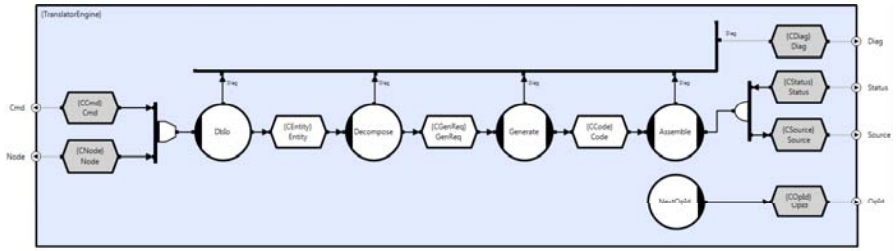
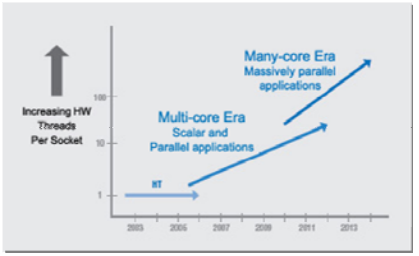
An Object Oriented Programming Technology for Multi-Core Architectures



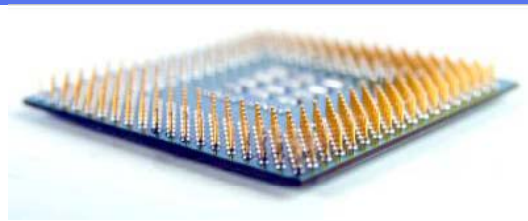
The Relevance of Multi-Core

- **The extraordinary 20 year 'free lunch' is over; CPU speeds have peaked**
- **The only way for the industry to advance is to exploit multi-core**
- **History shows that the more performance you have, the more performance you need**
- **Parallel programming isn't new, but developing for a target with a core count that doubles every 18 months IS**

"Humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code"
Herb Sutter and James Larus - Microsoft



The Relevance of Multi-Core



- **Software MUST automatically scale or it will become obsolete**
- **The target hardware at the end of a 3 year development cycle will be 4 times as powerful as the hardware at project outset**
- **Worse still, the first wave of devices capitalize on Symmetric Shared Memory, but this is not sustainable past 16 cores and the next generation (within 5 years) will almost certainly have non-uniform memory**
- **Applications written today that assume SMP may well need considerable re-work in as little as 5 years time.**

The Relevance of Multi-Core

- **Distributed applications are a particular problem because you may start with 4 x 8-core machines and end up with 2 x 16-core machines therefore topology must not be assumed**
- **An OpenMP/MPI hybrid approach could easily assume a particular topology**
- **Concurrent programming is extremely specialized and there is a serious skills shortage**
- **Up until now parallel programming has been a niche activity that has focused on relatively simple data parallel problems**
- **The 'General Case' is orders of magnitude more difficult to address**

Current Technology Status

- **Solution Complexity**
  - How difficult is it to understand, develop and maintain
- **Core Count Dependence**
  - Does it scale automatically?
- **Memory Architecture Dependence**
  - Does it require specific implementations for specific memory models?
- **Topology Dependence**
  - Is the application isolated from network topology changes?

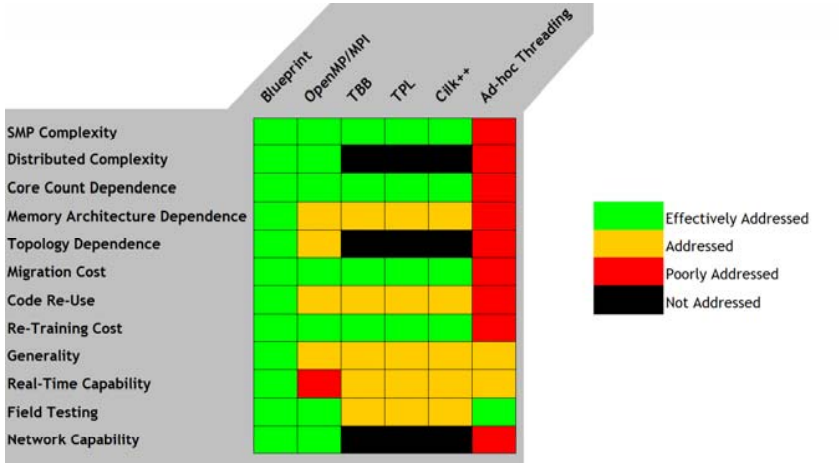
Current Technology Status

- **Migration Cost**
  - How much does it cost to migrate legacy applications?
- **Code Re-use**
  - How easy is it to create re-usable components?
- **Re-training Cost**
  - What is the cost of re-training engineers?
- **Application Niches**
  - Does it address general purpose irregular applications?

Current Technology Status

- **Real-time Capability**
  - Can it be used with real-time applications?
- **Field Testing**
  - Has the technology been validated in the field?
- **Network Capability**
  - Are networks of computers supported?
- **Compose-ability**
  - Can components be arbitrarily re-used (the locking problem)?

Technology Comparison (Subjective)



Why Is Concurrency A Problem?

- Humans deal well with concurrency on a day to day basis.
- Soccer involves keeping track of 22 players, the referee and linesmen, the ball and more. It's natural and effortless.
- Now try to listen to 2 sentences or read two sentences simultaneously.
- Watching football on TV is a far richer experience than listening to it on the radio.
- Words need order and our brains process text sequentially
- Pictures can be scanned in any order and persist in our brains
- Concurrency should be seen and not heard!

"Humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code"

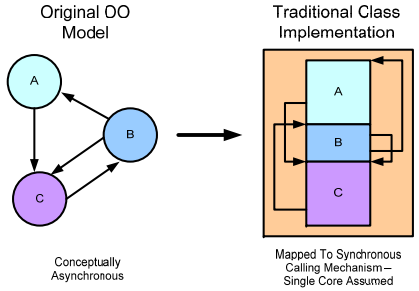
Herb Sutter and James Larus

Concurrency and OO Concepts

- The OO programming model is generally agreed to be one of the most successful paradigms to emerge in the last three decades
- As originally conceived it's inherently concurrent and at its highest level of abstraction makes no assumption about the underlying platform
- So why isn't multi-core seen as enabling the OO abstraction?
- And why hasn't the 'Actor' approach had more impact?

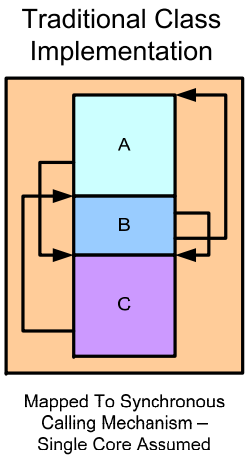
Why Is Concurrent OO a Problem?

- Implementing Concurrent models is difficult
  - Additional synchronization and scheduling logic is required
  - This is assumed, and so not prescribed, by typical models



How Sequential Implementations Avoid The Problems

- By constraining to a sequential solution, this additional synchronization information is not required:
  - Synchronous function calling can replace the more complicated asynchronous invocation idealized in the original OO model
  - The 'stack' can take care of data lifetimes in a simple and intuitive manner
- But a sequential solution does not make use of multiple processing cores

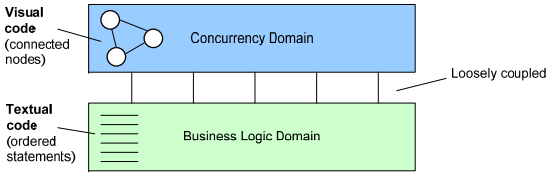


Principal Goals

- Present developers with OO's familiar and intuitive view of concurrently executing objects
- Explicitly allow them to express their synchronization and scheduling logic with a similarly high level of abstraction
- Do so in a manner that makes no assumptions about the target platform's architecture and/or memory topology
- Capitalize on OO concepts such as encapsulation and inheritance (modularity, component re-use etc)
- Re-use existing code with minimal modification
- Specialize concurrency and alleviate the skills shortage

Separation Of Concerns

- Processing and scheduling are different, separable concerns requiring a hybrid approach
  - Leave algorithmic and business logic in it's current sequential form
  - Describe application's concurrency in terms of it's connectivity and dependency



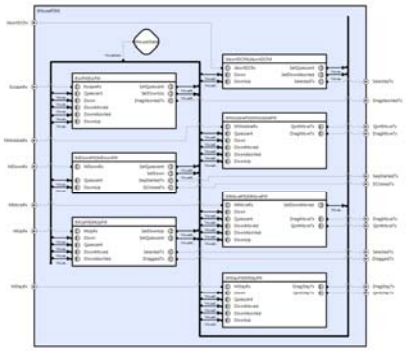
- This means:
  - Existing applications are largely unaffected by multi-core migration
  - Most developers continue to work in a familiar sequential environment using familiar tools and languages

Abstracting The Platform

- Concurrency must be expressed in a manner that does not make any assumptions about the target platform
  - Number of cores
  - Number of machines
  - Memory distribution
- Programmers must be presented with a simple and intuitive 'idealized platform'
- Mapping functionality to target hardware must be another separate stage that should not involve or concern application developers

Principal Components

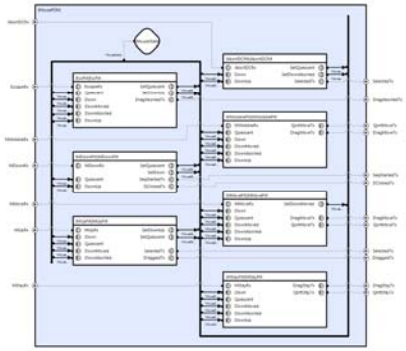
- CDL
  - Domain specific Visual Programming Language (VPL)
  - Alternative to threading and messaging
  - Event based and parallel at the top level
  - Conventional sequential at the lower level
  - Most code is unchanged
  - Hybrid (GUI designer) approach



Connective Logic

Principal Components

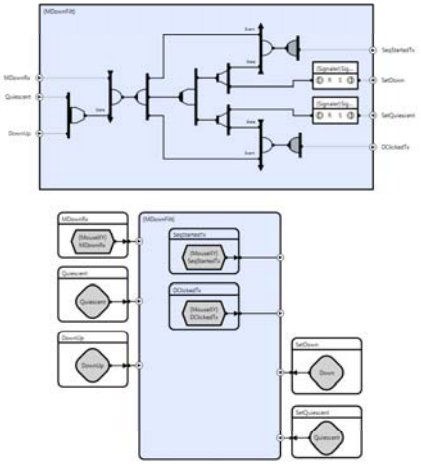
- **Language Runtime**
  - Provides text based API
  - Called by generated and user code
  - Schedules and synchronizes CDL programs
- **Development Environment**
  - Graphical CDL editor
  - Text editor
  - CDL to C++ Translator



Connective Logic

General Approach

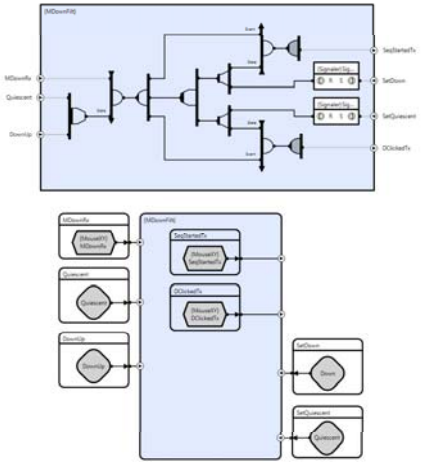
- **Domain Separation**
  - Infrastructure
  - Processing
  - Data definition
  - Device I/O
  - Man/Machine Interfacing



Connective Logic

General Approach

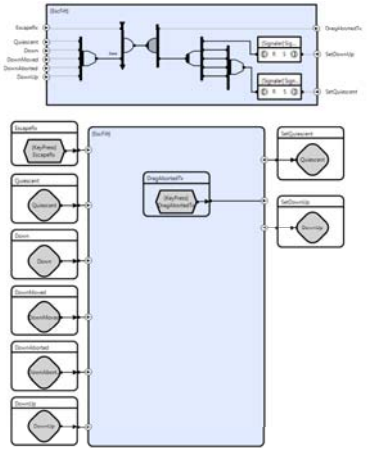
- **Single Virtual Process Model**
  - Logical model doesn't assume physical topology
  - No distinction between inter-thread and inter-process communication
  - Separate Accretion
- **Parallel Execution at Top Level**
  - Asynchronously executing objects
  - Arguments replaced by events
  - Synchronized by event exchange



Connective Logic

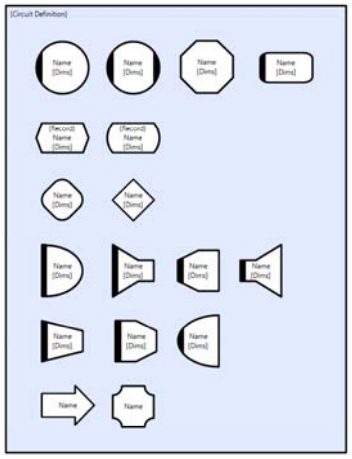
The CDL (Infrastructure) Domain

- **Parallelism**
  - Describes the application's inherent concurrency
- **Sequencing**
  - Specifies parallel execution order
- **Event Flow Management**
  - Provides distribution, collection, multiplexing etc
- **Exclusion**
  - Controls shared data access
- **Data Management**
  - Construction/Destruction of data objects
- **Interfacing**
  - Connection Prototyping



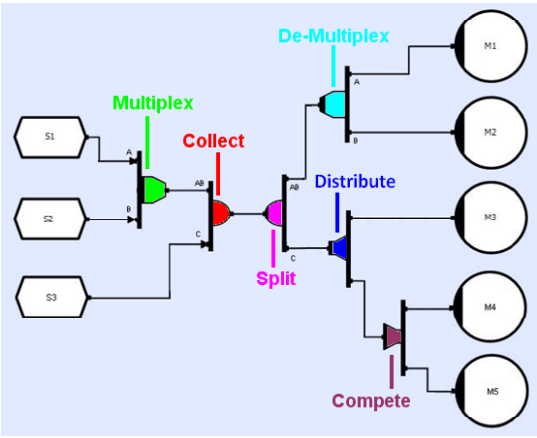
CDL's Basic Components

- **Active Objects**
  - Contain algorithmic processing code
  - Analogous to GUI call-backs
  - Properties of lightweight threads
- **Stores**
  - Containers for data objects
- **Semaphores**
  - Synchronization objects
- **Event Propagators**
  - Coordinate the flow of events
  - DBX, CLX, MPX etc.
- **Devices**
  - I/O handlers & interfaces
- **Circuits**
  - Encapsulated compositions of the above
  - Analogous to C++ classes



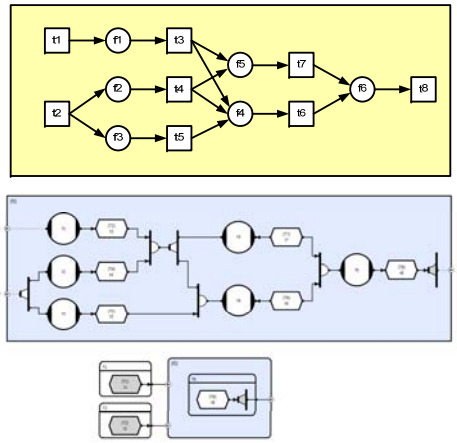
Event Operators

- **Merge**
  - Multiplex multiple events
  - Collect multiple events
- **Branch**
  - Split collected events
  - De-multiplex multiplexed events
  - Share events (Distribute)
  - Compete for events
- **Compose**
  - CDL operations are compose-able and can be re-used

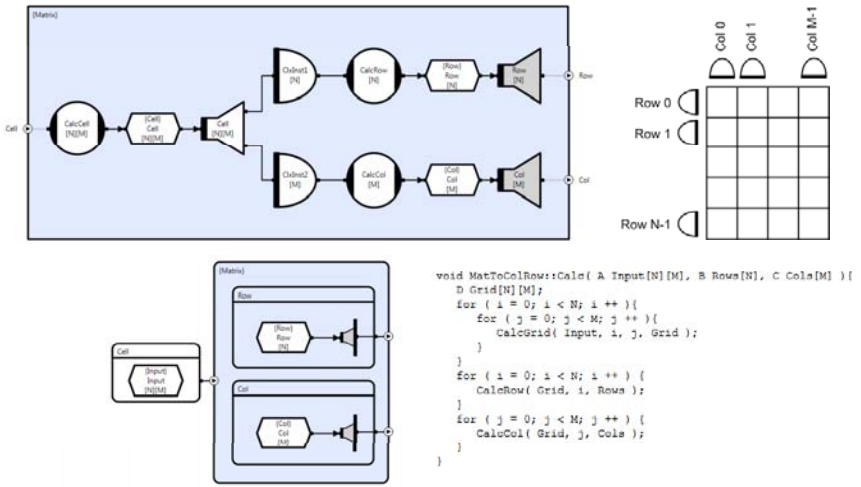


Simple Example

```
T8 f0( T1 t1, T2 t2 ) {

   T3 t3;
   T4 t4;
   T5 t5;
   T6 t6;
   T7 t7;
   T8 t8;

   t3 = f1( t1 );
   t4 = f2( t2 );
   t5 = f3( t2 );
   t6 = f4( t3, t4, t5 );
   t7 = f5( t3, t4 );
   t8 = f6( t6, t7 );

   return t8;
}
```

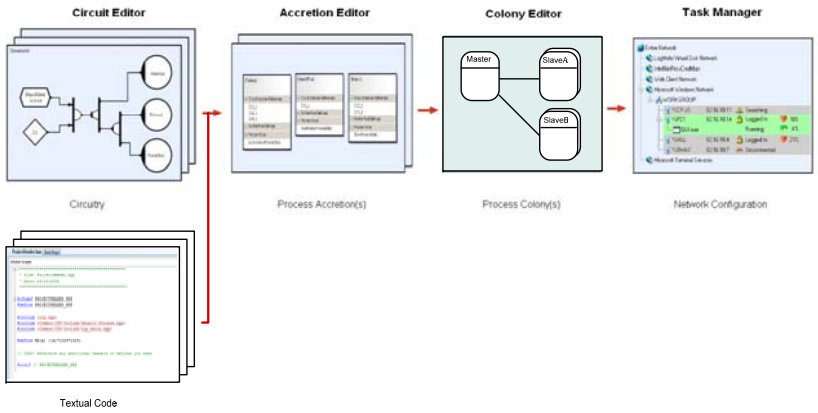


Data Parallel Example




```
void MatToColRow::Calc( A Input[N][M], B Rows[N], C Cols[M] ){
   D Grid[N][M];
   for ( i = 0; i < N; i ++ ){
      for ( j = 0; j < M; j ++ ){
         CalcGrid( Input, i, j, Grid );
      }
   }
   for ( i = 0; i < N; i ++ ) {
      CalcRow( Grid, i, Rows );
   }
   for ( j = 0; j < M; j ++ ) {
      CalcCol( Grid, j, Cols );
   }
}
```

Toolset



Project History

- **First prototype used for sidescan sonar (November 1993)**
- **First release of the current runtime used for 'Generic Sound Room' distributed interactive naval trainer; (April 1995)**
- **Various applications including Software Radio, Synthetic Aperture Sonar, UK's Surface Ship Torpedo Defense system, Australia's Air Warfare Destroyer Sonar, Radar Simulators, Finite Difference Time Domain solvers, Mine Hunting, Voice over IP demonstrator, Signals Intelligence demonstrator, Blueprint Toolset.**
- **Early adopter trials of Blueprint (May 2008)**
- **COTS Implementation (March 2009)**

SSTD Case Study

- 50+ Circuits
- 38K of 96K Generated Code
- Peak Concurrency 168
- <1% CPU Overhead
- SSTD APRO
  - Single Process (Windows)
  - DY4 MP 14 Power PCs (VxWorks)
  - Networked 32-bit Pentiums (Windows)
  - Dual Core 64-bit Opteron (Windows)
- SSTD HCI
  - Windows QT
  - Linux QT

Opportunities for Collaboration

- **Eclipse Editor Implementation**
- **Static Circuit Analysis**
  - **Deadly Embraces**
  - **Log Jams**
  - **Data Races**
  - **Priority Inversions**
- **Language Support**
  - **Java**
  - **Others**
- **Component Libraries**
  - **Sonar**
  - **Radar**
  - **Others**
- **Platform Drivers**