

**CCA**

Common Component Architecture

---

# Language Interoperable CCA Components via



**CCA Forum Tutorial Working Group**

<http://www.cca-forum.org/tutorials/>

*tutorial-wg@cca-forum.org*



**JPL**

Lawrence Livermore  
National Laboratory

Los Alamos  
NATIONAL LABORATORY



**ornl**  
OAK RIDGE NATIONAL LABORATORY



UNIVERSITY  
OF COLORADO

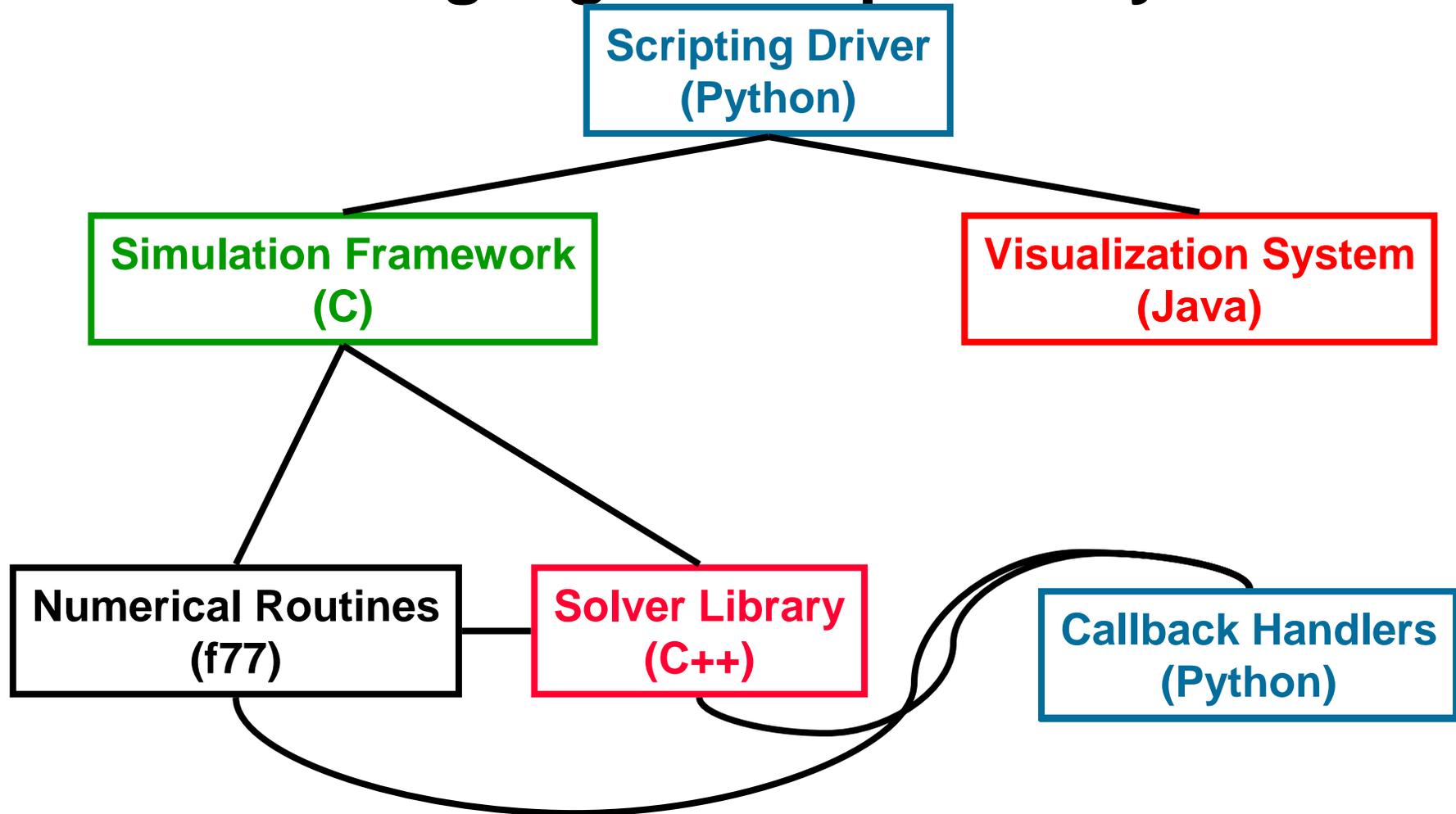
Sandia  
National  
Laboratories

# Goal of This Module

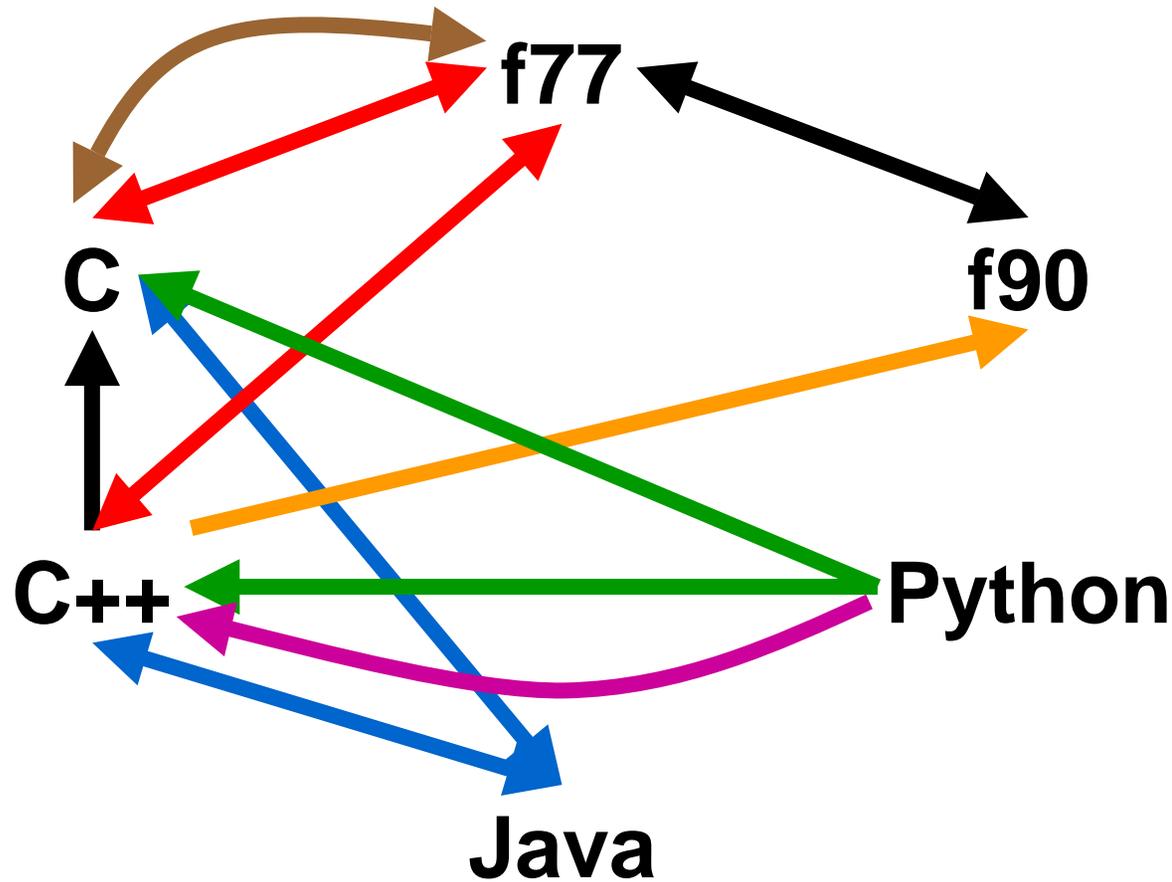
## Legacy codes → Babelized CCA Components

- Introduction To:
  - Babel
  - SIDL
- See Babel in use
  - “Hello World” example
  - Legacy Code (Babel-wrapped MPI)
  - CCA Tutorial Example (Numerical Integration)
- Relationship between Babel & CCA

# What I mean by “Language Interoperability”

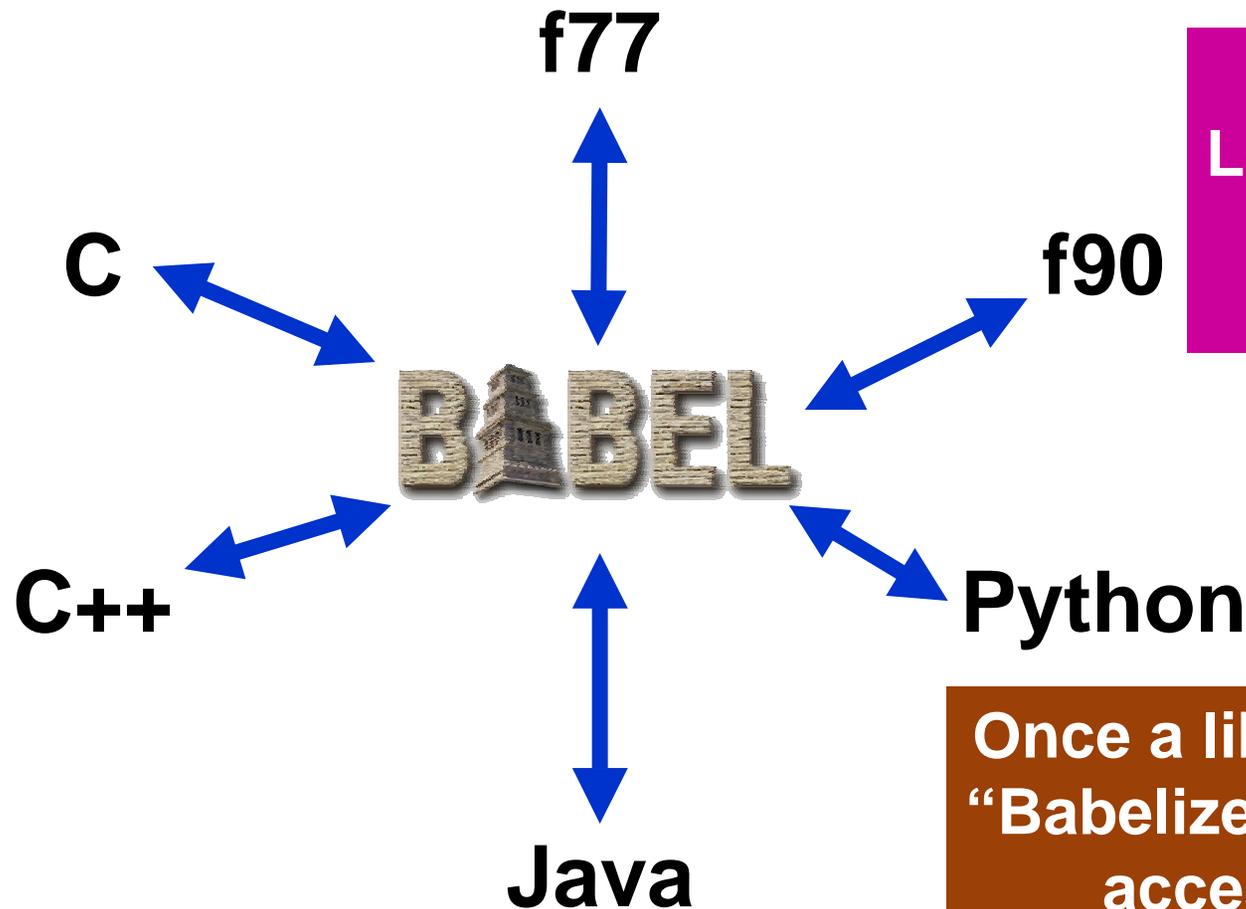


# One reason why mixing languages is hard



- Native
- cfortran.h
- SWIG
- JNI
- Siloon
- Chasm
- Platform  
Dependent

# Babel makes all supported languages peers



This is not a  
Lowest Common  
Denominator  
Solution!

Once a library has been  
“Babelized” it is equally  
accessible from all  
supported languages

# Babel Module's Outline

- Introduction



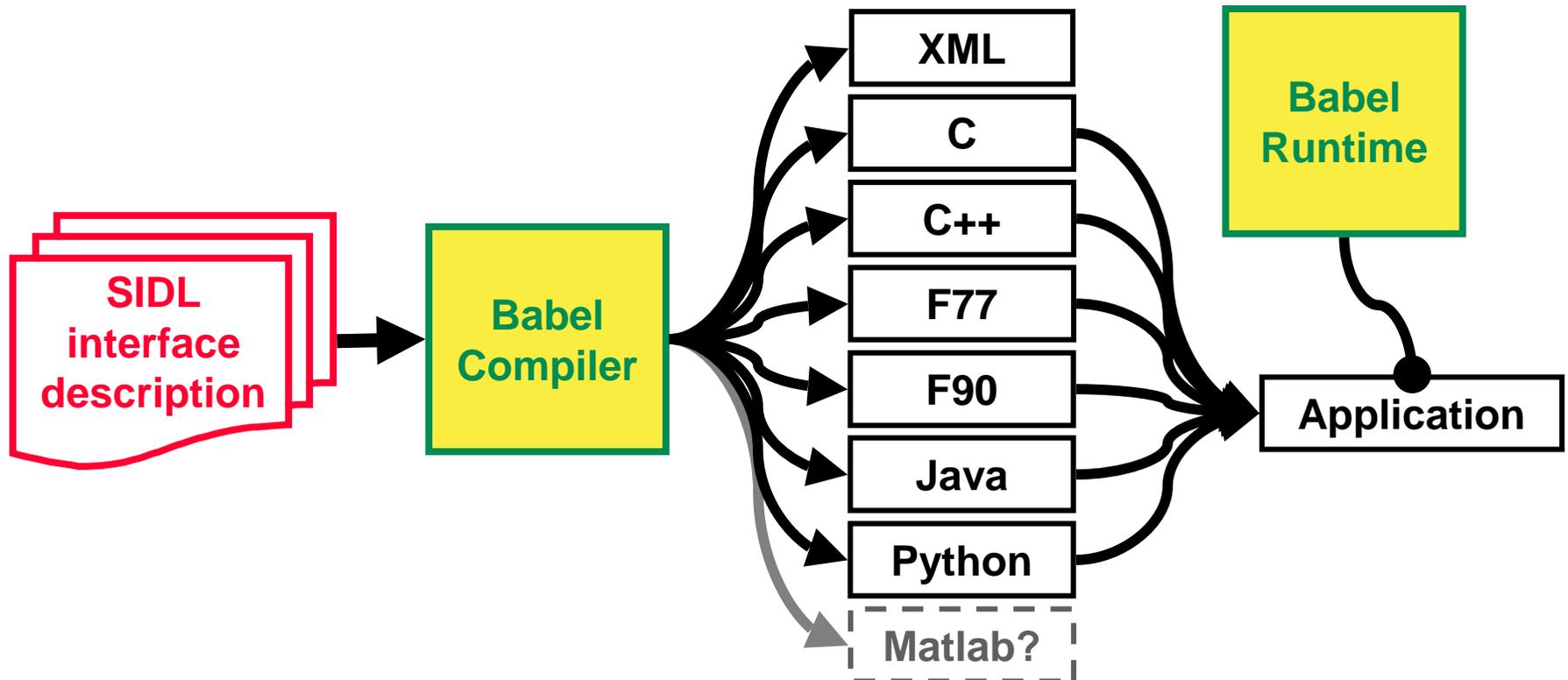
- **Babel Basics**

- How to use Babel in a “Hello World” Example
- SIDL Grammar
- Example: Babel & Legacy Code

- Babel & CCA

- Relationship between them
- How to write a Babelized CCA Component

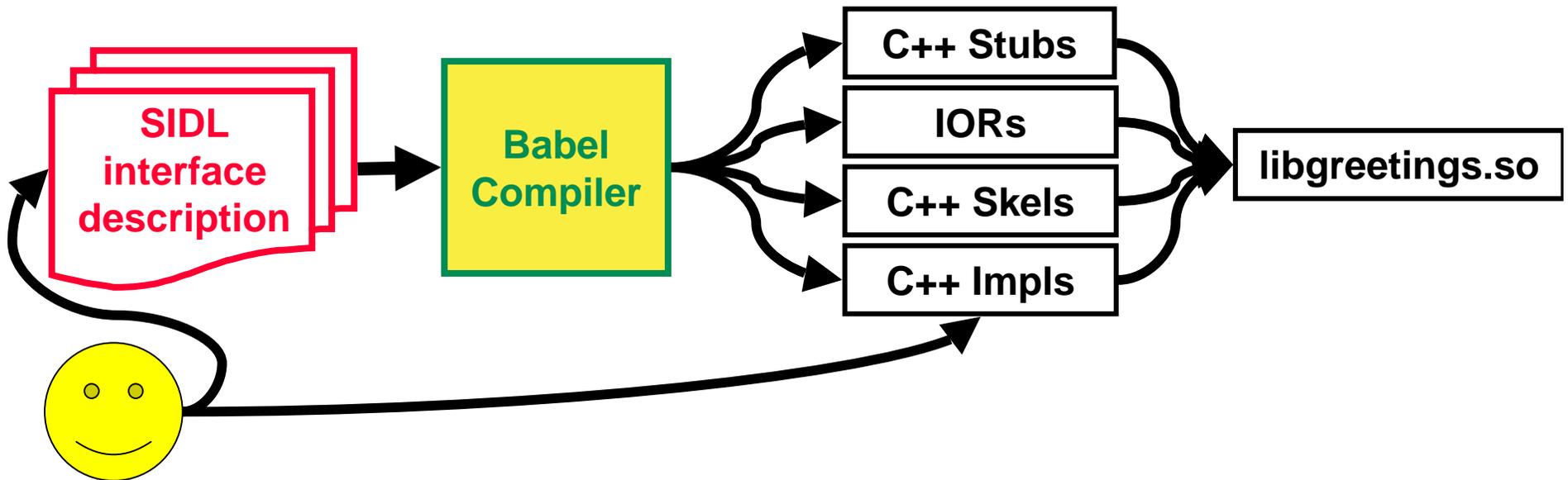
# Babel's Two Parts: Code Generator + Runtime Library



## greetings.sidl: A Sample SIDL File

```
package greetings version 1.0 {  
    interface Hello {  
        void setName( in string name );  
        string sayIt ( );  
    }  
    class English implements-all Hello { }  
}
```

## Library Developer Does This...



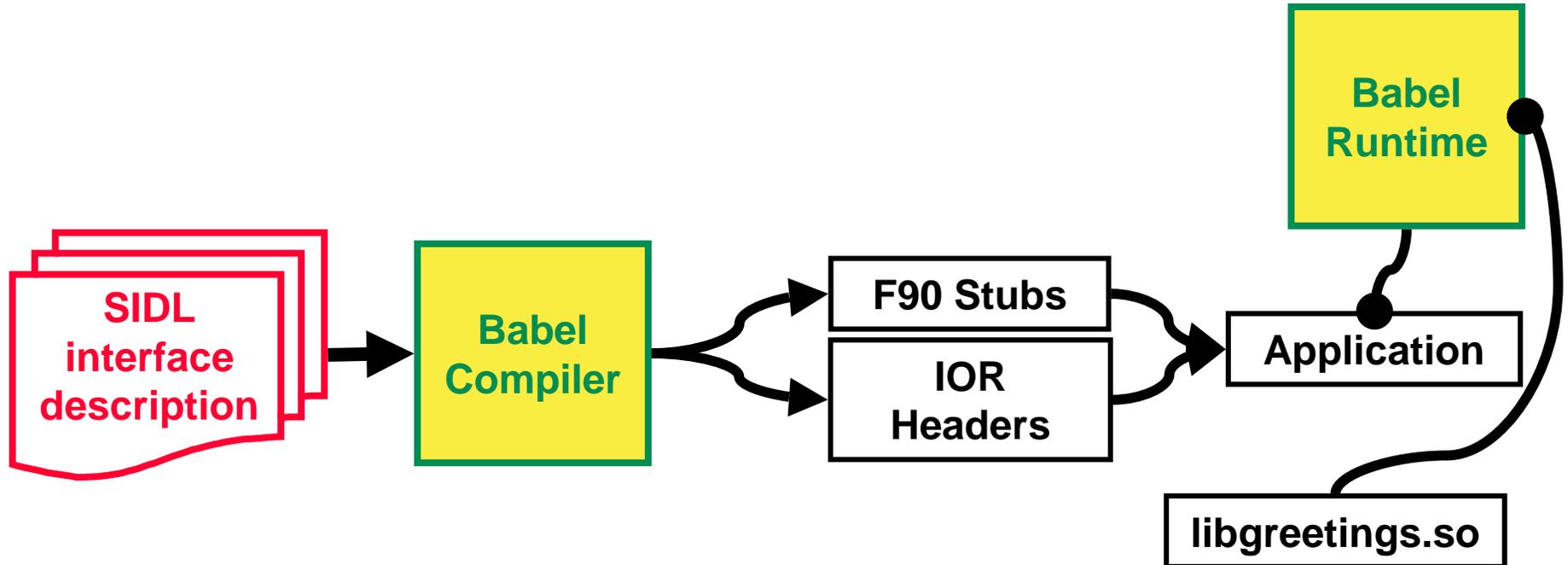
1. ``babel --server=C++ greetings.sidl``
2. Add implementation details
3. Compile & Link into Library/DLL

# Adding the Implementation

```
namespace greetings {  
class English_impl {  
private:  
    // DO-NOT-DELETE spl i cer. begi n(greeti ngs. Engl i sh. _i mpl )  
    string d_name;  
    // DO-NOT-DELETE spl i cer. end(greeti ngs. Engl i sh. _i mpl )
```

```
string  
greetings::English_impl::sayl t()  
throw ()  
{  
    // DO-NOT-DELETE spl i cer. begi n(greeti ngs. Engl i sh. sayl t)  
    string msg("Hello ");  
    return msg + d_name + "!";  
    // DO-NOT-DELETE spl i cer. end(greeti ngs. Engl i sh. sayl t)  
}
```

## Library User Does This...



1. ``babel --client=F90 greetings.sidl``
2. Compile & Link generated Code & Runtime
3. Place DLL in suitable location

# F90/Babel “Hello World” Application

```
program helloclient
  use greetings_english
  implicit none
  type(greetings_english_t) :: obj
  character (len=80)          :: msg
  character (len=20)         :: name
```

```
  name=' World'
  call new( obj )
  call setName( obj, name )
  call sayIt( obj, msg )
  call deleteRef( obj )
  print *, msg
```

```
end program helloclient
```

**These subroutines  
come from directly  
from the SIDL**

**Some other subroutines  
are “built in” to every  
SIDL class/interface**

# SIDL Grammar (1/3): Packages and Versions

- Packages can be nested

```
package foo version 0.1 { package bar { ... } }
```

- Versioned Packages
  - defined as packages with explicit version number OR packages enclosed by a versioned package
  - Reentrant by default, but can be declared final
  - May contain interfaces, classes, or enums
- Unversioned Packages
  - Can only enclose more packages, not types
  - Must be re-entrant. Cannot be declared final

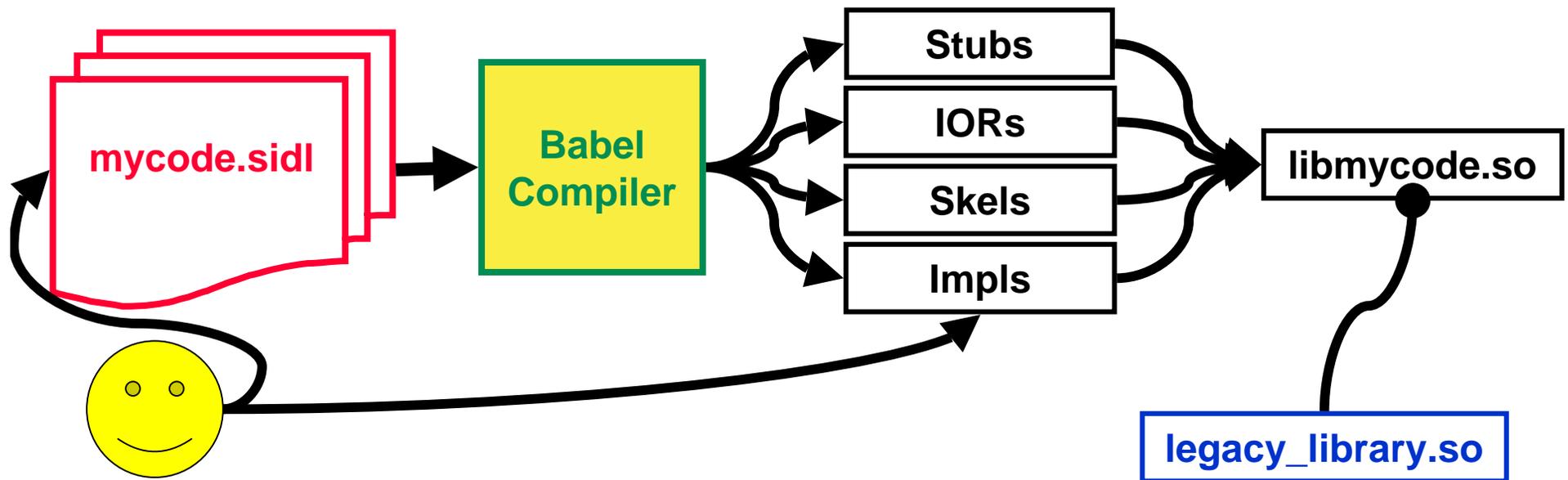
## SIDL Grammar (2/3): Classes & Interfaces

- SIDL has 3 user-defined objects
  - **Interfaces** – APIs only, no implementation
  - **Abstract Classes** – 1 or more methods unimplemented
  - **Concrete Classes** – All methods are implemented
- Inheritance (like Java/Objective C)
  - Interfaces may **extend** Interfaces
  - Classes **extend** no more than one Class
  - Classes can **implement** multiple Interfaces
- Only concrete classes can be instantiated

## SIDL Grammar (3/3): Methods and Arguments

- Methods are **public virtual** by default
  - **static** methods are not associated with an object instance
  - **final** methods can not be overridden
- Arguments have 3 parts
  - Mode: can be **in**, **out**, or **inout** (like CORBA, but semantically different than F90)
  - Type: one of (bool, char, int, long, float, double, fcomplex, dcomplex, *array<Type,Dimension>*, enum, interface, class )
  - Name

# Babelizing Legacy Code



1. Write your SIDL interface
2. Generate server side in your native language
3. Edit Implementation (Impls) to dispatch to your code  
(Do NOT modify the legacy library itself!)
4. Compile & Link into Library/DLL

# Known Projects Using Babel

(see [www.llnl.gov/CASC/components/gallery.html](http://www.llnl.gov/CASC/components/gallery.html) for more)

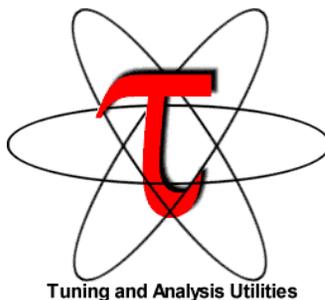
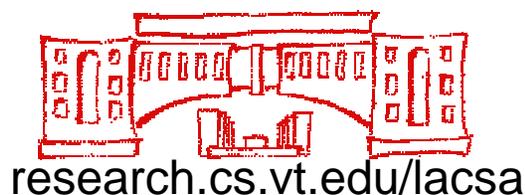


*hypre*  
high performance  
preconditioners



I implemented a Babel-based interface for the hypre library of linear equation solvers. The Babel interface was straightforward to write and gave us interfaces to several languages for less effort than it would take to interface to a single language.

--Jeff Painter, LLNL.



# Babel & Legacy Code (e.g. MPI)

```
package mpi version 2.0 {  
  class Comm {  
    int send[Int]( in array<int, 1, row-major> data,  
                  in int dest, in int tag );  
    ...  
  }  
}
```

mpi . si dl

## Babel & Legacy Code (e.g. MPI)

```
struct mpi_Comm__data {  
    /* DO-NOT-DELETE spl i cer. begi n(mpi . Comm. _data) */  
    MPI_Comm com;  
    /* DO-NOT-DELETE spl i cer. end(mpi . Comm. _data) */  
};
```

mpi\_comm\_impl . h

```
int32_t  
impl_mpi_Comm_sendInt( mpi_Comm self, SIDL_int__array data,  
                        int32_t dest, int32_t tag ) {  
    /* DO-NOT-DELETE spl i cer. begi n(mpi . Comm. sendInt) */  
    struct mpi_Comm__data *dptr = mpi_Comm__get_data( self );  
    void * buff = (void*) SIDL_int__array_fi rst(data);  
    int count = length(data);  
    return mpi_send( buff, count, MPI_INT, dest, tag, dptr->comm);  
    /* DO-NOT-DELETE spl i cer. end(mpi . Comm. sendInt) */  
}
```

mpi\_comm\_impl . c 19

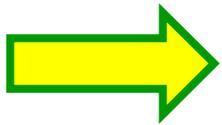
## Investing in Babelization can **improve** the interface to the code.

***“When Babelizing LEOS [an equation of state library at LLNL], I completely ignored the legacy interface and wrote the SIDL the way I thought the interface should be. After running Babel to generate the code, I found all the hooks I needed to connect LEOS without changing any of it. Now I’ve got a clean, new, object-oriented python interface to legacy code. Babel is doing much more than just wrapping here.”***

**-- Charlie Crabb, LLNL  
(conversation)**

# Babel Module's Outline

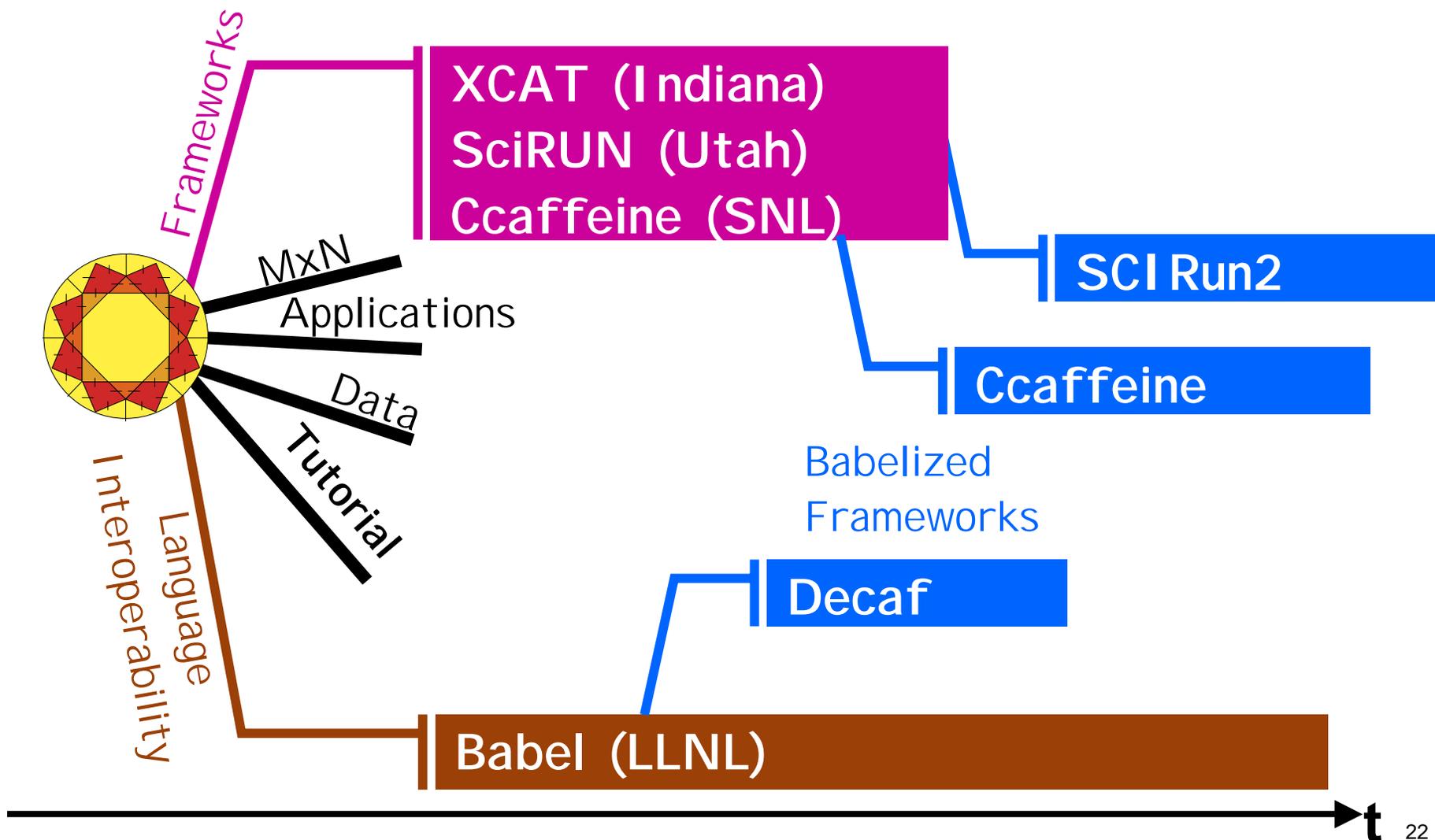
- Introduction
- Babel Basics
  - How to use Babel in a “Hello World” Example
  - SIDL Grammar
  - Example: Babel & Legacy Code



## Babel & CCA

- Relationship between them
- How to write a Babelized CCA Component

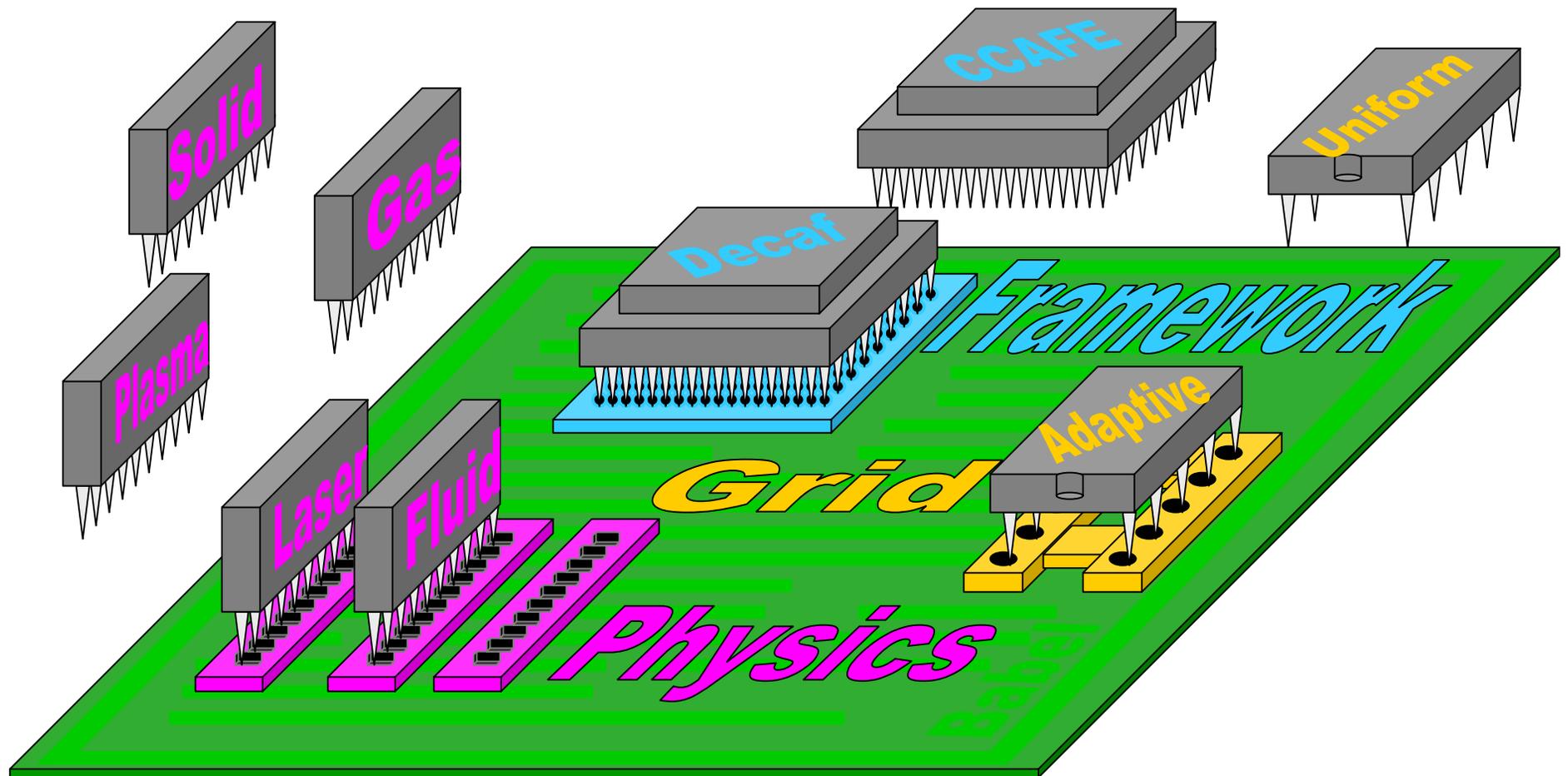
# History of Babel & CCA



# The CCA Spec is a SIDL File

```
package gov {  
package cca version 0.6.1 {  
    interface Port { }  
    interface Component {  
        void setServices( in Services svcs );  
    }  
    interface Services {  
        Port getPort( in string portName );  
        registerUsesPort( /*etc*/ );  
        addProvidesPort( /*etc*/ );  
  
        /*etc*/  
    }  
}
```

# The CCA from Babel's POV

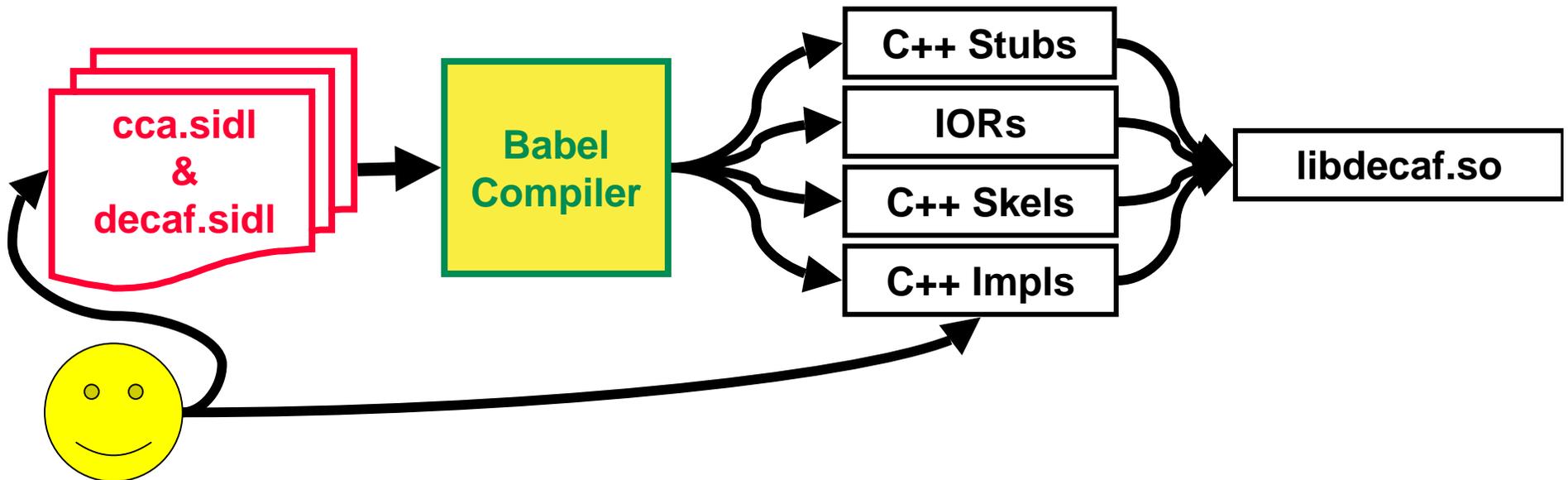


# Decaf:

## Details & Disclaimers

- Babel is a hardened tool
- Decaf is an example, not a product
  - Distributed in “examples” subdirectory of Babel
  - Decaf has no GUI
- Decaf is CCA compliant
  - Babelized CCA Components can be loaded into Decaf, CCAFFEINE, and SCIRun2
- **“Understanding the CCA Specification Using Decaf”**  
**<http://www.llnl.gov/CASC/components/docs/decaf.pdf>**

## How I Implemented Decaf



1. wrote decaf.sidl file
2. ``babel --server=C++ cca.sidl decaf.sidl``
3. Add implementation details
4. Compile & Link into Library/DLL

# How to Write and Use Babelized CCA Components

1. Define “Ports” in SIDL
2. Define “Components” that implement those Ports, again in SIDL
3. Use Babel to generate the glue-code
4. Write the guts of your component(s)

# How to Write A Babelized CCA Component (1/3)

## 1. Define “Ports” in SIDL

- CCA Port =
  - a SIDL Interface
  - extends gov.cca.Port

```
package functions version 1.0 {  
    interface Function extends gov.cca.Port {  
        double evaluate( in double x );  
    }  
}
```

# How to Write A Babelized CCA Component (2/3)

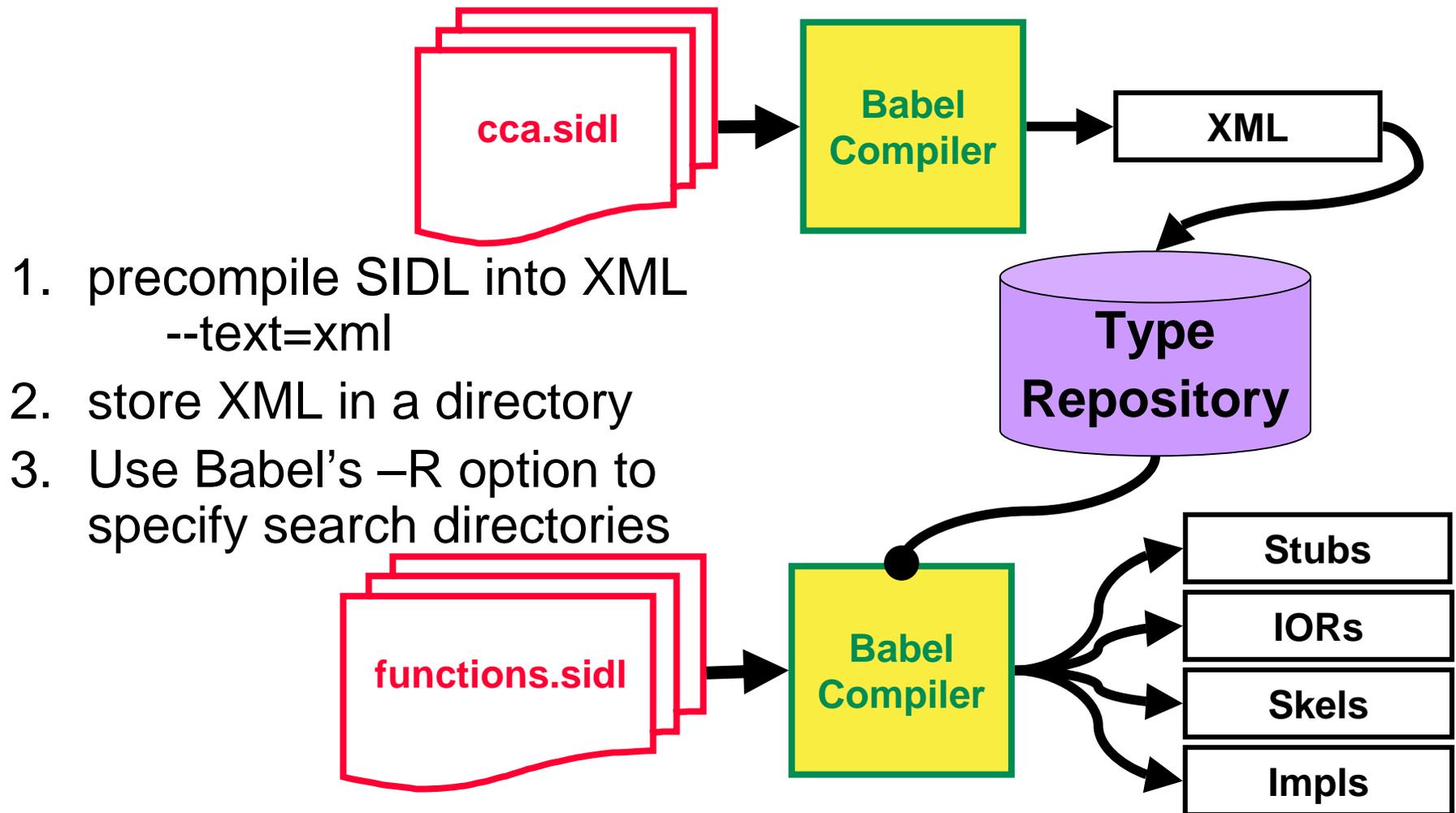
## 2. Define “Components” that implement those Ports

- CCA Component =
  - SIDL Class
  - implements gov.cca.Component (& any provided ports)

```
class LinearFunction implements functions.Function,  
                                gov.cca.Component {  
    double evaluate( in double x );  
    void setServices( in cca.Services svcs );  
}
```

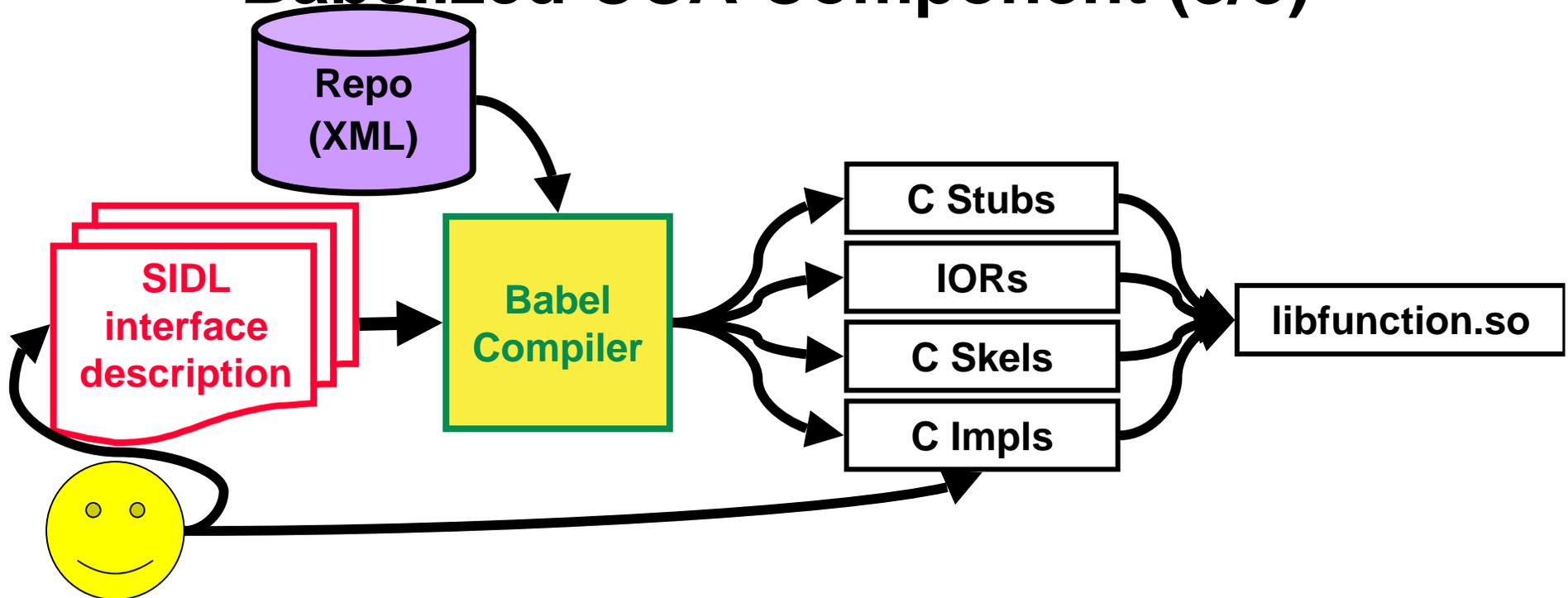
```
class LinearFunction implements-all  
    functions.Function, gov.cca.Component { }
```

## Tip: Use Babel's XML output like precompiled headers in C++



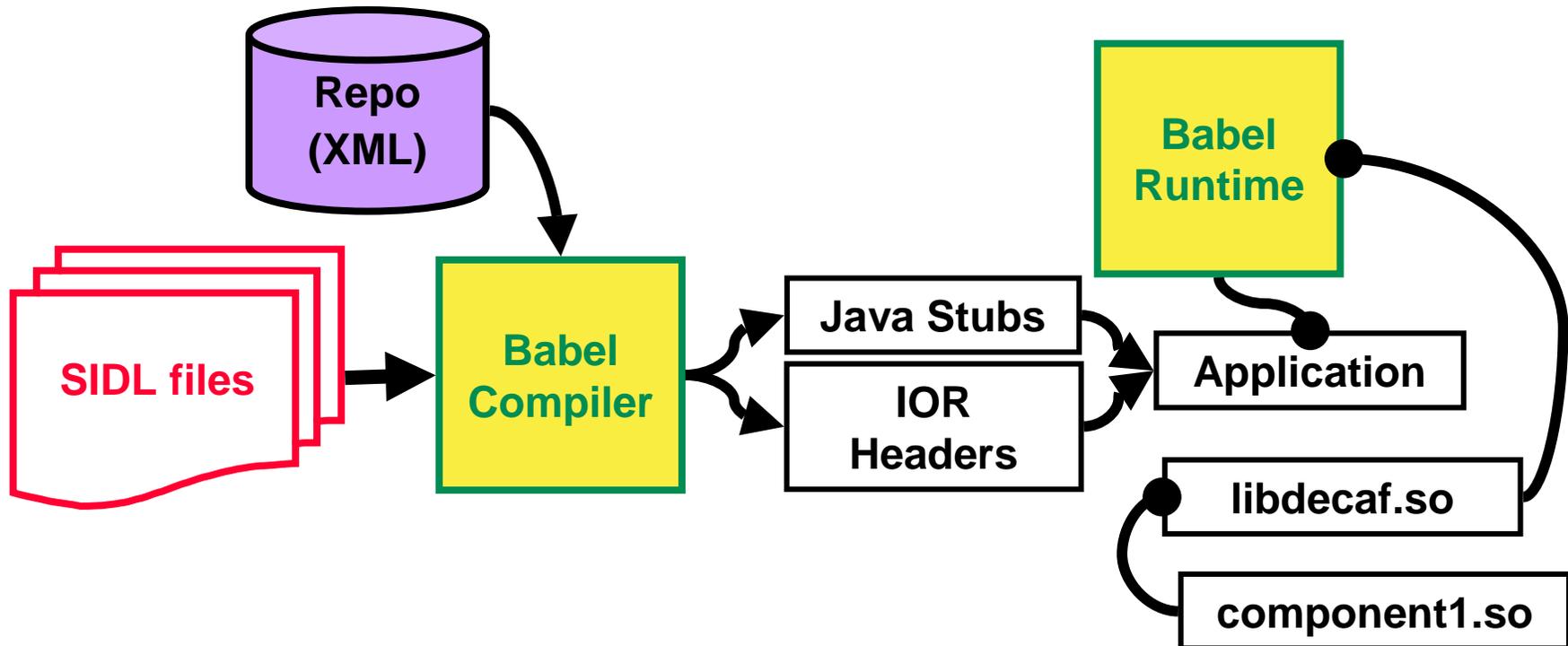
1. precompile SIDL into XML  
--text=xml
2. store XML in a directory
3. Use Babel's `-R` option to specify search directories

## How to Write A Babelized CCA Component (3/3)



3. Use Babel to generate the glue code
  - ``babel --server=C -Rrepo function.sidl``
4. Add implementation details

## To Use the Decaf Framework

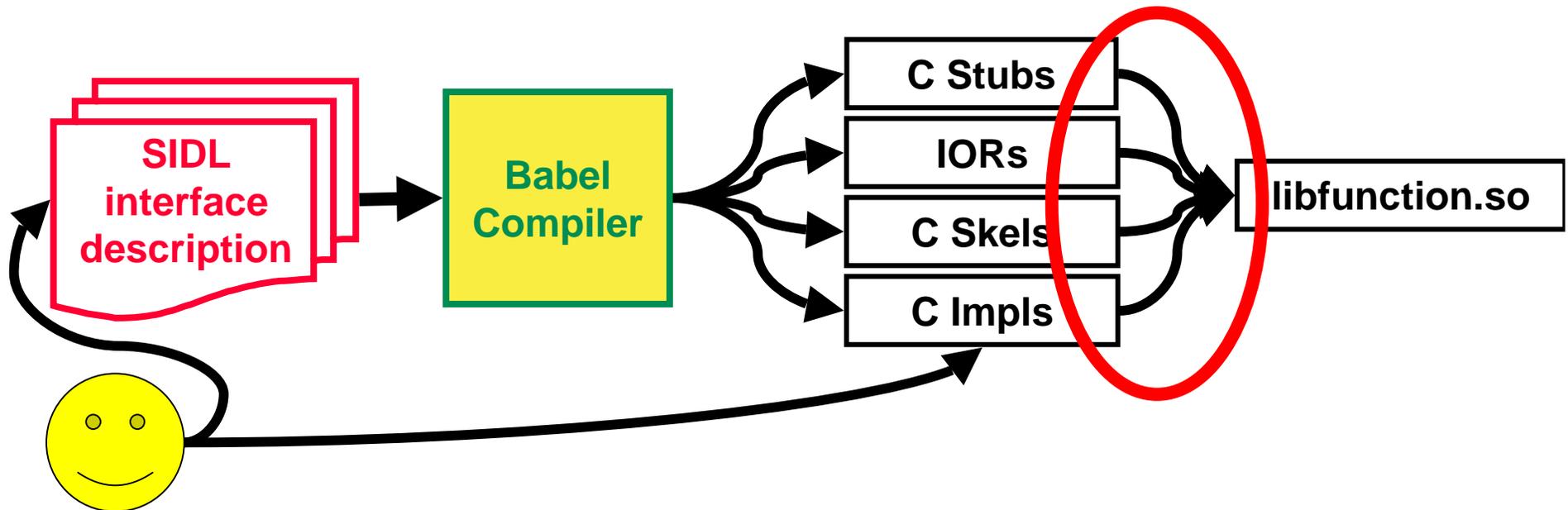


1. ``babel --client=Java -Rrepo function.sidl``
2. Compile & Link generated Code & Runtime
3. Place DLLs in suitable location

# Limitations of Babel's Approach to Language Interoperability

- Babel is a code generator
  - Do obscure tricks no one would do by hand
  - Don't go beyond published language standards
- Customized compilers / linkers / loaders beyond our scope
  - E.g. icc and gcc currently don't mix on Linux
  - E.g. No C++-style templates in SIDL. (Would require special linkers/loaders to generate code for template instantiation, like C++ does.)
- Babel makes language interoperability feasible, but not trivial
  - Build tools severely underpowered for portable multi-language codes

## What's the Hardest Part of this Process?



- Properly building libraries for multi-language use
- Dynamically loadable .so files are especially error prone
  - Not a lot of understanding or expertise in community
  - Causality chain between improperly constructed DLLs and observed bugs is often inscrutable and misleading

# Summary

## Legacy codes → Babelized CCA Components

- Reclassify your objects in your legacy code
  - Things customers create → CCA components
  - Logical groups of a component's functionality → CCA Port
  - Low level objects in your implementation → not exposed
- Generate SIDL File
  - CCA port → Babel Interface that extends the Babel interface called "gov.cca.Port"
  - CCA component → Babel Class that implements the Babel interface called "gov.cca.Component" (and possibly its "provides ports")
- Run Babel (choose server-language for your code)
- Articulate Impl files to dispatch to legacy code

## Contact Info

- Project: <http://www.llnl.gov/CASC/components>
  - Babel: language interoperability tool
  - Alexandria: component repository
  - Quorum: web-based parliamentary system
  - Gauntlet (coming soon): testing framework
- Bug Tracking: <http://www-casc.llnl.gov/bugs>
- Project Team Email: [components@llnl.gov](mailto:components@llnl.gov)
- Mailing Lists: [majordomo@lists.llnl.gov](mailto:majordomo@lists.llnl.gov)
  - subscribe babel-users *[email address]*
  - subscribe babel-announce *[email address]*

UCRL-PRES-148796

5, July 2002



This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48