Better Code

- Regular Type
  - Goal: Implement Complete and Efficient Types
- Algorithms
  - Goal: No Raw Loops
- Data Structures
  - Goal: No Incidental Data Structures
- Runtime Polymorphism
  - Goal: No Raw Pointers
- Concurrency
  - Goal: No Raw Synchronization Primitives
- ...

Common Themes

- Manage Relationships
- Understand the Fundamentals
- Code Simply
Demo
Concurrency

- Concurrency: when tasks start, run, and complete in overlapping time periods
- Parallelism: when two or more tasks execute simultaneously

Why?
- Enable performance through parallelism
- Improve interactivity by handling user actions concurrent with processing and IO
No Raw Synchronization Primitives
What are raw synchronization primitives?

- Synchronization primitives are basic constructs such as:
  - Mutex
  - Atomic
  - Semaphore
  - Memory Fence
  - Condition Variable
You Will Likely Get It Wrong
template<typename T>
class bad_cow {
  struct object_t {
    explicit object_t(const T& x) : data_m(x) { ++count_m; }
    atomic<int> count_m;
    T data_m;
  };

  object_t* object_m;

public:
  explicit bad_cow(const T& x) : object_m(new object_t(x)) { }  
  ~bad_cow() { if (0 == --object_m->count_m) delete object_m; }
  bad_cow(const bad_cow& x) : object_m(x.object_m) { ++object_m->count_m; }

  bad_cow& operator=(const T& x) {
    if (object_m->count_m == 1) object_m->data_m = x;
    else {
      object_t* tmp = new object_t(x);
      --object_m->count_m;
      object_m = tmp;
    }
    return *this;
  }
};

• There is a subtle race condition here:
  • if count != 1 then the bad_cow could also is owned by another thread(s)
  • if the other thread(s) releases the bad_cow between these two atomic operations
  • then our count will fall to zero and we will leak the object
template<typename T>
class bad_cow {

  struct object_t {
    explicit object_t(const T& x) : data_m(x) { ++count_m; }
    atomic<int> count_m;
    T data_m; 
  }
  object_t* object_m;

public:
  explicit bad_cow(const T& x) : object_m(new object_t(x)) { }
  ~bad_cow() { if (0 == --object_m->count_m) delete object_m; }
  bad_cow(const bad_cow& x) : object_m(x.object_m) { ++object_m->count_m; }
  bad_cow& operator=(const T& x) {
    if (object_m->count_m == 1) object_m->data_m = x;
    else {
      object_t* tmp = new object_t(x);
      if (0 == --object_m->count_m) delete object_m;
      object_m = tmp;
    }
    return *this;
  }
};
Why do we want concurrency?

Performance through Parallelism
Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)

- OpenGL
- OpenCL
- CUDA
- Direct Compute
- C++ AMP
- DirectX
- Intrinsics
- Auto-vectorization
- OpenCL

- TBB
- GCD
- OpenMP
- C++11

- Straight C++

0 750 1500 2250 3000 (GFlops)

- Green: GPU
- Blue: Vectorization
- Purple: Multi-thread
- Red: Scalar

© 2014 Adobe Systems Incorporated. All Rights Reserved.
Amdahl’s Law

\[ S(N) = \frac{1}{(1 - P) + \frac{P}{N}} \]

http://en.wikipedia.org/wiki/Amdahl%27s_law
Amdahl’s Law
What Makes It Slow

- Starvation
- Latency
- Overhead
- Wait
Why No Raw Synchronization Primitives?
Minimize Locks
Threads and Tasks

- Thread: Execution environment consisting of a stack and processor state running in parallel to other threads
- Task: A unit of work, often a function, to be executed on a thread

- Tasks are scheduled on a thread pool to optimize machine utilization
C++14 and Tasks

- C++14 does not have a task system
  - Threads
  - Futures
Create Your Own Presentation

Build Task Stealing System

Futures
Building a Task System

- Portable Reference Implementation in C++14
  - Windows - Window Thread Pool and PPL
  - Apple - Grand Central Dispatch (libdispatch)
    - open source, runs on Linux and Android
- Intel TBB - many platforms
  - open source
- HPX - many platforms
  - open source
Building a Task System

![Diagram showing a task system with tasks branching into threads and cores.](http://docs.oracle.com/cd/E19253-01/816-5137/ggedn/index.html)
using lock_t = unique_lock<mutex>;

class notification_queue {
    deque<function<void()>> _q;
    mutex _mutex;
    condition_variable _ready;

public:
    void pop(function<void()>&& x) {
        lock_t lock{_mutex};
        while (_q.empty()) _ready.wait(lock);
        x = move(_q.front());
        _q.pop_front();
    }

    template<typename F>
    void push(F&& f) {
        lock_t lock{_mutex};
        _q.emplace_back(forward<F>(f));
        _ready.notify_one();
    }
};
class task_system {
    const unsigned _count{thread::hardware_concurrency()};
    vector<thread> _threads;
    notification_queue _q;

    void run(unsigned i) {
        while (true) {
            function<void()> f;
            _q.pop(f);
            f();
        }
    }

    public:
    task_system() {
        for (unsigned n = 0; n != _count; ++n) {
            _threads.emplace_back([& , n]{ run(n); });
        }
    }

    ~task_system() {
        for (auto& e : _threads) e.join();
    }

    template <typename F>
    void async_(F&& f) {
        _q.push(forward<F>(f));
    }
};
Building a Task System

class notification_queue {
  deque<function<void()>> _q;
  bool _done{false};
  mutex _mutex;
  condition_variable _ready;

public:
  void done() {
    unique_lock<mutex> lock{_mutex};
    _done = true;
  }

  bool pop(function<void()>& x) {
    lock_t lock{_mutex};
    while (!_q.empty() && !_done) _ready.wait(lock);
    if (_q.empty()) return false;
    x = move(_q.front());
    _q.pop_front();
    return true;
  }

  template<typename F>
  void push(F&& f) {
    lock_t lock{_mutex};
    _q.emplace_back(forward<F>(f));
  }
}
Building a Task System
Why No Raw Synchronization Primitives?
Building a Task System
Building a Task System

![Diagram of a task system with tasks, schedulers, threads, and cores]

- Task
- Scheduler
- Task
- Task
- Task
- Thread
- Thread
- Thread
- Core
- Core
- Core
- ...
Building a Task System

class task_system {
    const unsigned _count{thread::hardware_concurrency()};
    vector<thread> _threads;
    vector<notification_queue> _q{_count};
    atomic<unsigned> _index{0};

    void run(unsigned i) {
        while (true) {
            function<void()> f;
            if (!_q[i].pop(f)) break;
            f();
        }
    }

public:
    task_system() { }

    ~task_system() {
        for (auto& e : _q) e.done();
        for (auto& e : _threads) e.join();
    }

    template <typename F>
    void async_(F&& f) {
        auto i = _index++;
        _q[i % _count].push(forward<F>(f));
    }
};
Building a Task System

Thread

Task

Task

Task

Task

Task

Task

Task

Thread

Core

Core

Core

...
Building a Task System

Task

Scheduler

Task
Task
Task

Task
Task
Task

Task
Task
Task

Task Stealing

Thread
Thread
Thread

Core
Core

...
Building a Task System

class notification_queue {
    deque<function<void>>() _q;
    bool _done{false};
    mutex _mutex;
    condition_variable _ready;

public:
    bool try_pop(function<void>&& x) {
        lock_t lock{_mutex, try_to_lock};
        if (!lock || _q.empty()) return false;
        x = move(_q.front());
        _q.pop_front();
        return true;
    }

    template<typename F>
    bool try_push(F&& f) {
        lock_t lock{_mutex, try_to_lock};
        if (!lock) return false;
        _q.emplace_back(forward<F>(f));
        _ready.notify_one();
        return true;
    }

    void done() {
        lock {_mutex};
    }
}
void run(unsigned i) {
    while (true) {
        function<void()> f;
        for (unsigned n = 0; n != _count; ++n) {
            if (!_q[(i + n) % _count].try_pop(f)) break;
        }
        if (!f && !_q[i].pop(f)) break;
        f();
    }
}

public:
    task_system() {
    }

    ~task_system() {
    }

    template <typename F>
    void async_(F&& f) {
        auto i = _index++;
        for (unsigned n = 0; n != _count; ++n) {
            if (_q[(i + n) % _count].try_push(forward<F>(f))) return;
        }
        _q[i % _count].push(forward<F>(f));
    }
Building a Task System

Task

Scheduler

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task
Task System

- Within a few percentage points of Apple’s GCD (libdispatch) under load
  - Can be improved by spinning more on try_pop in run
No Raw Synchronization Primitives
Futures

future<cpp_int> x = async([] { return fibonacci<cpp_int>(1'000'000); });

// Do Something
cout << x.get() << endl;

- Fibonacci is often used as an example for parallel algorithms
  - Please stop...
template <typename T, typename N, typename O>
T power(T x, N n, O op)
{
    if (n == 0) return identity_element(op);

    while ((n & 1) == 0) {
        n >>= 1;
        x = op(x, x);
    }

    T result = x;
    n >>= 1;
    while (n != 0) {
        x = op(x, x);
        if ((n & 1) != 0) result = op(result, x);
        n >>= 1;
    }

    return result;
}

Egyptian Multiplication (Russian Peasant Algorithm)
See “From Mathematics to Generic Programming” - Alex Stepanov and Dan Rose
Public Service Announcement - How to Write Fibonacci

template <typename N>
struct multiply_2x2 {
    array<N, 4> operator()(const array<N, 4>& x, const array<N, 4>& y) {
        return { x[0] * y[0] + x[1] * y[2], x[0] * y[1] + x[1] * y[3],
    }
};

template <typename N>
array<N, 4> identity_element(const multiply_2x2<N>&) { return { N(1), N(0), N(0), N(1) }; }

template <typename R, typename N>
R fibonacci(N n) {
    if (n == 0) return R(0);
    return power(array<R, 4>{ 1, 1, 1, 0 }, N(n - 1), multiply_2x2<R>())[0];
}
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(1'000'000); });

// Do Something

cout << x.get() << endl;
\[ f(\ldots) \rightarrow r \]
Futures allow minimal code transformations to express dependencies
future<cpp_int> x = async([]{
    throw runtime_error("failure");
    return fibonacci<cpp_int>(1'000'000);
});

// Do Something

try {
    cout << x.get() << endl;
} catch (const runtime_error& error) {
    cout << error.what() << endl;
}
No Raw Synchronization Primitives
Futures: What year is this?

- C++14 futures lack:
  - Continuations - .then()
  - Joins - when_all()
  - Split
  - Cancelation
  - Progress Monitoring (Except Ready)

- And C++14 futures don't compose (easily) to add these features
Futures: Continuations
Futures: Continuations

- Blocking on std::future.get() has two problems
  - One thread resource is consumed, increasing contention
  - Any subsequent non-dependent calculations on the task are also blocked

- C++14 doesn't have continuations
  - GCD has serialized queues and groups
  - PPL has chained tasks
  - TBB has flow graphs
  - TS Concurrency will have them
  - Boost futures have them now
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(1'000); });
future<void> y = x.then([](future<cpp_int> x){ cout << x.get() << endl; });
// Do something
y.wait();

43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209
689623239873322471161642996440906533187938298969649928516003704476137795166849228875
Task Systems

Task

Result

Task

Result

Group

Task

*
Futures: Continuations

```cpp
auto x = async([]{ return fibonacci<cpp_int>(1'000'000); });
auto y = async([]{ return fibonacci<cpp_int>(2'000'000); });

auto z = when_all(std::move(x), std::move(y)).then([](auto f){
    auto t = f.get();
    return cpp_int(get<0>(t).get() * get<1>(t).get());
});

cout << z.get() << endl;
```

f is a future tuple of futures

result is 626,964 digits
Futures: Split
Futures: Continuations

```cpp
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(100); });

future<cpp_int> y = x.then([](future<cpp_int> x){ return cpp_int(x.get() * 2); });
future<cpp_int> z = x.then([](future<cpp_int> x){ return cpp_int(x.get() / 15); });
```

Assertion failed: (px != 0), function operator->, file shared_ptr.hpp, line 648.
Desired behavior

- A future should behave as a *regular* type - a token for the actual value
  - `shared_futures` let me pass "copy" them around and do multiple `get()` operations
  - But not multiple continuations
Continuations

- We can write a pseudo-copy, split().

![Diagram showing a continuation with a split node and corresponding arrows to other nodes.](image)
Futures: Continuations

```cpp
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(100); });
future<cpp_int> y = split(x).then([](future<cpp_int> x){ return cpp_int(x.get() * 2); });
future<cpp_int> z = x.then([](future<cpp_int> x){ return cpp_int(x.get() / 15); });
future<void> done = when_all(std::move(y), std::move(z)).then([](auto f){
    auto t = f.get();
    cout << get<0>(t).get() << endl;
    cout << get<1>(t).get() << endl;
});
done.wait();
```

708449696358523830150
23614989878617461005
Building Blocks

- Promise is the sending side of a future
- Promises are packaged with a function to formed a packaged task
  - Packaged tasks handle the exception marshalling through a promise
```cpp
promise<int> x;
future<int> y = x.get_future();

x.set_value(42);
cout << y.get() << endl;
```

42
template <typename T>
auto split(future<T>& x) {

    auto tmp = std::move(x);

    promise<T> p;
    x = p.get_future(); // replace x with new future

    return tmp.then([_p = move(p)](auto _tmp) mutable {
        auto value = _tmp.get();
        _p.set_value(value); // assign to new "x" future
        return value; // return value through future result
    });
}
Futures: Split

template <typename T>
auto split(future<T>& x) {

    auto tmp = std::move(x);

    promise<T> p;
    x = p.get_future(); // replace x with new future

    return tmp.then([_p = std::move(p)](auto _tmp) mutable {
        if (_tmp.has_exception()) {
            auto error = _tmp.get_exception_ptr();
            _p.set_exception(error);
            rethrow_exception(error);
        }

        auto value = _tmp.get();
        _p.set_value(value); // assign to new "x" future
        return value; // return value through future result
    });
}
Futures: Continuations

```cpp
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(100); });

future<cpp_int> y = split(x).then([](future<cpp_int> x){ return cpp_int(x.get() * 2); });
future<cpp_int> z = x.then([](future<cpp_int> x){ return cpp_int(x.get() / 15); });

future<void> done = when_all(std::move(y), std::move(z)).then([](auto f){
    auto t = f.get();
    cout << get<0>(t).get() << endl;
    cout << get<1>(t).get() << endl;
});

done.wait();
```

708449696358523830150
23614989878617461005
Cancelation

- When the (last) future destructs
  - The associated task that has not started, should not execute (NOP)
  - The resource held by that task should be released
    - Since that task may hold futures for other tasks, the system unravels

- I do not know of a good way to compose such cancelation with current futures
  - Except to create something more complex than re-implementing futures
Cancelation
shared
Futures
template <typename>
struct result_of_; //not defined

template <typename R, typename... Args>
struct result_of_<R(Args...)>
  { using type = R; };
template <typename> class packaged_task; //not defined

template <typename R>
class future {
    shared_ptr</* ... */> _p;

public:
    future() = default;

    template <typename F>
    auto then(F&& f) {
    }

    const R& get() const {
};

template<
typename R, typename ...Args >
class packaged_task<R (Args...)>{
    weak_ptr</* ... */> _p;

public:
    packaged_task() = default;

    template <typename... A>
    void operator()(A&&... args) const {
};
template <typename> class packaged_task; //not defined
template <typename> class future;

template <typename S, typename F>
auto package(F& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>
;

template <typename R>
class future {
    shared_ptr</* ... */> _p;

template <typename S, typename F>
friend auto package(F& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>
;
    explicit future(shared_ptr</* ... */> p) : _p(move(p)) { } /* ... */
};

template <typename R, typename ...Args>
class packaged_task<R (Args...)>
{ 
    weak_ptr</* ... */> _p;

template <typename S, typename F>
friend auto package(F& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>
;
    explicit packaged_task(weak_ptr</* ... */> p) : _p(move(p)) { } /* ... */
};
template <typename S, typename F>
auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>> {
    auto p = make_shared<shared<S>>(forward<F>(f));
    return make_pair(packaged_task<S>(p), future<result_of_t_<S>>(p));
}

package<int(double>>(f) -> { void(double), future<int> }
template <typename R>
struct shared_base {
    vector<R> _r; // optional
    mutex _mutex;
    condition_variable _ready;
    vector<function<void()>> _then;

    virtual ~shared_base() {} /* ... */
};

template <typename> struct shared; // not defined

template <typename R, typename... Args>
struct shared<R(Args...)> : shared_base<R> {
    function<R(Args...)> _f;

    template<typename F>
    shared(F&& f) : _f(forward<F>(f)) {} /* ... */
};
template<typename R, typename ...Args >
class packaged_task<R (Args...)>
{
    weak_ptr<shared<R(Args...)>> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit packaged_task(weak_ptr<shared<R(Args...)>> p) : _p(move(p)) { }

public:
    packaged_task() = default;

    template <typename... A>
    void operator()(A&&... args) const {
        auto p = _p.lock();
        if (p) (*p)(forward<A>(args)...);
    }
};
template <typename R, typename... Args>
struct shared<R(Args...)>
    : shared_base<R> {
    function<R(Args...)>& _f;

    template<typename F>
    shared(F&& f) : _f(forward<F>(f)) { }

    template<typename... A>
    void operator()(A&&... args) {
        this->set(_f(forward<A>(args)...));
        _f = nullptr;
    }
};
template <typename R>
struct shared_base {
    vector<R> _r; // optional
    mutex _mutex;
    condition_variable _ready;
    vector<function<void()>> _then;

    virtual ~shared_base() { }

    void set(R&& r) {
        vector<function<void()>> then;
        {
            lock_t lock(_mutex);
            _r.push_back(move(r));
            swap(_then, then);
        }
        _ready.notify_all();
        for (const auto& f : then) _system.async_(move(f));
    }
};
template <typename R>
class future {
  shared_ptr<shared_base<R>> _p;

template <typename S, typename F>
friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>

  explicit future(shared_ptr<shared_base<R>> p) : _p(move(p)) { }

public:
  future() = default;

  template <typename F>
  auto then(F&& f) {
    auto pack = package<result_of_t<F(R)>>((){[p = _p, f = forward<F>(f)]{
      return f(p->_r.back());
    });
    _p->_r.back().then(move(pack.first));
    return pack.second;
  }

  const R& get() const { return _p->get(); }
};
template <typename R>
struct shared_base {
    vector<R> _r; // optional
    mutex _mutex;
    condition_variable _ready;
    vector<function<void()>> _then;

    virtual ~shared_base() {}  

    void set(R&& r) {}  

    template <typename F>
    void then(F&& f) {
        bool resolved{false};
        {
            lock_t lock{_mutex};
            if (_r.empty()) _then.push_back(forward<F>(f));
            else resolved = true;
        }
        if (resolved) _system.async_(move(f));
    }  

    const R& get() {
        lock_t lock{_mutex};
        while (_r.empty()) _ready.wait(lock);
        return _r.back();
    }  
};
template<typename F, typename ...Args>
auto async(F&& f, Args&&... args)
{
    using result_type = result_of_t<F (Args...)>>;
    using packaged_type = packaged_task<result_type>();

    auto pack = package<result_type()>(bind(forward<F>(f), forward<Args>(args)...));

    thread(move(get<0>(pack))).detach(); // Replace with task queue
    return get<1>(pack);
}
Futures: Continuations

```cpp
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(100); });

future<cpp_int> y = x.then([](const cpp_int& x){ return cpp_int(x * 2); });
future<cpp_int> z = x.then([](const cpp_int& x){ return cpp_int(x / 15); });

cout << y.get() << endl;
cout << z.get() << endl;
```

```
708449696358523830150
23614989878617461005
```
Exercises

- Add support for:
  - Join (when_all)
  - Broken promises
  - Exception marshalling
  - Progress reporting
Property Models
What if we persist the graph?
What if we persist the graph?

- Allow multiple invocations of the tasks by setting the source values
- Each change triggers a notification to the sink values
  - This is a reactive programming model and futures are known as *behaviors*
How do the graphs change during execution?
A function is a directed relationship

- We can remove the arrows by providing a package of functions to represent the relationship
  - \( a = b \times c \)
  - \( b = a \div c \)
  - \( c = a \div b \)
- This forms a type of constraint system called a *property model*
- Flow is determined by value, or *cell*, priority

- Cells can only have one in-edge for a given flow or the system is over constrained
Property Model
Reflowing a property model doesn't require all relationships to be resolved
- The task representing them can still be executing concurrently
- This creates a single dependency graph that is appended to for each new flow and is pruned and *unravels* as tasks are complete
Property Model
Final Thoughts

- Perhaps representing such systems *as if* it were imperative code is not the correct approach.
- Instead, the graph description can be compiled and statically validated.