Introducing
Design by Contract to SIDL/Babel

Tammy Dahlgren
with
Tom Epperly, Scott Kohn, and Gary Kumfert
Center for Applied Scientific Computing

Common Component Architecture Working Group

October 3, 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.

UCRL-PRES-150101

Overview

- Goals
- Basic Constructs
- Impact on SIDL/Babel
- Benefits for the CCA
- Future Work

Why support assertions at the interface specification level?

The interface specification can provide a simple, concise description of the *requirements, behavior,* and *constraints*.

Generated code will automatically ensure compliance regardless of the underlying implementation language.

The SIDL grammar defines packages, interfaces, etc.

- Packages & Versions
- Interfaces & Classes
- Inheritance Model
- Methods
- Method Modifiers
- Intrinsic Data Types
- Parameter Modes
- And more…

Optional assertion specifications added here

There are several types of assertions mentioned in the literature.

Type	Express…
Precondition	● Constraints to enable proper method function ● Conditions that must be true *prior to* invocation
Postcondition	● Guarantees of proper method function ● Conditions that must be true *after* invocation
Class Invariant	● Global properties of instances that must be true *upon* instance creation and preserved by all routines *before and after* every invocation
Loop Invariant	● Instance properties that must be true *prior to* the first execution of a loop *and* preserved by every iteration so *hold on* loop termination
Loop Variant	● An integer value that must be non-negative *prior to* first execution of a loop and decreased by every iteration to guarantee loop termination

Only the first two Design by Contract clauses will be added at this time.

Clause	Comment
Preconditions	● Specify library's requirements ● Obligations on callers
Postconditions	● Specify library's guarantees ● Obligations on callee to *provided* preconditions were satisfied *and* no exceptions raised

Method sequencing will be implemented using a sequence, or state , clause the values of which can be utilized in the pre- and post-conditions to specify method ordering.

The new clauses require a number of additions to the SIDL grammar.

● Eiffel keywords	
— Preconditions	requires
— Postconditions	ensures
● Simple conditional expression operators	
— Logical	&&, ||, !
— Bitwise Logical	&, ^, |
— Relational	<, <=, ==, !=, >=, >
— Shift	<<, >>
— Additive	+, -
— Multiplicative	*, /, %
● Logical grouping	()
● Literal keywords	TRUE, FALSE, NULL, return
● Terminals	

The following specification snippet illustrates the use of both clauses in SIDL.

```
interface Vector {
  Vector axpy (in Vector a, in Vector x) {
        requires a != NULL;
                 x != NULL;
        ensures return != NULL;
  };
  double norm () {
        ensures return >= 0.0;
  };
```

Vector.sidl

Recall: If method raises an exception then *no* guarantee the ensures will be met!

If method sequencing were incorporated, call ordering state would be added.

First item is the initial state.

```
interface Vector {
 state { uninitialized, initialized };

  void setData (in double data){
    requires uninitialized;
    ensures initialized;
  };
  …
```

Transition to *initialized* is automatic if library call is successful and all (other) postcondition entries met.

VectorWithOrdering.sidl

Note: Sequencing constructs subject to change.

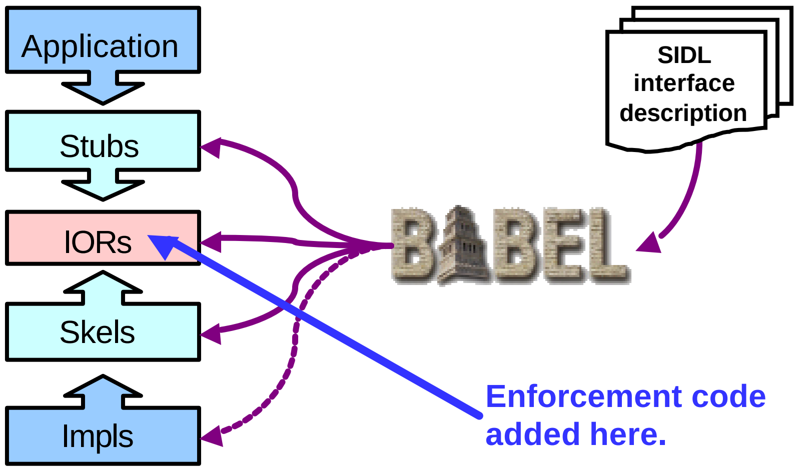
Methods that require the instance to be in a state could annotate it accordingly.

```
interface Vector {
  …
  Vector axpy (in Vector a, in Vector x) {
      requires initialized;
               a != NULL;
               x != NULL;
      ensures return != NULL;
  };
  double norm () {
      requires initialized;
      ensures return >= 0.0;
  };
```

VectorWithOrdering.sidl

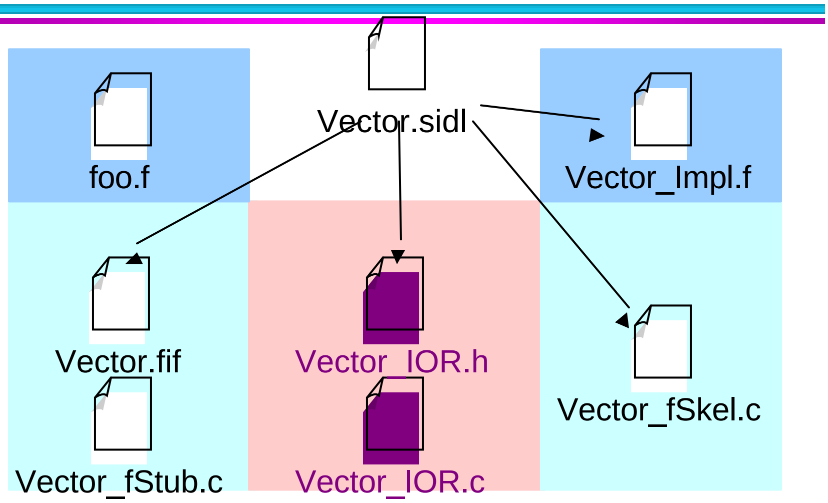
Note: Sequencing constructs subject to change.

Babel takes a SIDL file and will generate expanded glue code.

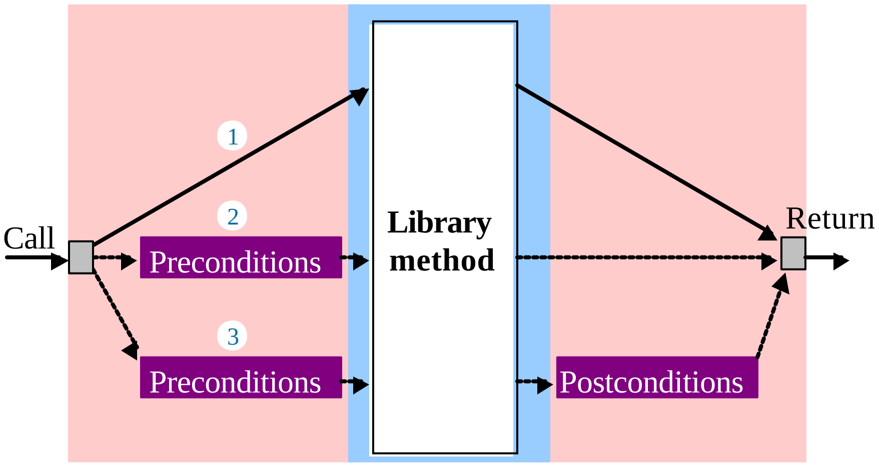


Enforcement code added here.

The IOR files will be changed to add the generated checks.



There will be three execution paths available through the IOR.

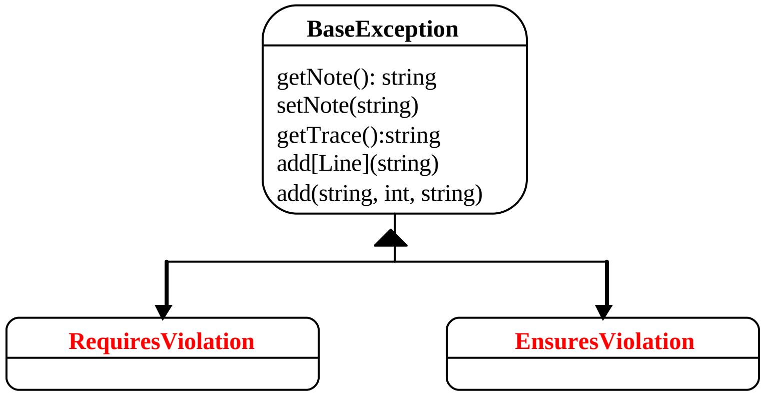


Dynamic switching between paths can be available at up to four levels.

- Instance
  - Vector.__create(/* desired setting */);
  - Vector.__noChecks(); // Path #1
  - Vector.__checkRequires(); // Path #2
  - Vector.__checkAsserts(); // Path #3
- *Class -- For all instances of a class*
- *Package -- For a subset of packages*
- *Global -- Through the SIDL Loader*

What degree of flexibility is needed?
- Dynamic switching at the Class level? Package level? Loader level?
- ~~Regular expression support for specifying classes? Packages?~~

Violations of assertions will result in the raising of new SIDL exceptions.



One or more *requires* conditions failed!

One or more *ensures* conditions failed!

Which means several parser-related files must change in the compiler.

File	Change(s)
dtds/SIDL.dtd	Add elements for the assertion lists and conditions.
parsers/sidl/SIDL.jj	Add support for the new grammar productions.
parsers/xml/ParseSymbolXML.java	Add parsing for new structures from XML.
symbols/Method.java	Add support for assertion lists.
symbols/*newclass(es)*.java	New file(s) associated with the assertion list productions to support the lists.

The backends must also be modified to support the IOR and stub changes.

Files	Change(s)
backend/IOR.java backend/IOR/IORHeader.java backend/IOR/IORSource.java	Add support for new built-in methods for dynamic switching and the new entry point vectors.
backend/*language*/StubHeader.java* backend/*language*/StubSource.java*	Add support for new built-in methods for dynamic switching.

* These or their equivalent are generally present for each supported language.

Interface-level assertions will ultimately facilitate wider reuse of CCA Components!

For **Library developers**:
+ requirements explicit
+ constraints explicit
+ sequencing explicit
+ automatic enforcement
+ enhanced debugging

For **Domain Scientists**, components will be:
+ well-debugged
+ well-documented
+ easier to use

Future work focuses on adding and exploring more features.

- Add support for specifying and enforcing method sequencing
- Explore annotations and mechanisms for:
  - Enabling the use of a method within assertions (e.g., const or immutable)
  - Checking features of an instance (e.g., comparing sizes of two matrices or vectors)
  - Integrating and checking relevant domain-specific properties (e.g., standard units, types of matrices)
    - Automated determination of compatibility
    - Generation and automated use of translation routines
- *Anything else?*