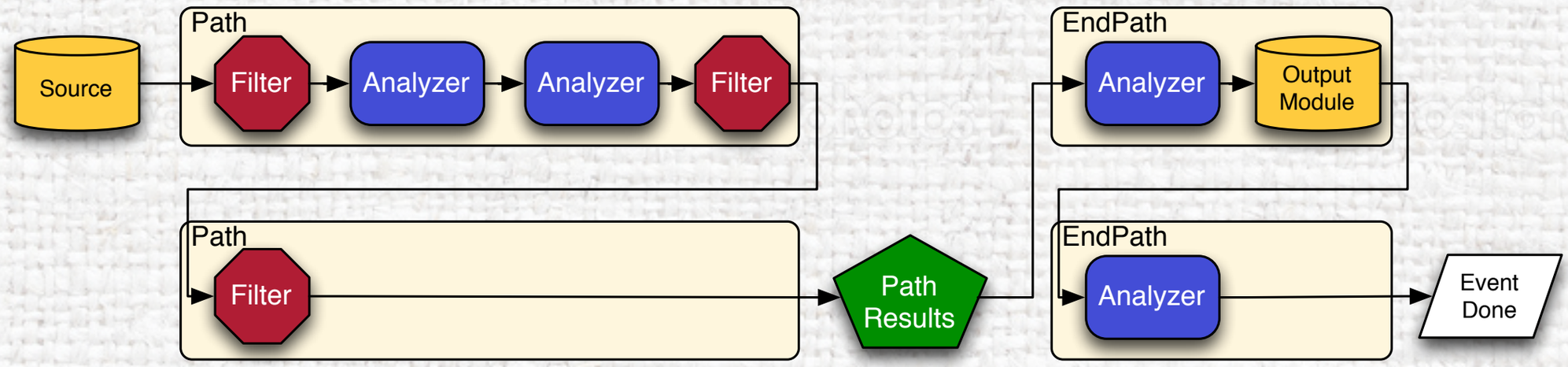
State Transitions



Event Processing

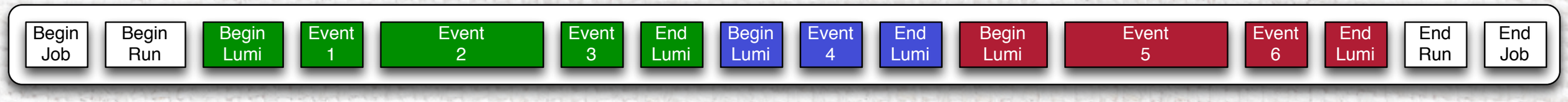
Algorithms are encapsulated into modules





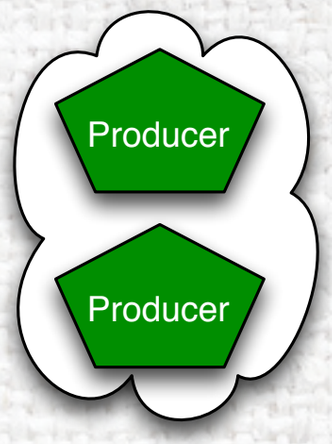
Legacy Design

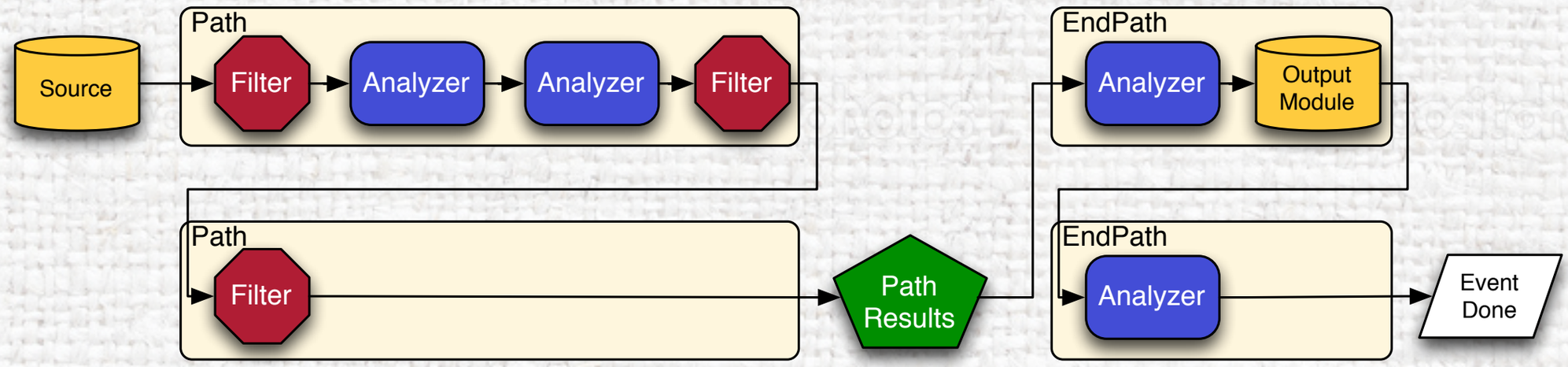
State Transitions



Event Processing

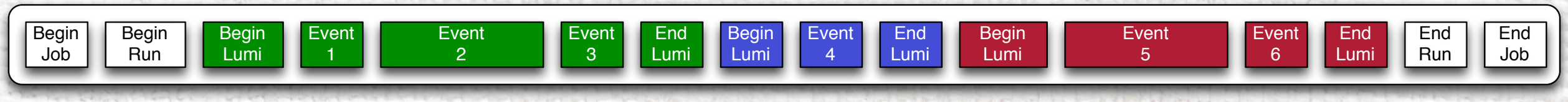
Algorithms are encapsulated into modules





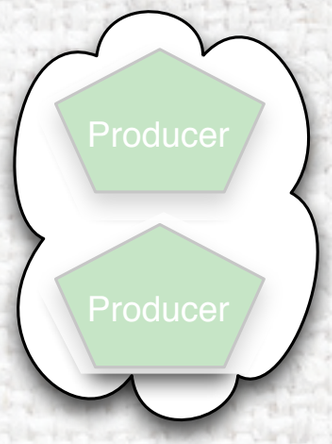
Legacy Design

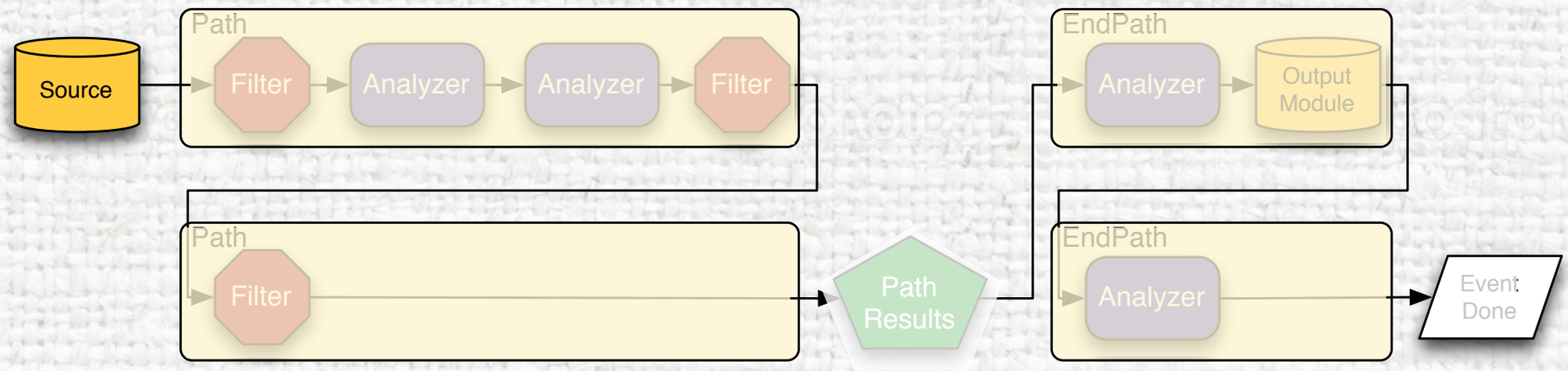
State Transitions



Event Processing

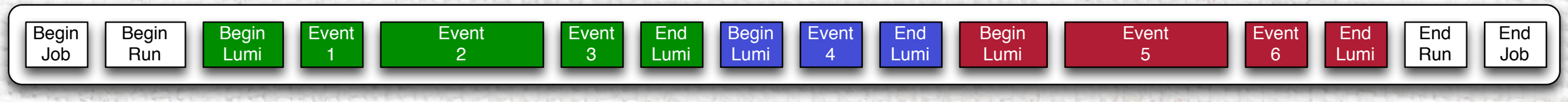
Algorithms are encapsulated into modules





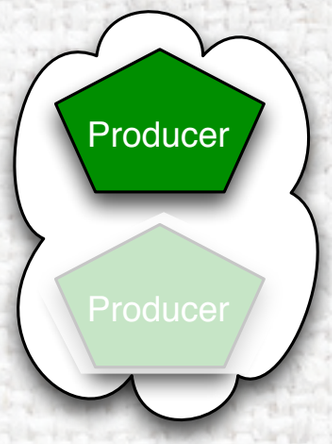
Legacy Design

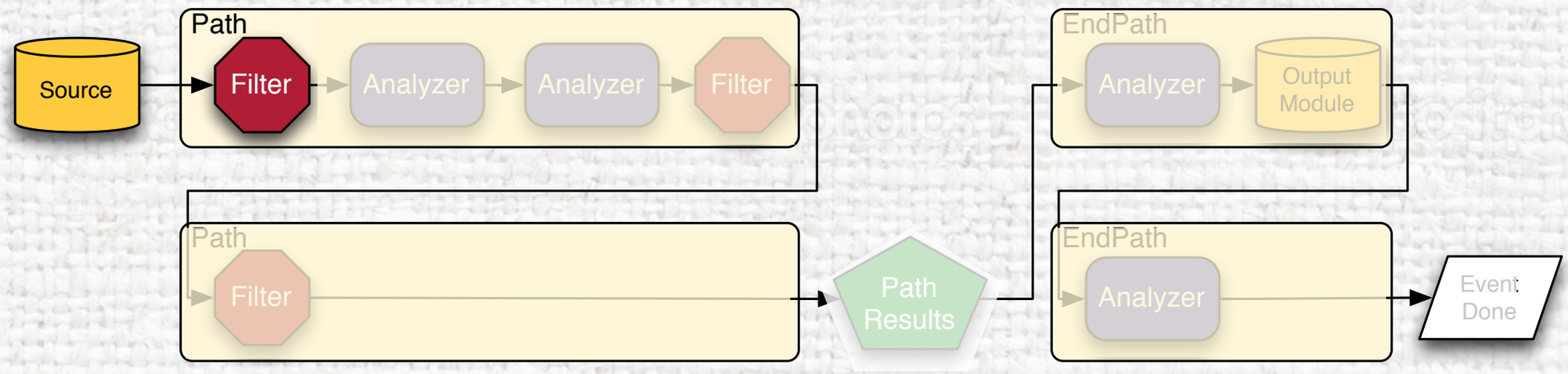
State Transitions



Event Processing

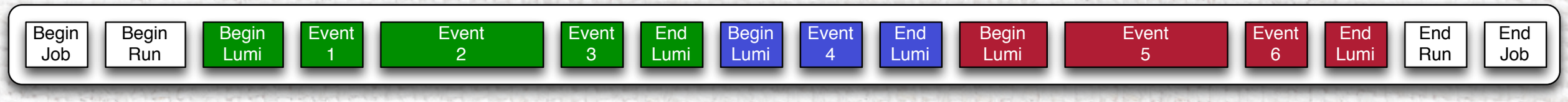
Algorithms are encapsulated into modules





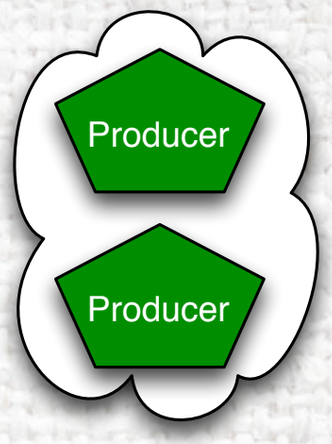
Legacy Design

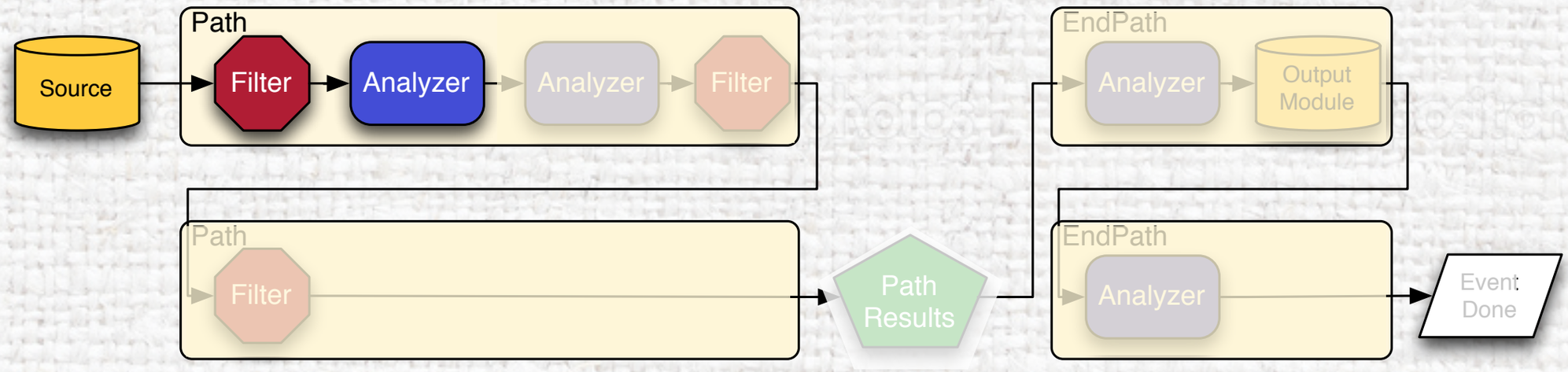
State Transitions



Event Processing

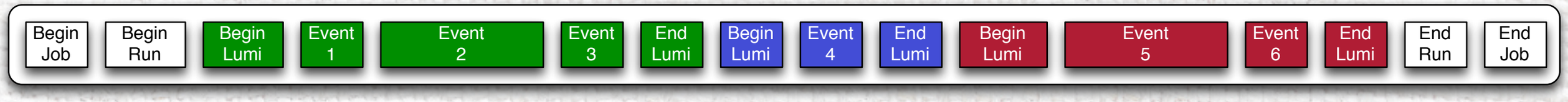
Algorithms are encapsulated into modules





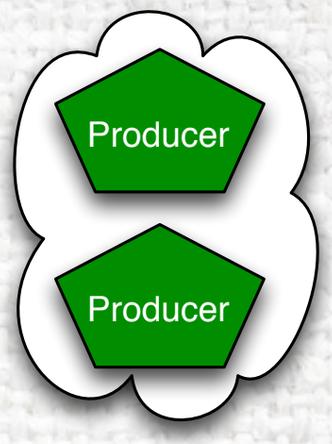
Legacy Design

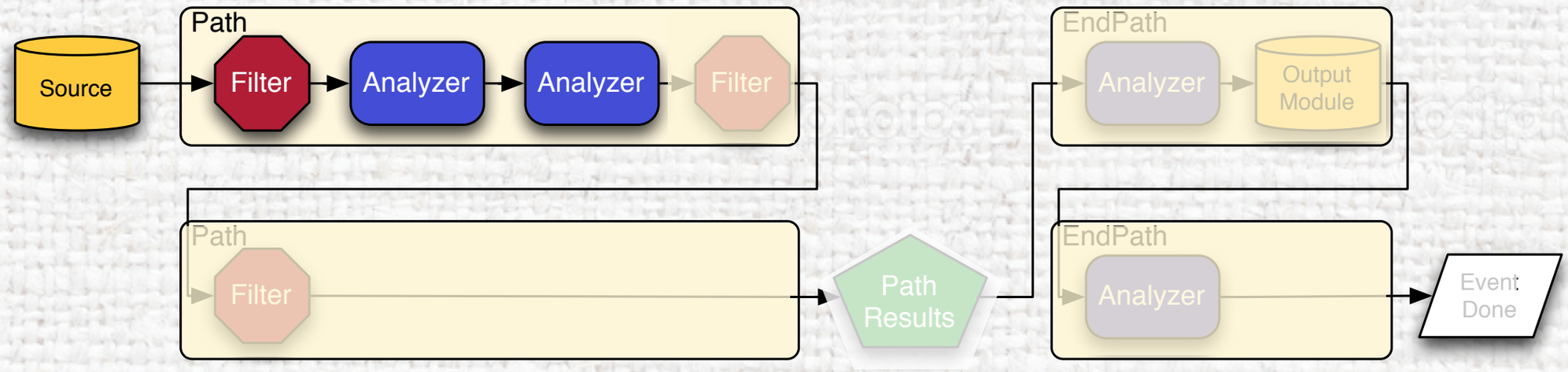
State Transitions



Event Processing

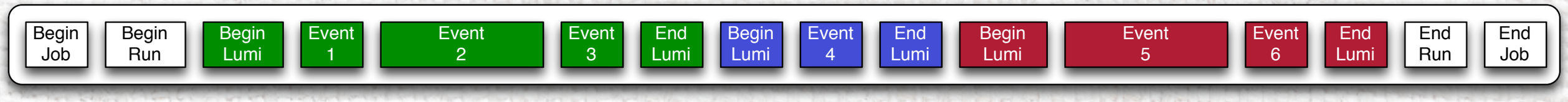
Algorithms are encapsulated into modules





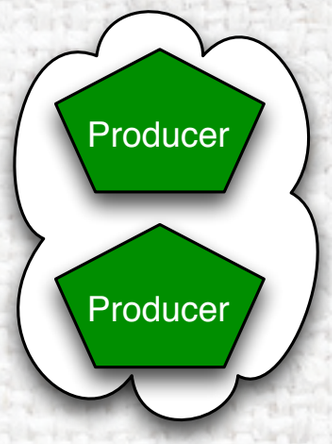
Legacy Design

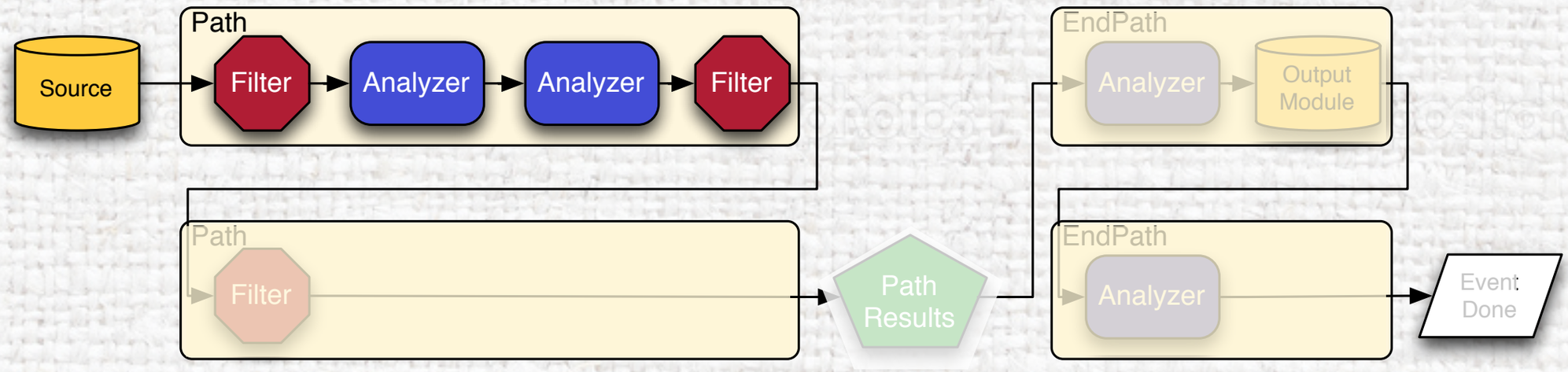
State Transitions



Event Processing

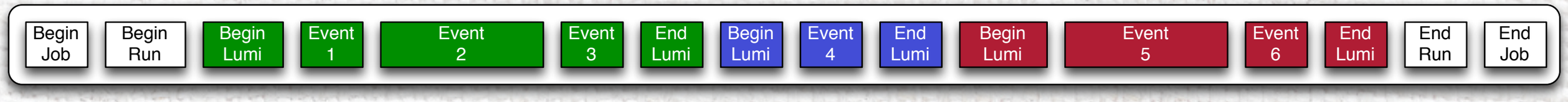
Algorithms are encapsulated into modules





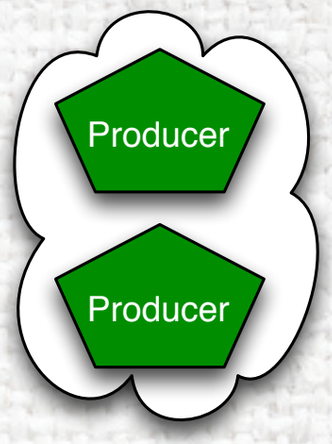
Legacy Design

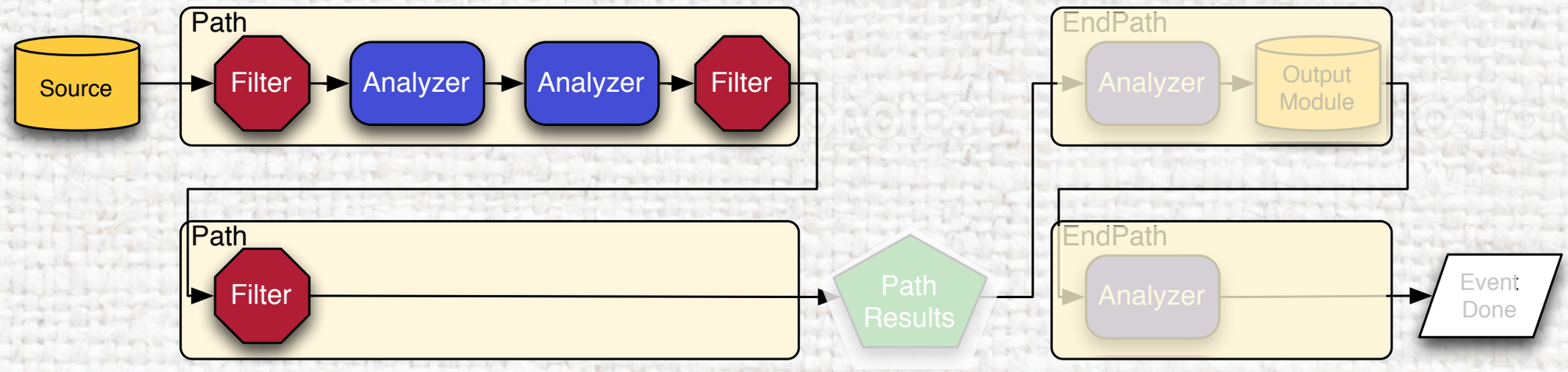
State Transitions



Event Processing

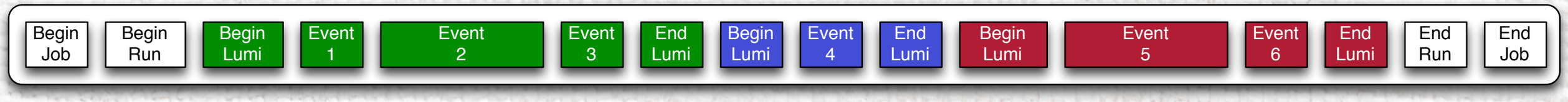
Algorithms are encapsulated into modules





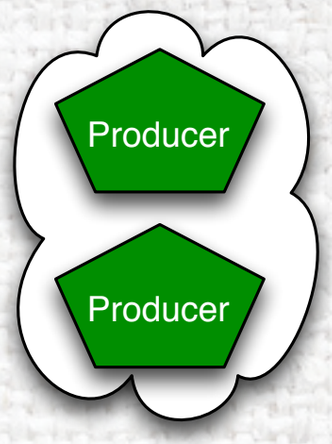
Legacy Design

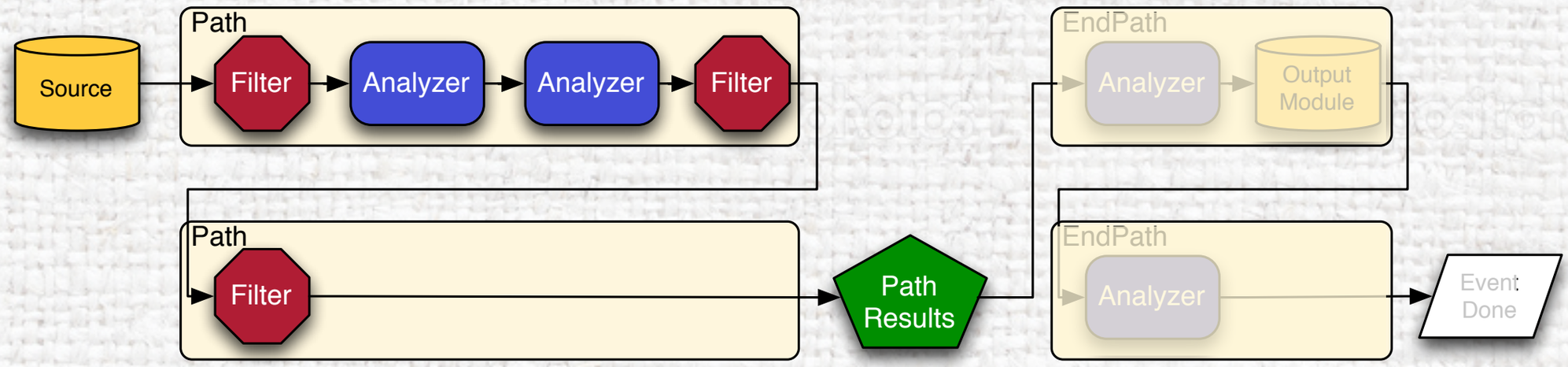
State Transitions



Event Processing

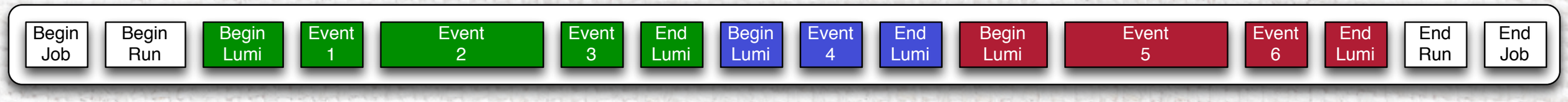
Algorithms are encapsulated into modules





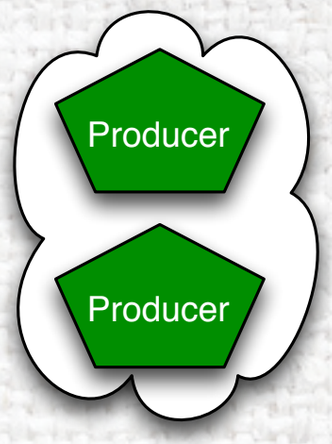
Legacy Design

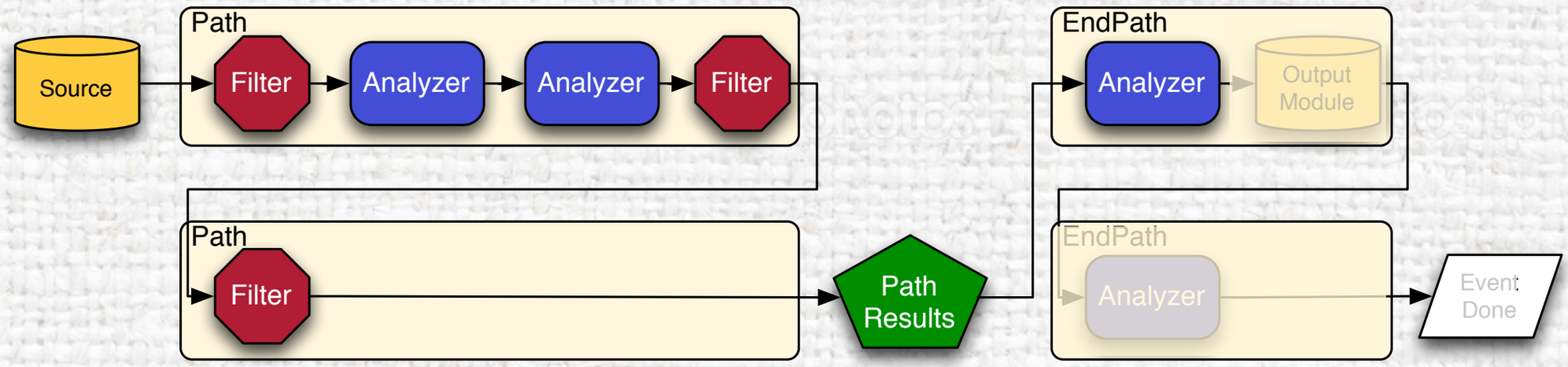
State Transitions



Event Processing

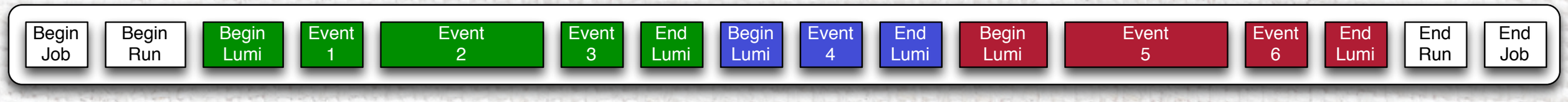
Algorithms are encapsulated into modules





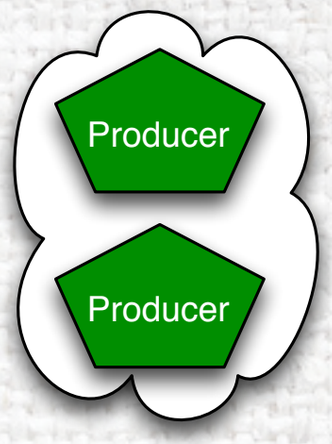
Legacy Design

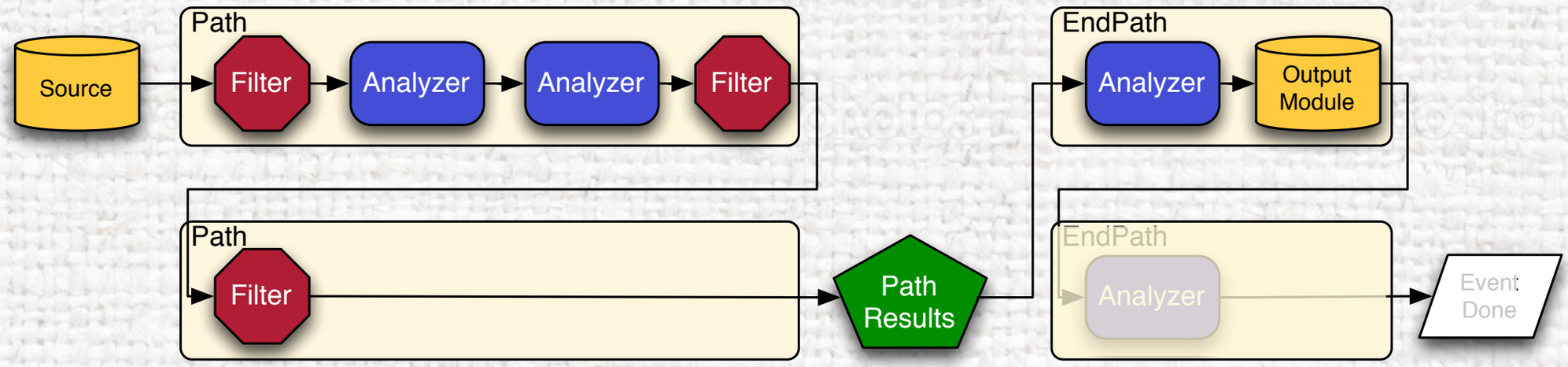
State Transitions



Event Processing

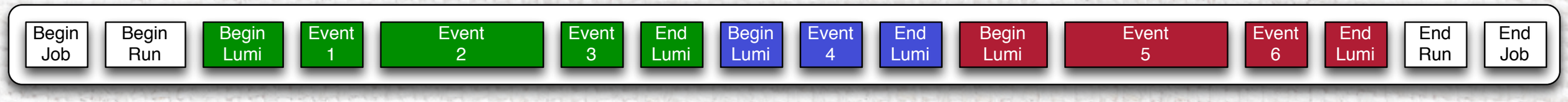
Algorithms are encapsulated into modules





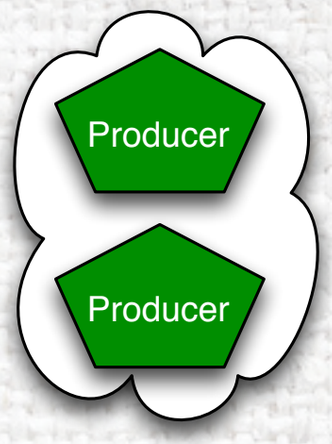
Legacy Design

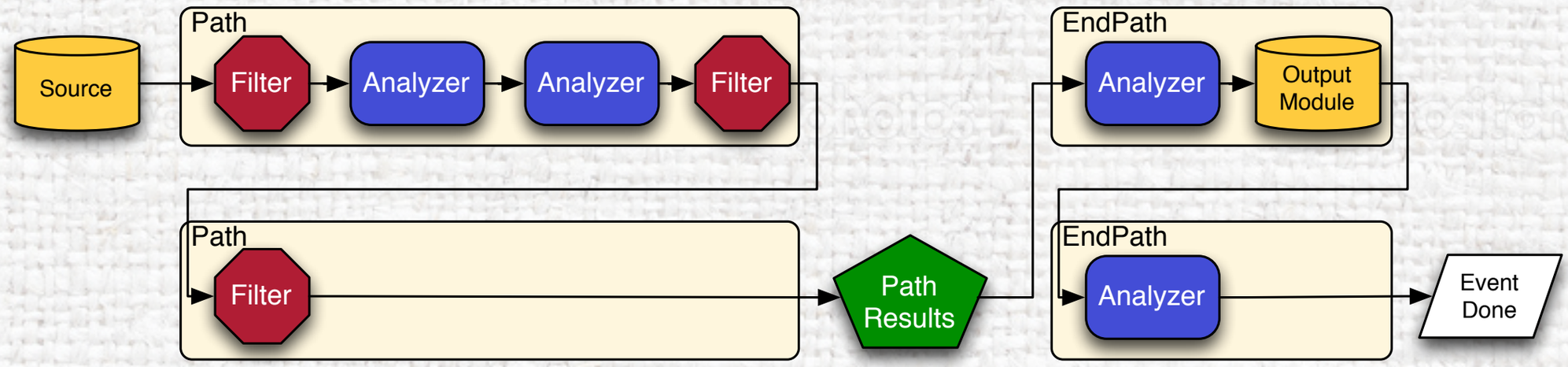
State Transitions



Event Processing

Algorithms are encapsulated into modules





Threaded Design

Run multiple transitions, i.e. events, concurrently

Introduces new concepts: *Global* and *Stream*

Within one event run multiple modules concurrently

Have to take into account module dependencies

Want to minimize any required changes to module code

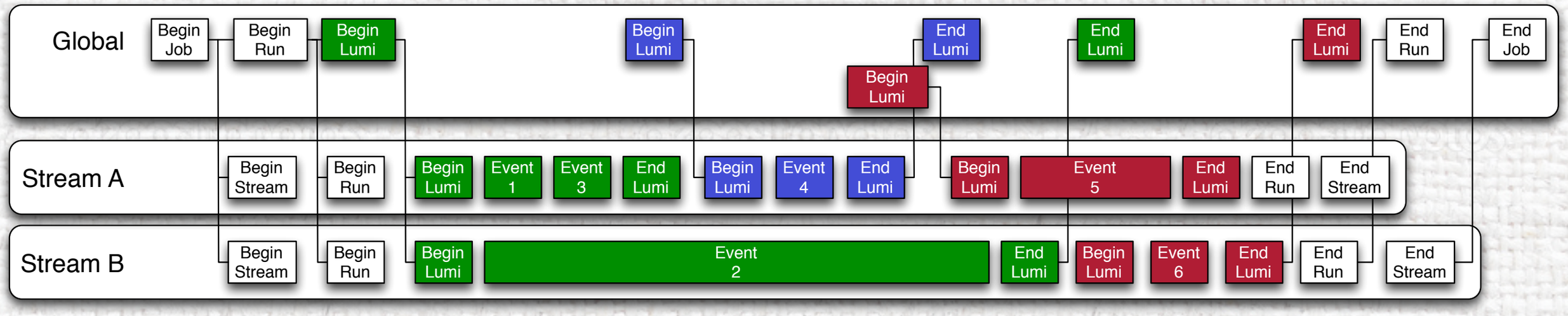
Within one module be able to run multiple tasks concurrently

Intel's Threaded Building Blocks used for all of the above

Break down work into 'tasks' and TBB can run the tasks in parallel

http://threadingbuildingblocks.org

Concurrent Transitions



Global

Sees transitions on a 'global' scale

- see begin of Run and begin of Lumi when source first reads them
- sees end of Run and end of Lumi once all processing has finished for them

Multiple transitions can be running concurrently

Events are not seen 'globally'

Stream

Processes transitions serially

- begin run, begin lumi, events, end lumi, end run

Multiple streams can be running concurrently each with own events

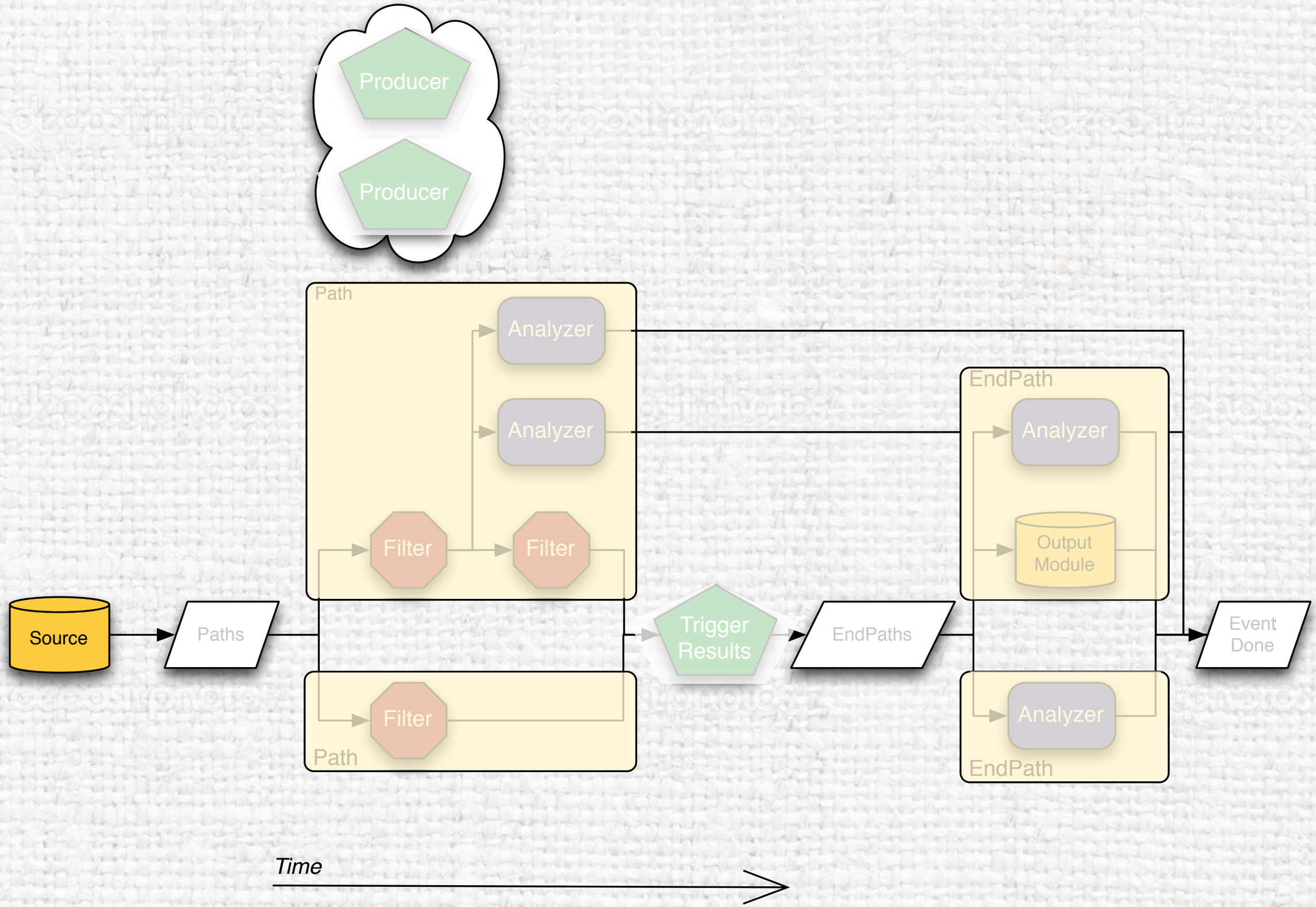
- One stream only sees a subset of the events in a job

Present CMS framework is equivalent to running with only one stream

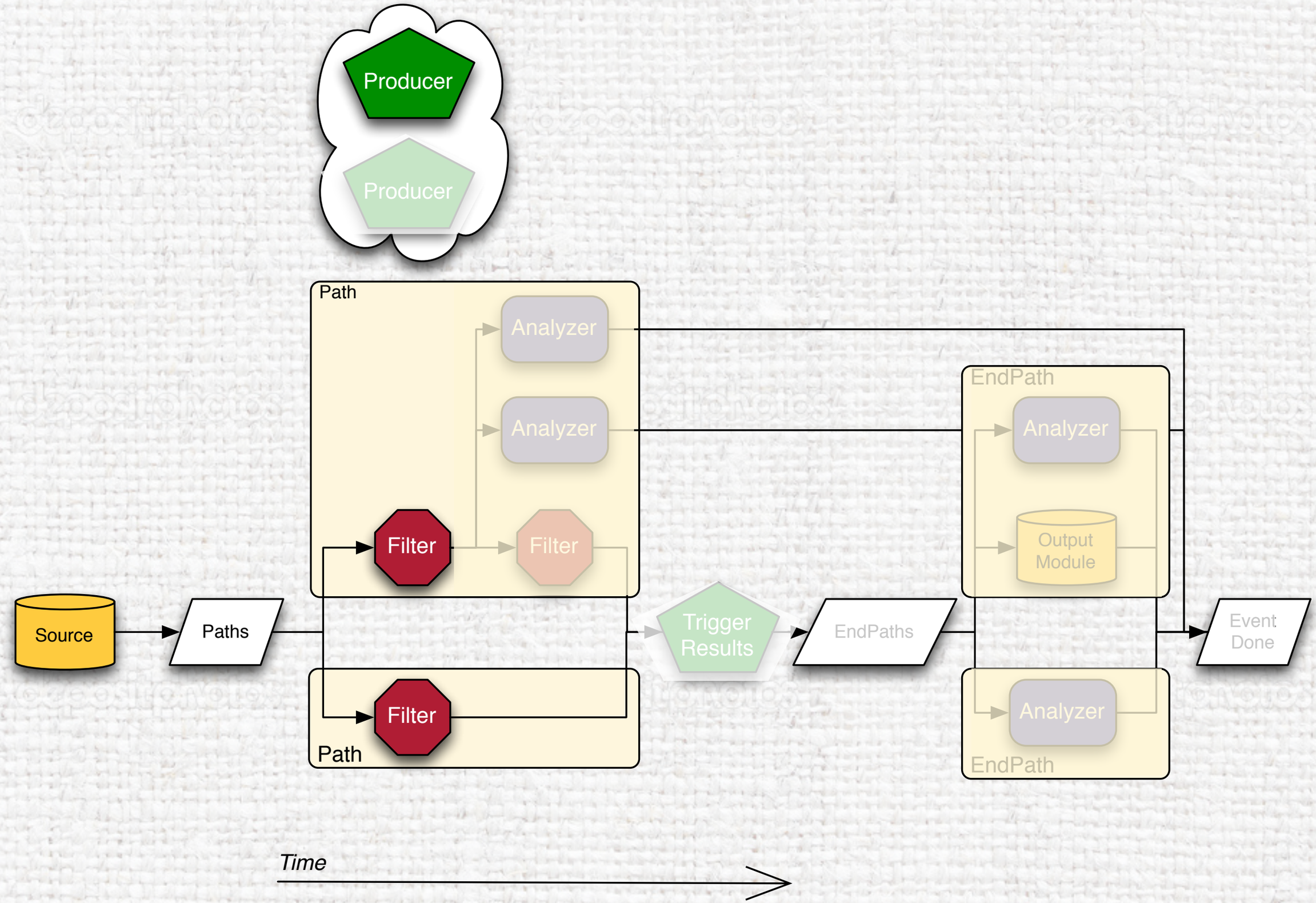
Paths and EndPaths are a per Stream construct

- The same module can be shared across Streams
- The Stream knows if a module was run for a particular event

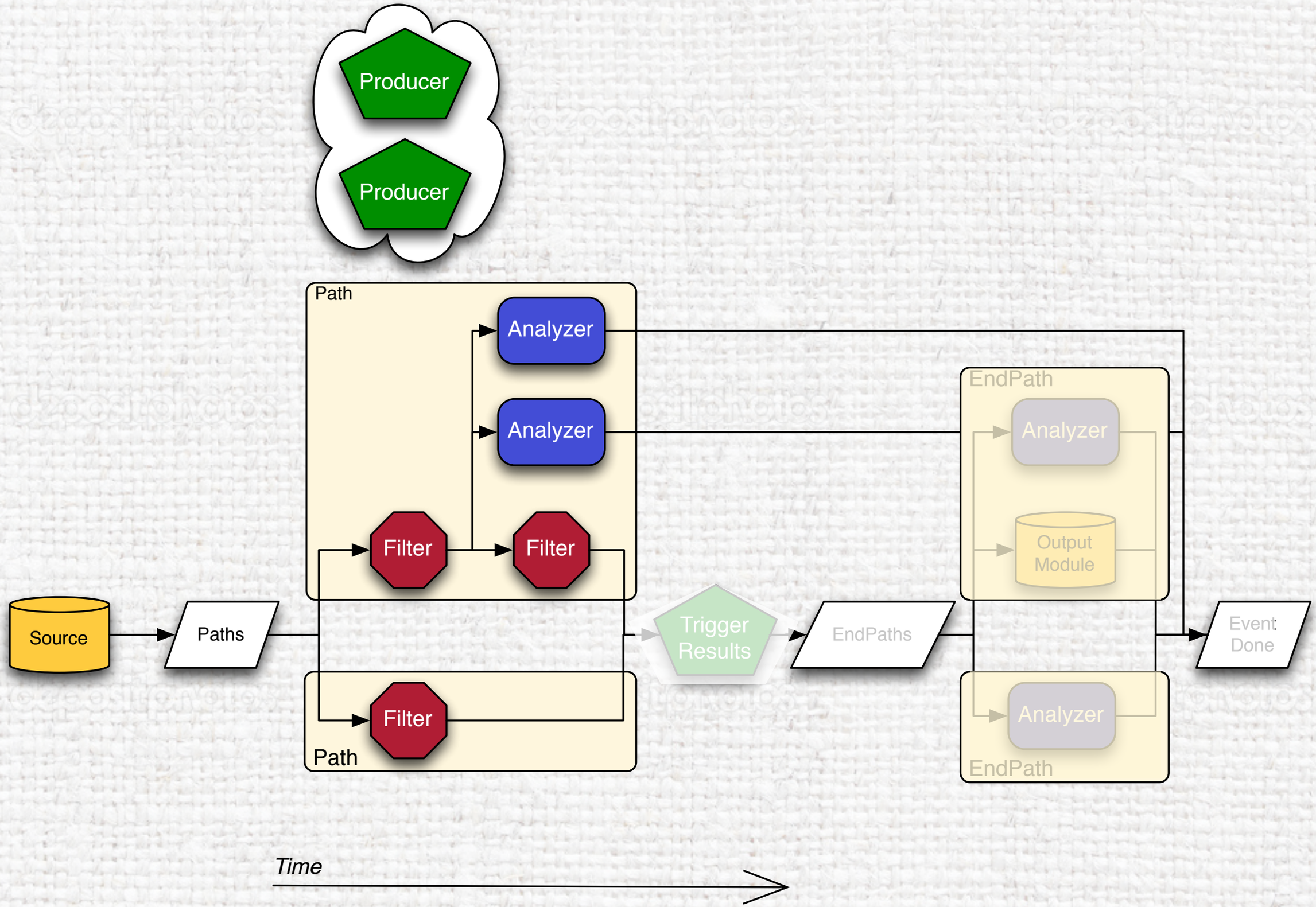
Concurrent Modules



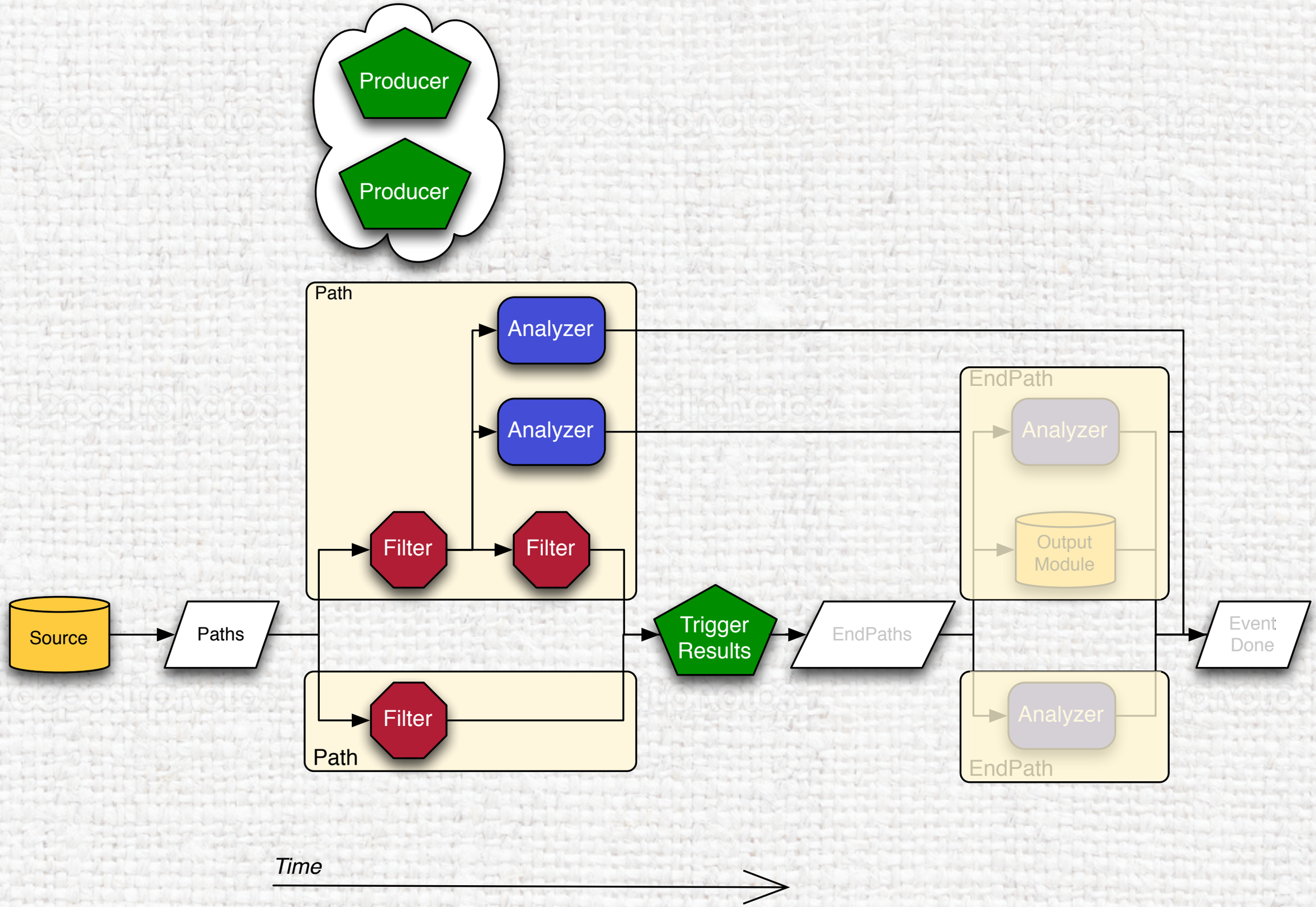
Concurrent Modules



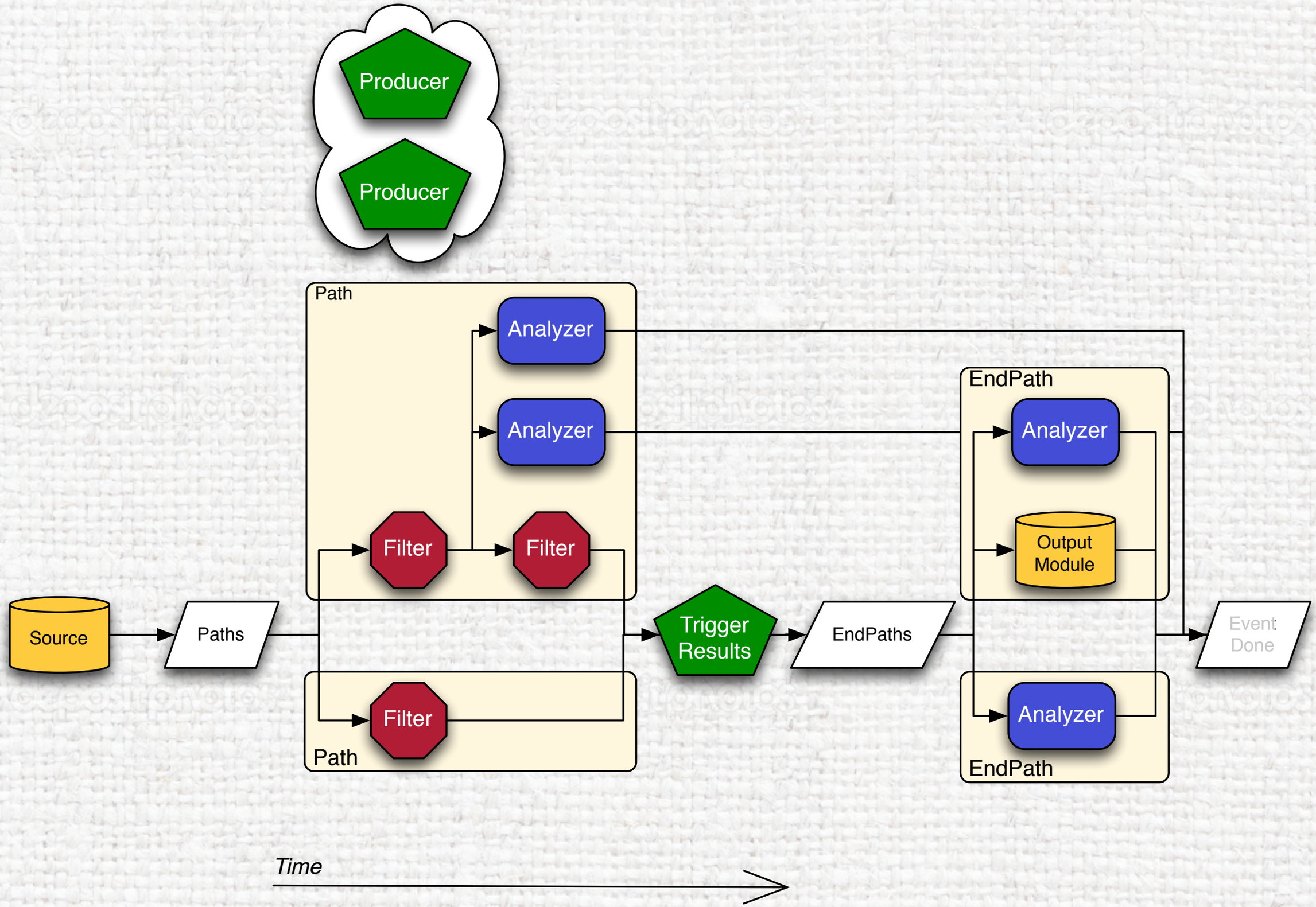
Concurrent Modules



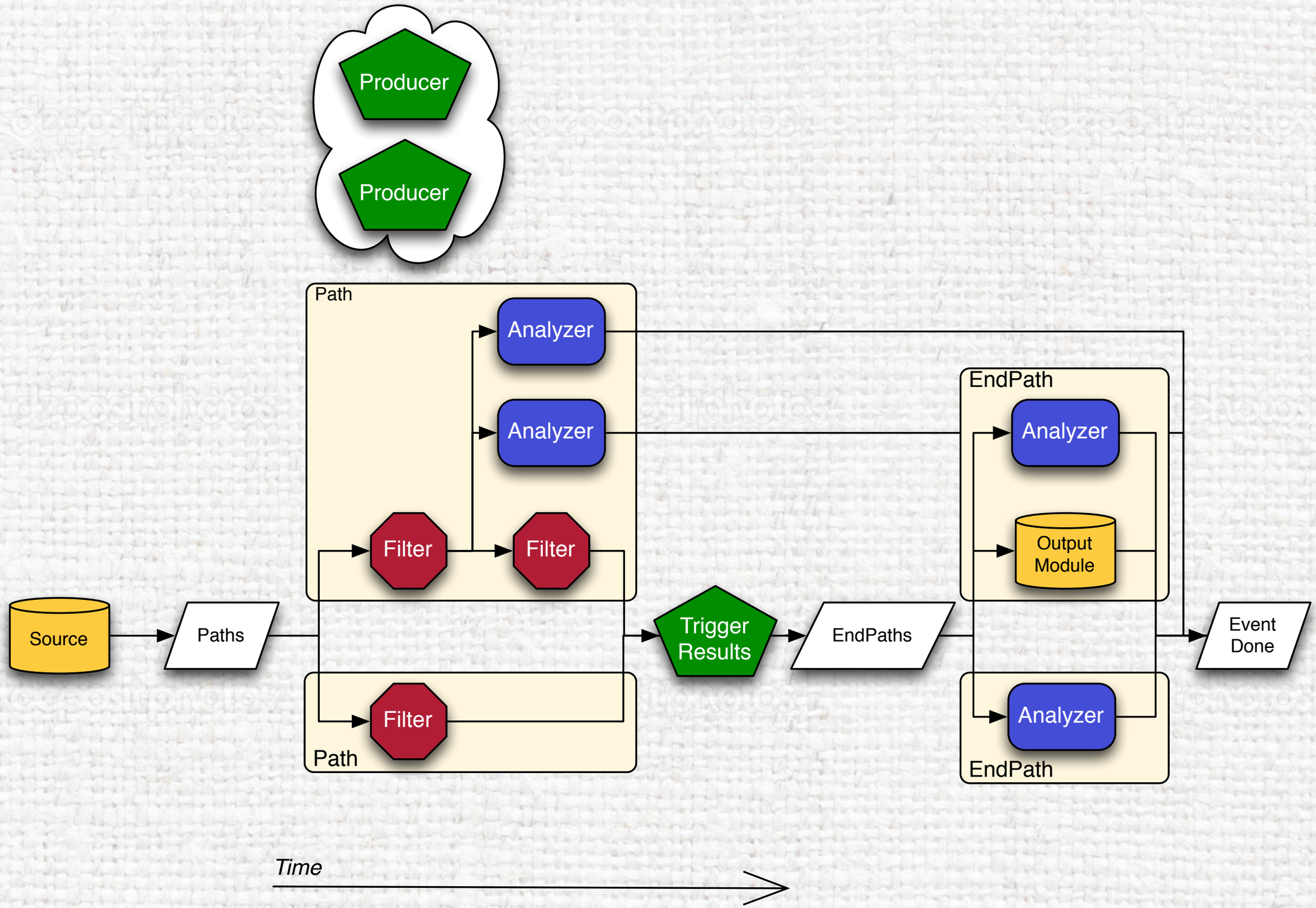
Concurrent Modules



Concurrent Modules



Concurrent Modules



Concurrent Tasks

Can use TBB directly inside a module

TBB will handle scheduling tasks for both modules and sub-modules

TBB has some convenience functions

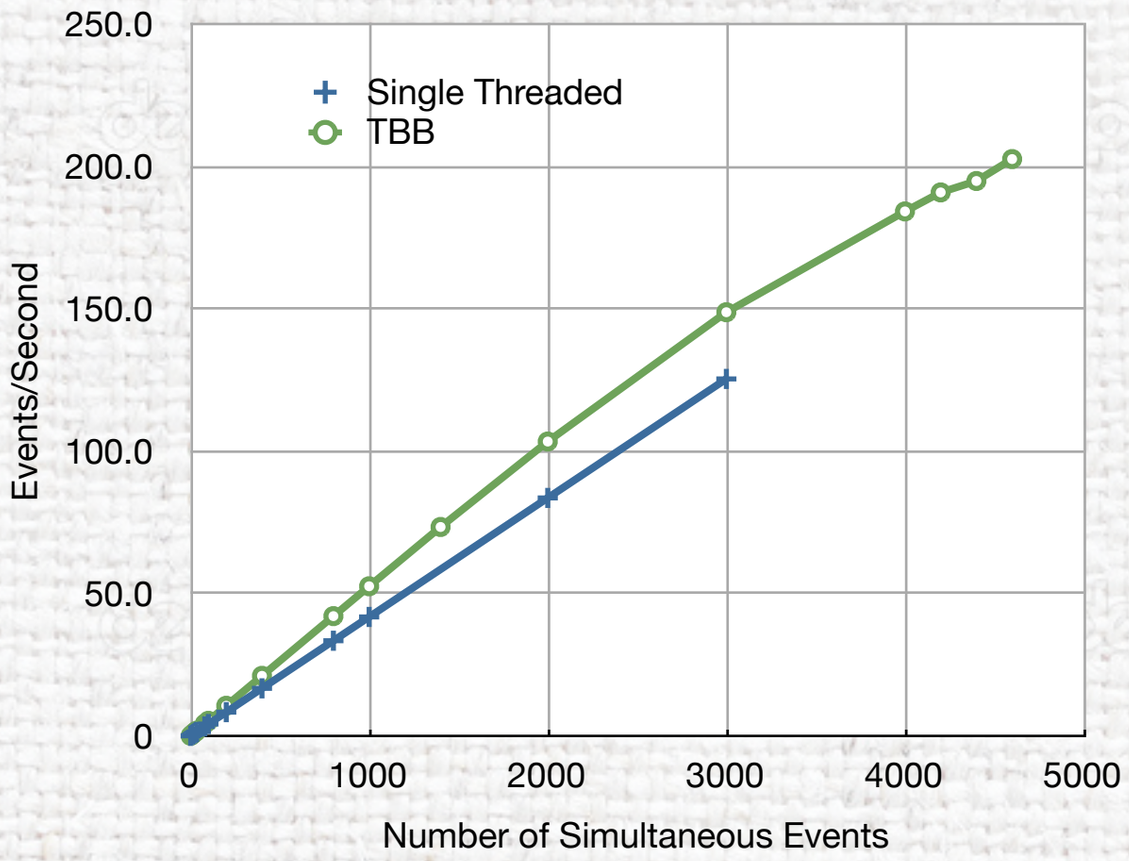
```
std::vector<Results> results(input.size(),Results());
tbb::parallel_for(0U,input.size(), DoWork(results) );
```

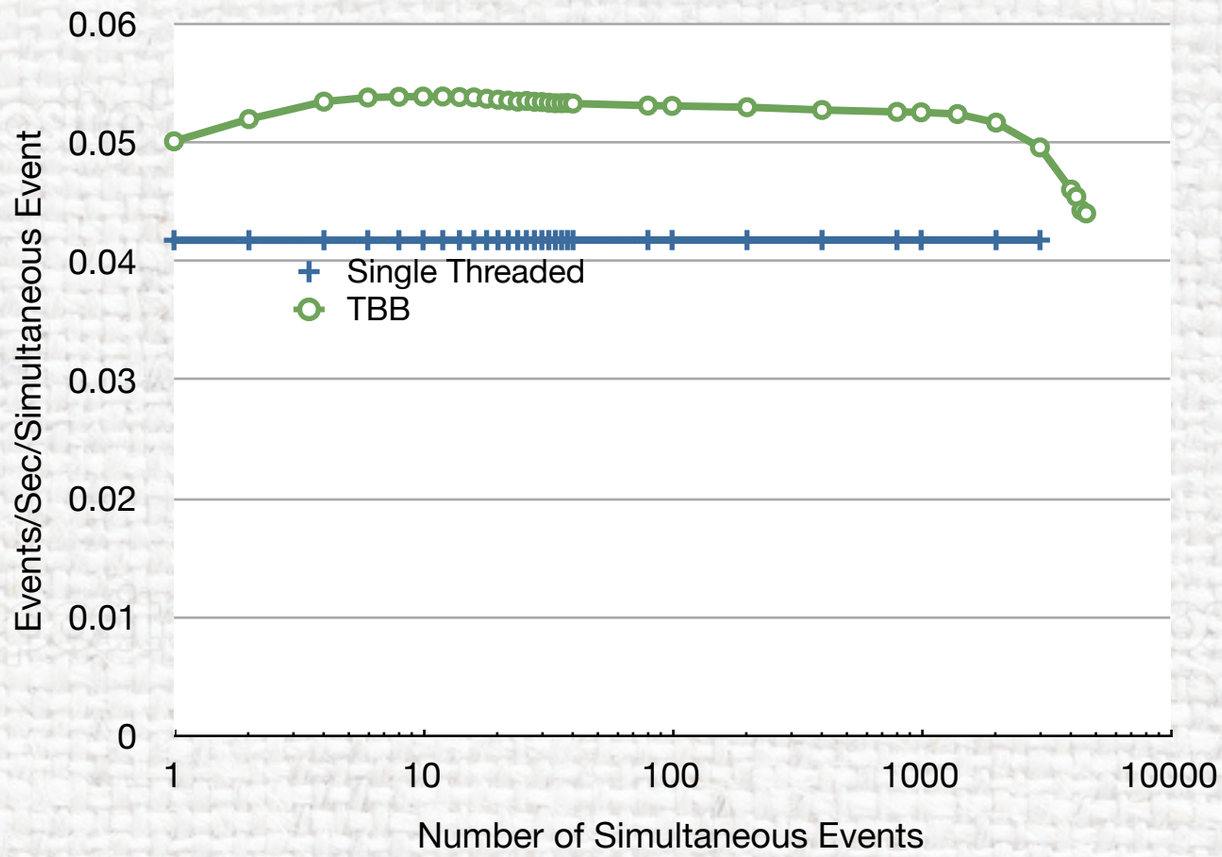
Can create own tasks for complex algorithms

```
class MyTask : public tbb::task { ... };
...
MyTask* mt = new (tbb::task::allocate_root()) MyTask;
tbb::task::spawn_root_and_wait( mt );
```

Users tasks must finish before returning from module

Scaling: Infinite Cores





32 core AMD Opteron Processor 6128 w/ 64GB RAM

All modules are calling usleep

TBB stops perfect scaling around 2000 simultaneous events (se)

Is using 1.3 threads/simultaneous event

Single threaded framework hits memory limit at 3000 se

Thread-Safety

Thread Safety

Data Products

Information passed from module to module

Framework only provides 'const' access to data products

'const' member functions must be thread safe

Matches C++11 thread-safety guarantee for containers

Modules

Majority of user defined code

Different module varieties define different levels of thread safety

- Stream
- Global
- One
- Legacy

Stream Module

Replicate an instance of a module configuration for each Stream

E.g. if have 8 Streams in a job will have 8 copies of a module

A Stream only processes one Event at a time

A module copy will only be called at most once per event

Member data does not have to be thread safe

One Stream only sees a fraction of the Events in the job

Therefore a module copy only sees a fraction of the events

Not a problem for most Producers and Filters

Easy to convert from Legacy to Stream interface

```
class TrackClusterRemover : public stream::Producer<> {
…};
```

Most Filters and Producers should be easy to convert

Global Module

One instance of a module shared by all Streams

One module sees all Runs, LuminosityBlocks and Events

All member functions and member data must be thread-safe

Member functions called on each transition are 'const'

The interface provides ways to help you with thread-safety

per transition caching

```
class Counter : public global::Analyzer<StreamCache<int>> {
...
   void analyze(StreamID id, Event const& event) const {
     ++(*streamCache(id) ); }
};
```

Only use if

Need to share as much memory across Streams as possible or

Algorithm must see all Runs, LuminosityBlocks or Events

High performance OutputModules would be Global

Global Module

One instance of a module shared by all Streams

One module sees all Runs, LuminosityBlocks and Events

All member functions and member data must be thread-safe

Member functions called on each transition are 'const'

The interface provides ways to help you with thread-safety

per transition caching

```
class Counter : public global::Analyzer<StreamCache<int>> {
...
   void analyze(StreamID id, Event const& event) const {
     ++(*streamCache(id) ); }
};
```

Only use if

Need to share as much memory across Streams as possible or

Algorithm must see all Runs, LuminosityBlocks or Events

High performance OutputModules would be Global

Global Module

One instance of a module shared by all Streams

One module sees all Runs, LuminosityBlocks and Events

All member functions and member data must be thread-safe

Member functions called on each transition are 'const'

The interface provides ways to help you with thread-safety

per transition caching

```
class Counter : public global::Analyzer<StreamCache<int>> {
...
   void analyze(StreamID id, Event const& event) const {
     ++(*streamCache(id) ); }
};
```

Only use if

Need to share as much memory across Streams as possible or

Algorithm must see all Runs, LuminosityBlocks or Events

High performance OutputModules would be Global

One Module

One instance of a module shared by all Streams

One module sees all transitions

Module instance sees only one transition at a time

Framework guarantees the serialization

Member data does not need to be thread-safe

Can use a resource shared across different modules

Modules declare the use of the resource

Framework guarantees only one module using the resource runs at a time

Can call code which uses 'static'

E.g. legacy FORTRAN based MC event generators

Easy to convert from Legacy to One interface

```
class NTupleMaker : public one::Analyzer<> {
…};
```

Good for OutputModules and ntuple making Analyzers

Legacy Module

Modules which have not been ported to new interface

Just need to recompile

Only one legacy module will run at a time

Have to assume the modules can interfere with one another

Performance problem

Eases code migration

Thread Safe Coding

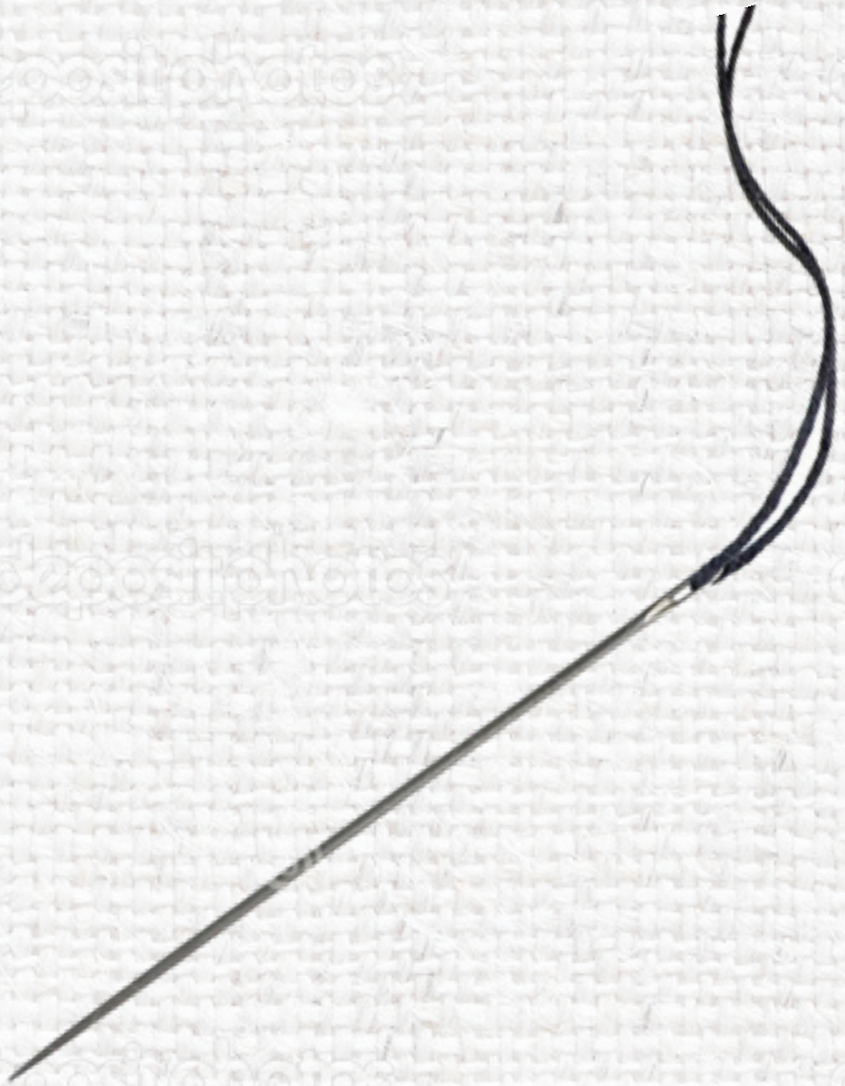
CMS module users that wish to implement fine grain parallelism must keep in mind that their own internal data structures and algorithms have to be thread safe.

The framework group has identified three patterns that can be reused:

Use of C++11 std::atomic to guarantee synchronization between threads.
Thread safe Pointer Caches
Thread safe Value Caches

See the backup slides for how these concepts can be implemented.

Tools



Tool Categories

Static code analysis

Clang

Run time checking

Helgrind

Static Code Analysis

CMS extended clang static analysis tool

http://clang-analyzer.llvm.org

Types of Checkers

Problem with const member functions of data products

Finding statics that affect modules

Data Products Checking

Data Products are shared between modules

Only const access is allowed

We check for

Non-const statics
Mutable member data which is not std::atomic<>
Member functions casting away const on member data
Pointer member data being returned from const function
Pointer member data being passed as non-const argument to function
includes calling a non-const member function of the pointed to class

Checks done recursively on all data members which are classes

Modules & Statics

Any non-const static used by a module is shared state

Working on tool which

Finds which functions in system interact with statics

For each function in system, determine which other functions they call

For a given module, see if any functions it calls ultimately reach a static

Helgrind

Tool in Valgrind suite

Searches for data races between threads

Records memory reads/writes done by each thread

Flags if multiple threads use same memory address and one does a write

Ignores cases where posix synchronization mechanism protects memory

mutex, semaphore, pthread_join

```
Possible data race during write of size 1 at 0x8D878A0 by thread #7
Locks held: none
   at 0x8E7B62C: MessageLogger::establishModule(...) (in libFWCoreMessageService.so)
   ...
   by 0x49CF6E9: EventProcessor::processEvent(unsigned int) (in libFWCoreFramework.so)

This conflicts with a previous write of size 1 by thread #2
Locks held: none
   at 0x8E7B62C: MessageLogger::establishModule(...) (in libFWCoreMessageService.so)
   ...
   by 0x49CF6E9: EventProcessor::processEvent(unsigned int) (in libFWCoreFramework.so)

Address 0x8D878A0 is 144 bytes inside a block of size 152 alloc'd
   at 0x4807A85: operator new(unsigned long) (in vgpreload_helgrind-amd64-linux.so)
   ...
```

Does not understand lock-free designs

Generates lots of false positives

Helgrind

Tool in Valgrind suite

Searches for data races between threads

Records memory reads/writes done by each thread

Flags if multiple threads use same memory address and one does a write

Ignores cases where posix synchronization mechanism protects memory

mutex, semaphore, pthread_join

```
Possible data race during write of size 1 at 0x8D878A0 by thread #7
Locks held: none
   at 0x8E7B62C: MessageLogger::establishModule(...) (in libFWCoreMessageService.so)
   ...
   by 0x49CF6E9: EventProcessor::processEvent(unsigned int) (in libFWCoreFramework.so)

This conflicts with a previous write of size 1 by thread #2
Locks held: none
   at 0x8E7B62C: MessageLogger::establishModule(...) (in libFWCoreMessageService.so)
   ...
   by 0x49CF6E9: EventProcessor::processEvent(unsigned int) (in libFWCoreFramework.so)

Address 0x8D878A0 is 144 bytes inside a block of size 152 alloc'd
   at 0x4807A85: operator new(unsigned long) (in vgpreload_helgrind-amd64-linux.so)
   ...
```

Does not understand lock-free designs

Generates lots of false positives

Helgrind

Tool in Valgrind suite

Searches for data races between threads

Records memory reads/writes done by each thread

Flags if multiple threads use same memory address and one does a write

Ignores cases where posix synchronization mechanism protects memory

mutex, semaphore, pthread_join

```
Possible data race during write of size 1 at 0x8D878A0 by thread #7
Locks held: none
   at 0x8E7B62C: MessageLogger::establishModule(...) (in libFWCoreMessageService.so)
   ...
   by 0x49CF6E9: EventProcessor::processEvent(unsigned int) (in libFWCoreFramework.so)

This conflicts with a previous write of size 1 by thread #2
Locks held: none
   at 0x8E7B62C: MessageLogger::establishModule(...) (in libFWCoreMessageService.so)
   ...
   by 0x49CF6E9: EventProcessor::processEvent(unsigned int) (in libFWCoreFramework.so)

Address 0x8D878A0 is 144 bytes inside a block of size 152 alloc'd
   at 0x4807A85: operator new(unsigned long) (in vgpreload_helgrind-amd64-linux.so)
   ...
```

Does not understand lock-free designs

Generates lots of false positives

Helgrind

Tool in Valgrind suite

Searches for data races between threads

Records memory reads/writes done by each thread

Flags if multiple threads use same memory address and one does a write

Ignores cases where posix synchronization mechanism protects memory

mutex, semaphore, pthread_join

```
Possible data race during write of size 1 at 0x8D878A0 by thread #7
Locks held: none
   at 0x8E7B62C: MessageLogger::establishModule(...) (in libFWCoreMessageService.so)
   ...
   by 0x49CF6E9: EventProcessor::processEvent(unsigned int) (in libFWCoreFramework.so)

This conflicts with a previous write of size 1 by thread #2
Locks held: none
   at 0x8E7B62C: MessageLogger::establishModule(...) (in libFWCoreMessageService.so)
   ...
   by 0x49CF6E9: EventProcessor::processEvent(unsigned int) (in libFWCoreFramework.so)

Address 0x8D878A0 is 144 bytes inside a block of size 152 alloc'd
   at 0x4807A85: operator new(unsigned long) (in vgpreload_helgrind-amd64-linux.so)
   ...
```

Does not understand lock-free designs

Generates lots of false positives

Conclusion

CMS is in the process of moving to a multi-threaded framework

The design allows many different levels of concurrency

Events, modules and sub-module

Thread-unsafe code is allowed via 'One' module variety

Framework guarantees serialization

Need tools to find thread-safety issues

Next CHEP we will have exciting results to report

Backup

Thread Safe Coding Patterns

C++11 std::atomic

C++11 defines a threading memory model

Access to the same memory location by multiple threads involving at least one write is not defined unless explicitly synchronized

std::atomic<> defines an explicit synchronization

Initialization	**Thread 1**	**Thread 2**
`int a=0, b=0;`		
`std::atomic<bool> isSet{false};`		
	`a=2; b=3;`	`while(not isSet.load()){};`
	`isSet.store(true);`	
		`cout << a <<' '<< b << endl;`
Output	2 3	

C++11 std::atomic

C++11 defines a threading memory model

Access to the same memory location by multiple threads involving at least one write is not defined unless explicitly synchronized

std::atomic<> defines an explicit synchronization

Initialization	**Thread 1**	**Thread 2**
`int a=0, b=0;`		
`std::atomic<bool> isSet{false};`		
	`a=2; b=3;`	`while(not isSet.load()){};`
	`isSet.store(true);`	
		`cout << a <<' '<< b << endl;`
Output	2 3	

C++11 std::atomic

C++11 defines a threading memory model

Access to the same memory location by multiple threads involving at least one write is not defined unless explicitly synchronized

std::atomic<> defines an explicit synchronization

Initialization	**Thread 1**	**Thread 2**
`int a=0, b=0;`		
`std::atomic<bool> isSet{false};`		
	`a=2; b=3;`	`while(not isSet.load()){};`
	`isSet.store(true);`	
		`cout << a <<' '<< b << endl;`
Output	2 3	

C++11 std::atomic

C++11 defines a threading memory model

Access to the same memory location by multiple threads involving at least one write is not defined unless explicitly synchronized

std::atomic<> defines an explicit synchronization

Initialization	**Thread 1**	**Thread 2**
`int a=0, b=0;`		
`std::atomic<bool> isSet{false};`		
	`a=2; b=3;`	`while(not isSet.load()){};`
	`isSet.store(true);`	
		`cout << a <<' '<< b << endl;`
Output	2 3	

C++11 std::atomic

C++11 defines a threading memory model

Access to the same memory location by multiple threads involving at least one write is not defined unless explicitly synchronized

std::atomic<> defines an explicit synchronization

Initialization	**Thread 1**	**Thread 2**
`int a=0, b=0;`		
`std::atomic<bool> isSet{false};`		
	`a=2; b=3;`	`while(not isSet.load()){};`
	`isSet.store(true);`	
		`cout << a <<' '<< b << endl;`
Output	2 3	

Pointer Caches

Caching is often used in const member functions

Cache needs to be updated in a thread safe manner

```
class Blah {
   ...
   mutable std::atomic<Foo*> m_foo;
};

const Foo& Blah::foo() const {
   if(nullptr==m_foo.load()) {
      std::unique_ptr<Foo> f{ new Foo(...) }; //make value to cache

      //see if we should keep our instance
      Foo* expected = nullptr;
      if(m_foo.compare_exchange_strong(expected, f.get()) ) {
         //m_foo was equal to nullptr and now is equal to f.get()
         f.release();
      }
   }
   return *m_foo;
}
```

Pointer Caches

Caching is often used in const member functions

Cache needs to be updated in a thread safe manner

```
class Blah {
   ...
   mutable std::atomic<Foo*> m_foo;
};

const Foo& Blah::foo() const {
   if(nullptr==m_foo.load()) {
      std::unique_ptr<Foo> f{ new Foo(...) }; //make value to cache

      //see if we should keep our instance
      Foo* expected = nullptr;
      if(m_foo.compare_exchange_strong(expected, f.get()) ) {
         //m_foo was equal to nullptr and now is equal to f.get()
         f.release();
      }
   }
   return *m_foo;
}
```

Pointer Caches

Caching is often used in const member functions

Cache needs to be updated in a thread safe manner

```
class Blah {
   ...
   mutable std::atomic<Foo*> m_foo;
};

const Foo& Blah::foo() const {
   if(nullptr==m_foo.load()) {
      std::unique_ptr<Foo> f{ new Foo(...) }; //make value to cache

      //see if we should keep our instance
      Foo* expected = nullptr;
      if(m_foo.compare_exchange_strong(expected, f.get()) ) {
         //m_foo was equal to nullptr and now is equal to f.get()
         f.release();
      }
   }
   return *m_foo;
}
```

Pointer Caches

Caching is often used in const member functions

Cache needs to be updated in a thread safe manner

```
class Blah {
   ...
   mutable std::atomic<Foo*> m_foo;
};

const Foo& Blah::foo() const {
   if(nullptr==m_foo.load()) {
      std::unique_ptr<Foo> f{ new Foo(...) }; //make value to cache

      //see if we should keep our instance
      Foo* expected = nullptr;
      if(m_foo.compare_exchange_strong(expected, f.get()) ) {
         //m_foo was equal to nullptr and now is equal to f.get()
         f.release();
      }
   }
   return *m_foo;
}
```

Pointer Caches

Caching is often used in const member functions

Cache needs to be updated in a thread safe manner

```
class Blah {
   ...
   mutable std::atomic<Foo*> m_foo;
};

const Foo& Blah::foo() const {
   if(nullptr==m_foo.load()) {
      std::unique_ptr<Foo> f{ new Foo(...) }; //make value to cache

      //see if we should keep our instance
      Foo* expected = nullptr;
      if(m_foo.compare_exchange_strong(expected, f.get()) ) {
         //m_foo was equal to nullptr and now is equal to f.get()
         f.release();
      }
   }
   return *m_foo;
}
```

Pointer Caches

Caching is often used in const member functions

Cache needs to be updated in a thread safe manner

```
class Blah {
   ...
   mutable std::atomic<Foo*> m_foo;
};

const Foo& Blah::foo() const {
   if(nullptr==m_foo.load()) {
      std::unique_ptr<Foo> f{ new Foo(...) }; //make value to cache

      //see if we should keep our instance
      Foo* expected = nullptr;
      if(m_foo.compare_exchange_strong(expected, f.get()) ) {
         //m_foo was equal to nullptr and now is equal to f.get()
         f.release();
      }
   }
   return *m_foo;
}
```

Value Caches

```
class Blah {
   ...
   mutable Foo m_foo;
   enum FooStates {kUnset, kSetting, kSet};
   mutable std::atomic<char> m_fooState = kUnset;
};

Foo Blah::foo() const {
   if(kSet==m_fooState.load()) return m_foo;

   Foo tmp{...}; //need to make one

   //Try to cache
   char expected = kUnset;
   if(m_fooState.compare_exchange_strong(expected, kSetting) ) {
     //it is our job to set the value
     m_foo.swap(tmp);

     //this must be after the swap
     m_fooState.store(kSet);
     return m_foo;
   }
   //another thread beat us to trying to set m_foo
   return tmp;
}
```

Value Caches

```
class Blah {
   ...
   mutable Foo m_foo;
   enum FooStates {kUnset, kSetting, kSet};
   std::atomic<char> m_fooState = kUnset;
};

Foo Blah::foo() const {
   if(kSet==m_fooState.load()) return m_foo;

   Foo tmp{...}; //need to make one

   //Try to cache
   char expected = kUnset;
   if(m_fooState.compare_exchange_strong(expected, kSetting) ) {
     //it is our job to set the value
     m_foo.swap(tmp);

     //this must be after the swap
     m_fooState.store(kSet);
     return m_foo;
   }
   //another thread beat us to trying to set m_foo
   return tmp;
}
```

Value Caches

```
class Blah {
   ...
   mutable Foo m_foo;
   enum FooStates {kUnset, kSetting, kSet};
   mutable std::atomic<char> m_fooState = kUnset;
};

Foo Blah::foo() const {
   if(kSet==m_fooState.load()) return m_foo;

   Foo tmp{...}; //need to make one

   //Try to cache
   char expected = kUnset;
   if(m_fooState.compare_exchange_strong(expected, kSetting) ) {
     //it is our job to set the value
     m_foo.swap(tmp);

     //this must be after the swap
     m_fooState.store(kSet);
     return m_foo;
   }
   //another thread beat us to trying to set m_foo
   return tmp;
}
```

Value Caches

```
class Blah {
   ...
   mutable Foo m_foo;
   enum FooStates {kUnset, kSetting, kSet};
   mutable std::atomic<char> m_fooState = kUnset;
};

Foo Blah::foo() const {
   if(kSet==m_fooState.load()) return m_foo;

   Foo tmp{...}; //need to make one

   //Try to cache
   char expected = kUnset;
   if(m_fooState.compare_exchange_strong(expected, kSetting) ) {
     //it is our job to set the value
     m_foo.swap(tmp);

     //this must be after the swap
     m_fooState.store(kSet);
     return m_foo;
   }
   //another thread beat us to trying to set m_foo
   return tmp;
}
```

Value Caches

```
class Blah {
   ...
   mutable Foo m_foo;
   enum FooStates {kUnset, kSetting, kSet};
   mutable std::atomic<char> m_fooState = kUnset;
};

Foo Blah::foo() const {
   if(kSet==m_fooState.load()) return m_foo;

   Foo tmp{...}; //need to make one

   //Try to cache
   char expected = kUnset;
   if(m_fooState.compare_exchange_strong(expected, kSetting) ) {
     //it is our job to set the value
     m_foo.swap(tmp);

     //this must be after the swap
     m_fooState.store(kSet);
     return m_foo;
   }
   //another thread beat us to trying to set m_foo
   return tmp;
}
```

Value Caches

```
class Blah {
   ...
   mutable Foo m_foo;
   enum FooStates {kUnset, kSetting, kSet};
   mutable std::atomic<char> m_fooState = kUnset;
};

Foo Blah::foo() const {
   if(kSet==m_fooState.load()) return m_foo;

   Foo tmp{...}; //need to make one

   //Try to cache
   char expected = kUnset;
   if(m_fooState.compare_exchange_strong(expected, kSetting) ) {
     //it is our job to set the value
     m_foo.swap(tmp);

     //this must be after the swap
     m_fooState.store(kSet);
     return m_foo;
   }
   //another thread beat us to trying to set m_foo
   return tmp;
}
```

Value Caches

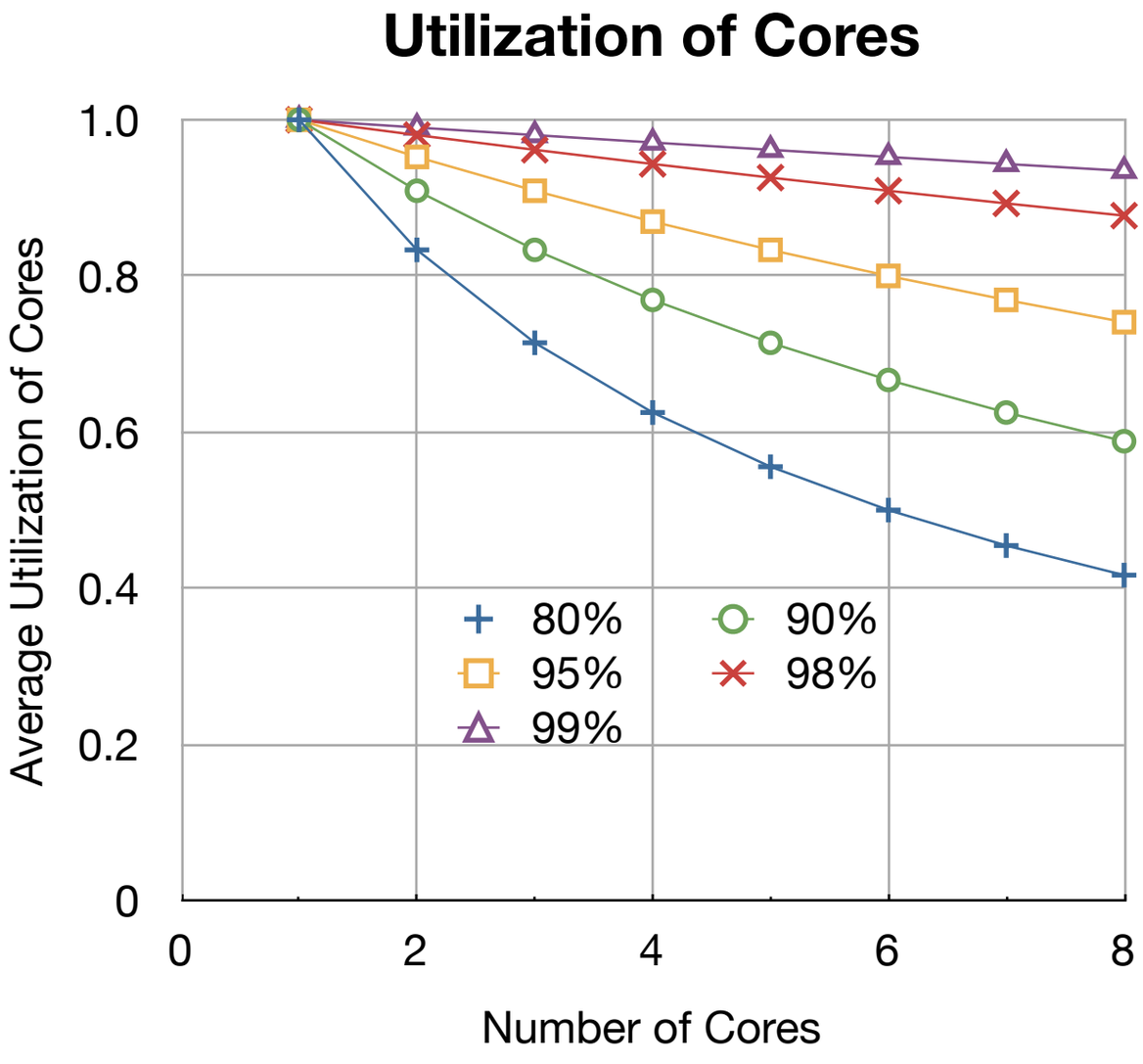
```
class Blah {
   ...
   mutable Foo m_foo;
   enum FooStates {kUnset, kSetting, kSet};
   mutable std::atomic<char> m_fooState = kUnset;
};

Foo Blah::foo() const {
   if(kSet==m_fooState.load()) return m_foo;

   Foo tmp{...}; //need to make one

   //Try to cache
   char expected = kUnset;
   if(m_fooState.compare_exchange_strong(expected, kSetting) ) {
     //it is our job to set the value
     m_foo.swap(tmp);

     //this must be after the swap
     m_fooState.store(kSet);
     return m_foo;
   }
   //another thread beat us to trying to set m_foo
   return tmp;
}
```

Value Caches

```
class Blah {
   ...
   mutable Foo m_foo;
   enum FooStates {kUnset, kSetting, kSet};
   mutable std::atomic<char> m_fooState = kUnset;
};

Foo Blah::foo() const {
   if(kSet==m_fooState.load()) return m_foo;

   Foo tmp{...}; //need to make one

   //Try to cache
   char expected = kUnset;
   if(m_fooState.compare_exchange_strong(expected, kSetting) ) {
     //it is our job to set the value
     m_foo.swap(tmp);

     //this must be after the swap
     m_fooState.store(kSet);
     return m_foo;
   }
   //another thread beat us to trying to set m_foo
   return tmp;
}
```

Amdahl's Law

Speedup of parallelization is limited by the sequential parts of a program



Parallel Fraction	4 Cores	8 Cores
0.8	0.63	0.42
0.9	0.77	0.59
0.95	0.87	0.74
0.984	0.95	0.90
0.992	0.98	0.95

Can not make good use of multiple cores till vast majority of CMSSW code can run threaded

Gustafson's Law

The larger the problem the better it parallelizes

Can solve larger problems in the same amount of