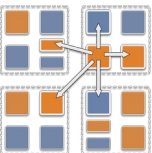


Chares Are Reactive

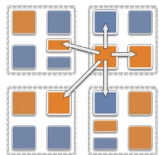
- The way we described Charm++ so far, a chare is a reactive entity:
 - If it gets this method invocation, it does this action,
 - If it gets that method invocation then it does that action
 - But what does it do?
 - In typical programs, chares have a life-cycle
- How to express the life-cycle of a chare in code?
 - Only when it exists
 - i.e. some chares may be truly reactive, and the programmer does not know the life cycle
 - But when it exists, its form is:
 - Computations depend on remote method invocations, and completion of other local computations
 - A DAG (Directed Acyclic Graph)!



Fibonacci Example

```
mainmodule fib {
  mainchare Main {
    entry Main(CkArgMsg* m);
  };

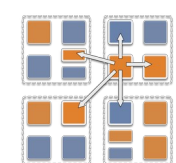
  chare Fib {
    entry Fib(int n, bool isRoot, CProxy_Fib
parent);
    entry void result(int value);
  };
};
```



Fibonacci Example

```
class Main : public CBase_Main {
public:
    Main(CkArgMsg* m) {
        CProxy_Fib::ckNew(atoi(m->argv[1]), true, CProxy_Fib());
    }
};

class Fib : public CBase_Fib {
public:
    CProxy_Fib parent; bool isRoot; int total, count;
    Fib(int n, bool isRoot_, CProxy_Fib parent_)
        : parent(parent_), isRoot(isRoot_), total(0), count(2) {
        if (n < THRESHOLD) {respond(seqFib(n)); }
        else { CProxy_Fib::ckNew(n - 1, false, thisProxy);
              CProxy_Fib::ckNew(n - 2, false, thisProxy);
              }
    }
};
```



Fibonacci Example

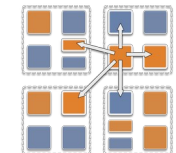
```
void result(int val) // when a child chare sends me its value
```

```
{total += val; if (--count == 0) respond(); }
```

```
void respond(int val) {  
    if (isRoot) { CkPrintf("Fibonacci number is: %d\n",  
result);  
                CkExit();  
                }  
    else { parent.result(total);  
           delete this;  
           // this is unusual. Tells the system to delete  
this  
           //chare after the entry method returns.  
    }  
}
```

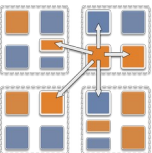
```
}
```

```
};
```



Consider the Fibonacci Chare

- The Fibonacci chare gets created
- If it is not a leaf,
 - It fires two chares
 - When both children return results (by calling respond):
 - It can compute my result and send it up, or print it
 - But in our example, this logic is hidden in the flags and counters
 - This is simple for this simple example, but ...
 - Lets look at how this would look with a little notational support

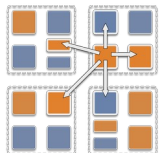


Structured Dagger: a script for a hare

- Actually, its a script for an entry method
 - But a common pattern is to use a single “run” method for a chare as an sdag (structured dagger) entry method
- You have to write this script in .ci file
 - Because we don't want to parse entire C++ code.
- Some entry methods are defined, rather than just declared, in the .ci file using sdag notation.
- Some other entry methods get implicitly defined if they get *used* in “when blocks” of sdag scripts

```
module xyz {
  chare abc {
    entry abc();
    entry f1();
    entry run() {
      sdag script here.
      includes when statements
    }
    entry g();
    entry h(..) {
      second sdag entry method.
    }
  };
}
```

.ci file

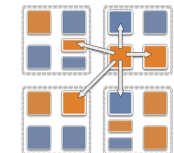


Structured Dagger

The when construct

- The when construct
 - Declare the actions to perform when a message is received
 - In sequence, it acts like a blocking receive

```
entry void someMethod() {  
    when entryMethod1(parameters) { /* block2  
*/ }  
    when entryMethod2(parameters) { /* block3  
*/ }  
};
```



Structured Dagger

The `serial` construct

- The `serial` construct
 - A sequential block of C++ code in the `.ci` file
 - The keyword `serial` means that the code block will be executed without interruption/preemption, like an entry method
 - Syntax: `serial <optionalString> { /* C++ code */ }`
 - The `<optionalString>` is used for identifying the serial for performance analysis
 - Serial blocks can access all members of the class they belong to

- Examples (`.ci` file):

```
entry void method1(parameters)
{
    serial {
        thisProxy.invokeMethod(10);
        callSomeFunction();
    }
};
```

```
entry void
method2(parameters) {
    serial "setValue" {
        value = 10;
    }
};
```

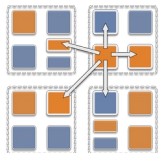

Structured Dagger

The implicit sequence construct

```
entry void someMethod() {  
    serial { /* block1 */ }  
    when entryMethod1(parameters) serial { /* block2  
*/ }  
    when entryMethod2(parameters) serial { /* block3  
*/ }  
}
```

- Sequence:

- Sequentially execute */*block1 */*
- Wait for `entryMethod1` to arrive, if it has not, return control back to the Charm++ scheduler, otherwise, execute */*block2 */*
- Wait for `entryMethod2` to arrive, if it has not, return control back to the Charm++ scheduler, otherwise, execute */*block3 */*



Structured Dagger

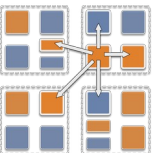
The when construct: waiting for multiple invocations

- Execute `sdagScript` when `method1` and `method2` arrive

```
when method1(int param1, int param2),  
      method2(bool param3)  
  {sdagScript}
```

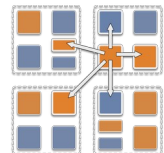
- Which is semantically the same as this:

```
when myMethod1(int param1, int param2) {  
  when myMethod2(bool param3) { }  
}  
{sdagScript}
```



Structured Dagger Boilerplate

- Structured Dagger can be used in any entry method (except for a constructor)
 - Can be used in a `mainchare`, `chare`, or `array`
- For any class that has Structured Dagger in it you must insert
 - The Structured Dagger macro: `[ClassName]_SDAG_CODE`



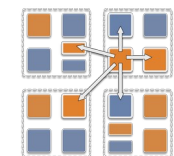
Structured Dagger Boilerplate

The .ci file:

```
[mainchare, chare, array] MyFoo {  
    ...  
    entry void method(parameters) {  
        // ... structured dagger code here ...  
    };  
    ...  
}
```

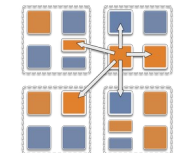
The .cpp file:

```
class MyFoo : public CBase_MyFoo {  
    MyFoo_SDAG_Code /* insert SDAG macro */  
public:  
    MyFoo() { }  
};
```



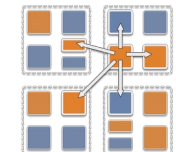
Fibonacci with Structured Dagger

```
mainmodule fib {
  mainchare Main {
    entry Main(CkArgMsg* m);
  };
  chare Fib {
    entry Fib(int n, bool isRoot, CProxy_Fib parent);
    entry void calc(int n) {
      if (n < THRESHOLD) serial { respond(seqFib(n)); }
      else {
        serial {
          CProxy_Fib::ckNew(n - 1, false, thisProxy);
          CProxy_Fib::ckNew(n - 2, false, thisProxy);
        }
        when result(int val), result(int val2)
          serial { respond(val + val2); }
      }
    };
    entry void result(int);
  };
};
```



Fibonacci with Structured Dagger

```
#include "fib.decl.h"
#define THRESHOLD 10
class Main : public CBase_Main {
public: Main(CkArgMsg* m) {
    CProxy_Fib::ckNew(atoi(m->argv[1]), true, CProxy_Fib()); } };
class Fib : public CBase_Fib {
public:
    Fib_SDAG_CODE
    CProxy_Fib parent; bool isRoot;
    Fib(int n, bool isRoot_, CProxy_Fib parent_):parent(parent_),
isRoot(isRoot_)
    { thisProxy.calc(n); }
    int seqFib(int n) { return (n < 2) ? n : seqFib(n - 1) + seqFib(n -
2); }
    void respond(int val) {
        if (!isRoot) { parent.response(val);
            delete this; }
        else { CkPrintf("Fibonacci number is: %d\n", val);
            CkExit(); }
    }
};
#include "fib.def.h"
```



Structured Dagger

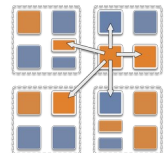
The when construct : reference number matching

- The `when` clause can wait on a certain reference number
- If a reference number is specified for a `when`, the first parameter for the `when` must be the reference number
- Semantics: the `when` will “block” until a message arrives with that reference number

```
when method1[100](int ref, bool param1)
    /* sdag block */
```

```
proxy.method1(200, false); /* will not be delivered to the above
when */
```

```
proxy.method1(100, true); /* will be delivered to the above when */
```

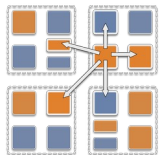


Structured Dagger

The `if-then-else` construct

- The `if-then-else` construct:
 - Same as the typical C `if-then-else` semantics and syntax

```
    if (thisIndex.x == 10) {  
        when method1[block](int ref, bool someVal) /* code  
block1 */  
    } else {  
        when method2(int payload) serial {  
            //... some C++ code  
        }  
    }
```



Structured Dagger

The for construct

The for construct:

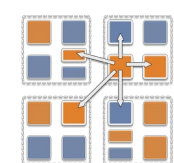
- Defines a sequenced for loop (like a sequential C for loop)
- Once the body for the i th iteration completes, the $i+1$ iteration is started

```
for (iter = 0; iter < maxIter; ++iter) {  
    when recvLeft[iter](int num, int len, double  
data[len])  
        serial { computeKernel(LEFT, data); }  
    when recvRight[iter](int num, int len, double  
data[len])  
        serial { computeKernel(RIGHT, data); }
```

iter must be defined as a class member

- Because no variables are allowed to be declared inside sdag scripts

```
class Foo : public CBase_Foo {  
    public: int iter;  
};
```



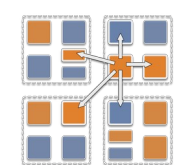
Structured Dagger

The `while` construct

The `while` construct:

- Defines a sequenced `while` loop (like a sequential C while loop)

```
while (i < numNeighbors) {  
    when recvData(int len, double data[len]) {  
        serial {  
            /* do something */  
        }  
        when method1() /* block1 */  
        when method2() /* block2 */  
    }  
    serial { i++; }  
}
```



Structured Dagger

The overlap construct

- The `overlap` construct:
 - By default, Structured Dagger defines a sequence that is followed sequentially
 - `overlap` allows multiple independent clauses to execute in any order
 - Any constructs in the body of an `overlap` can happen in any order
 - An `overlap` finishes in sequence when all the statements in it are executed
 - Syntax: `overlap { /* sdag constructs */ }`
- What are the possible execution sequences?

```
serial { /* block1 */ }  
overlap {  
    serial { /* block2 */ }  
    when entryMethod1[100](int ref_num, bool param1) /* block3 */  
    when entryMethod2(char myChar) /* block4 */  
}  
serial { /* block5 */ }
```

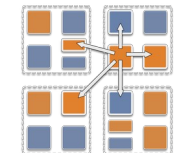


Illustration of a Long “Overlap”

- Overlap can be used to get back some of the asynchrony within a chore
 - But it is constrained
 - Makes for more disciplined programming
 - Fewer race conditions

