Component Approach to Integrated Modeling

Johan Carlsson

Tech-X Corporation

US-Japan Workshop on Integrated Modeling

September 21–24, 2004

We have to rethink how we code as code complexity grows!

- <1955: Binary code (zeros and ones)
  - Practical code size is a few hundred lines
- 1955–1960: Assembly language
  - Practical code size is a few thousand lines
- 1960–1990: High-level languages
  - Compiler translates portable, readable source code to binary
  - Subroutines are used to structure the code
- 1990–2005: Object-oriented programming
  - Data and associated operations grouped together in objects
  - Inheritance and polymorphism lets us add capability to existing, validated objects (code reuse)
- >2005: Components
  - Compose application at run time

Modern code design thinking could be particularly beneficial to fusion codes

- Focus has typically been on physics content
- Fairly well structured codes in terms of subroutines
- But, subroutines often act on global data (common or module variables) making their full effect non-obvious (hidden interactions)
- Codes have complicated internal state: high threshold to gaining the expertise needed to safely modify code
- A fusion code is never finished!
- More modular codes could simplify code maintenance and upgrades and make it easier to become a productive developer
- Makes it less impossible to get code work done at the dreaded 25% funding level

Components are a good fit for the Fusion Simulation Projects

Evolution of object-oriented ideas:

- Applications composed at runtime by loading components into a framework (somewhat similar to how a web browser loads plugins)
- Framework keeps track of the components and glues them together
- For the Fusion Simulation Project, existing fusion codes would be converted into components
  - Gives needed flexibility: to simulate RF-induced ITBs we compose an application from the RF and transport modules
  - We won't be doing the whole enchilada for a while

Common Component Architecture has a lot of what we need for FSP

The Common Component Architecture (CCA) is a DOE-funded framework developed specifically for HPC applications

- Very lightweight
- Framework supports components written in F77, F90, C, C++, Java, Python
- Allows components to share multi-dimensional arrays
- Does not yet support distributed components
  - We have an OASCR grant to investigate how to add a distributed capability
- Does not support multiple instantiations of components (loading the same component multiple times)

CCA components can be written in F77/90, C/C++, Python or Java

The Scientific Interface Description Language (SIDL) lets us specify component interfaces independently of implementation language.

```
package foo version 0.1 {
  class bar {
    static void baz(in double x, out float f);
  }
}
```

Run the Babel SIDL compiler to generate the glue code:

```
$ babel --server=F77 --output-dir=test test.sidl
```

Retrofit the interface to your legacy code:

```
        subroutine foo_bar_baz_fi(x, f)
        implicit none
        double precision x
        real f
C       DO-NOT-DELETE splicer.begin(foo.bar.baz)
C       Insert the implementation here...
C       DO-NOT-DELETE splicer.end(foo.bar.baz)
        end
```

Abuse of global variables in legacy codes makes componentization hard

Heavy use of global data (variables in common blocks or F90 modules) is a bad, old F77 habit!

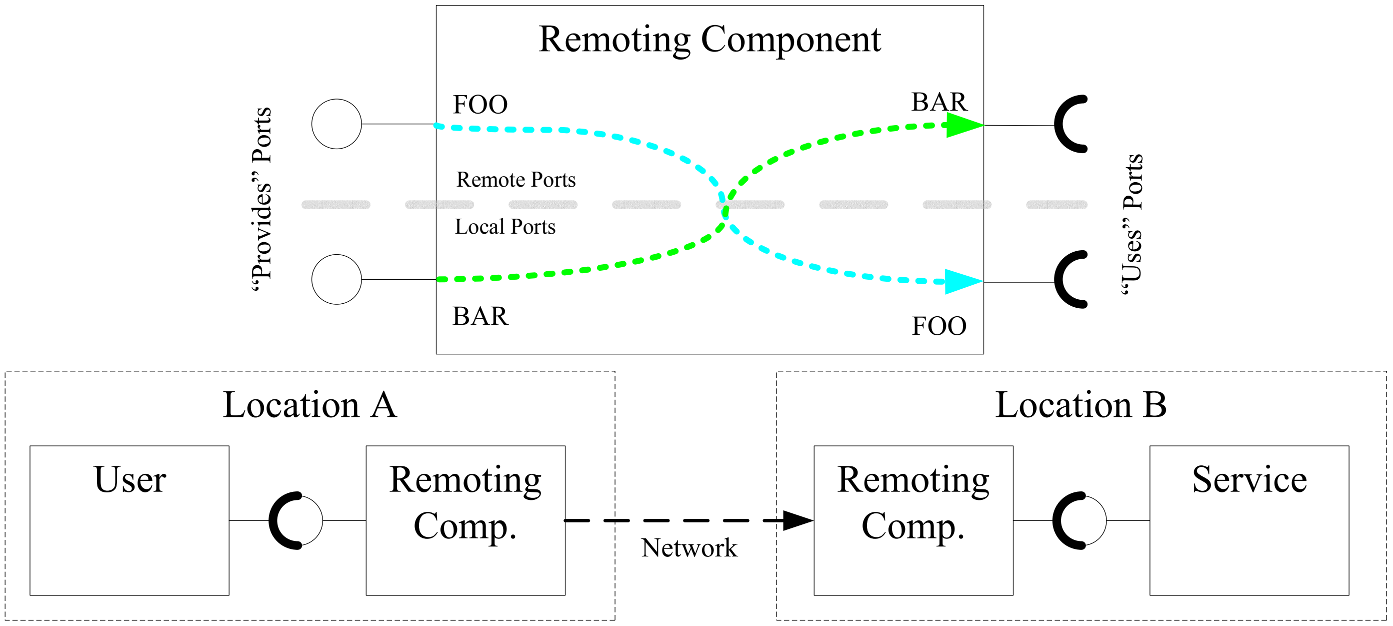
Exposes internal state of component (violates encapsulation).

Hard to avoid in F77, in F90 derived types provide the means to solve the problem (pass local variables of derived type on the stack).

```
program foo

call dostuf
call output

end

subroutine dostuf
real*8 bar
common /cmblok/ bar
bar = 2.0
return
end

subroutine output
real*8 bar
common /cmblok/ bar
write(*, *) bar
return
end
}
```

We're working on adding distributed capability to CCA

CCA currently doesn't support distributed components. Rather than bloating the framework, we have suggested a Remoting Component to add a distributed capability. The DOE Office of Advanced Scientific Computing (OASCR) has provided funding for this project.



Multiple component instantiation is needed for efficient time stepping

For Eulerian (first-order) time stepping we have (symbolically):

$$A(t + \Delta t) = A(t) + f[A(t)]\Delta t$$

Even for a simple second-order scheme (RK2), we get:

$$\bar{A} = A(t) + \frac{1}{2} f[A(t)]\Delta t$$
$$A(t + \Delta t) = A(t) + f[\bar{A}]\Delta t$$

Note that we need to copy $\bar{A} = A$! This is currently not possible (more about this later).

Similar for implicit schemes.

We expect this to be important for handling multi-scale problems (averaging over fast time scales).

Implementation of CCA components makes multiple instantiation hard

In CCA components are built on top of Shared Objects (.so files), aka Dynamically Loaded Libraries, and loaded using the function dlopen(). dlopen() returns a handle to a shared object. If called a second time, it returns a handle to the same Shared Object. Not what we want!

```
int Asc_DynamicLoad(CONST char *path, CONST char *initFun)
{
#define ASCDL_OK /* this line should appear inside each Asc_DynamicLoad */
  void *xlib;
  int (*install)();
  int result, addresult;
  char* _initFun;

  AscCheckDuplicateLoad(path); /* whine if we’ve see it before */
  /*
   * If the named library does not exist, if it’s not loadable or if
   * it does not define the named install proc, report an error
   */
  xlib = dlopen(path, 1);
  if (xlib == NULL) {
    FPRINTF(stderr,"%s\n",(char *)dlerror());
    return 1;
  }
```

Conclusion: CCA good foundation for FSP, some work needed

- Only component framework with support for Fortran modules
- Only component framework that supports multi-dimensional arrays
- Has momentum: DOE funded and very actively developed
- CCA people susceptible to FSP needs
- No distributed capability (yet)
- Multiple component instantiation (loading) not possible making implicit time stepping hard