Babel/SIDL
Design by Contract: Status

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

Common Component Architecture Working Group
April 10, 2003

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-152674
Overview

- Basic Constructs
- Impact on Babel/SIDL
- Status of Phase I
- Benefits
- Future Work
The SIDL grammar supports optional assertion and sequencing specifications.

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

Optional specifications added here
Three classic assertion mechanisms supported in the interface descriptions.

<table>
<thead>
<tr>
<th>Type</th>
<th>Specify…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invariant</td>
<td>• unchanging properties of instances of a class</td>
</tr>
<tr>
<td></td>
<td>• must be <em>true upon</em> instance creation and preserved by all routines</td>
</tr>
<tr>
<td></td>
<td>• <em>before and after</em> every invocation</td>
</tr>
<tr>
<td>Precondition</td>
<td>• when it is valid to invoke a method</td>
</tr>
<tr>
<td></td>
<td>• must be <em>true prior to</em> invocation</td>
</tr>
<tr>
<td>Postcondition</td>
<td>• effects of a method and results it will return</td>
</tr>
<tr>
<td></td>
<td>• must be <em>true after</em> invocation</td>
</tr>
</tbody>
</table>

Plus method call invocation sequencing!
Method call sequencing enforcement is provided by Babel using object states.

```idl
interface Vector {
    states { uninitialized, initialized[final] };

    void setData (in double data)
        require in uninitialized;
        ensure now_ready : in initialized;

    ...
}
```

Default initial state.

Explicit final state.

Optional assertion label.

Transition to `initialized` is automatic if library call is successful and all (other) postconditions and invariants met.
Pre- and post-conditions are typically used to constrain arguments and results.

interface Vector {
    states { uninitialized, initialized[final] };
    ...

    Vector axpy (in Vector a, in Vector x)
        require in initialized; a != NULL; x != NULL;
        ensure result_not_null : result != NULL;
    
    double norm ()
        require object_is_initialized : in initialized;
        ensure result >= 0.0; is pure;
}

Vector.sidl

Note: Argument a is vector instead of scalar for illustration purposes only.
A number of additions to the original SIDL grammar were made.

- **Clauses**: states, **invariant**, require, require else, ensure, ensure then

- **Conditional expressions**
  - Logical: **implies**, or, xor, and
  - Relational: ==, !=, <, <=, >=, >
  - Shift: <<, >>
  - Additive: +, -
  - Multiplicative: *, /, mod, rem
  - Unary: +, -, ~, not, in, is
  - Postfix: method call
  - Logical grouping: ()

- **Terminals**: boolean, double, float, integer\(^1\), long\(^1\), character, string, identifier

- **Literal keywords**: true, false, null, result, pure

\(^1\)Decimal, Hexadecimal, and Octal.
Optional object states and invariants added to classes and interfaces.

Class ::=  
[abstract] class name
[extends scoped-class-name]
[implements-all scoped-interface-name-list]
{ [ ObjectStates ] [ Invariants ]
class-methods-list
} [:]

Interface ::=  
interface name
[extends scoped-interface-name-list]
{ [ ObjectStates ] [ Invariants ]
methods-list
} [:]
Object states definition is used to specify list of valid states.

ObjectStates ::= states {
    state-1 [ initial | final ]
    [ , state-2 [ initial | final ] ]
    ...
    [ , state-n [ initial | final ] ]
} [ ; ]

Default initial state is first item in list.
Default final state is last item in list.

states { uninitialized, initialized[final] };

Explicit final state.
Invariant definition is used to specify unchanging properties of objects.

Invariants ::= invariant AssertionList;

AssertionList ::= [label-1 :] AssertionExpression-1;
[[label-2 :] AssertionExpression-2;]
...
[[label-n :] AssertionExpression-n;]

An “is pure” method must be specified elsewhere in this interface.

invariant {non-negative : entriesAreNonNegative()};

Optional assertion label for debugging messages.
Method definitions allow specification of pre- and post-conditions.

ClassMethod ::= \[ (\ abstract | \ final | \ static ) \ ]

Method

Method ::= ( void | [ copy ] Type ) name [ extension ]
         ( [ ArgumentList ] )
         [ local | oneway ]
         [ throws ScopedExceptionList ]
         [ Requires ] [ Ensures ] ;

Requires ::= require [ else ] AssertionList ;

Ensures ::= ensure [ then ] AssertionList ;
The modifications had a significant impact on the grammar and symbol table.

<table>
<thead>
<tr>
<th>Area</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIDL Grammar</td>
<td>• Added</td>
</tr>
<tr>
<td></td>
<td>— 42 terminal symbols/lexical tokens (↑ 91%)</td>
</tr>
<tr>
<td></td>
<td>— 21 productions (↑ 140%)</td>
</tr>
<tr>
<td></td>
<td>• Modified 3 productions</td>
</tr>
<tr>
<td>Symbol Table</td>
<td>• Added 17 classes (↑ 77%)</td>
</tr>
<tr>
<td></td>
<td>• Modified 4 classes</td>
</tr>
<tr>
<td>XML DTD</td>
<td>• Added 5 elements (↑ 22%)</td>
</tr>
<tr>
<td></td>
<td>• Modified 3 elements</td>
</tr>
</tbody>
</table>
Expanded glue code generated from enhanced interface descriptions.

In SIDL or XML.

Enforcement code added here.
The generated checks added to the IOR files.
Five basic execution paths available through the IOR.

*Method call sequencing enforcement cannot be supported.
There are still several features that need to be completed/addressed.

- DTD/XML support
- Assertion enforcement options
  - Dynamic switching basis
    - Class, object, method, etc.
  - Assertion type combinations
    - Preconditions only, pre & post, invariants, etc.
    - Assertion expression evaluation levels
      - State checks only, cheap only, etc.
- Generated code
Benefits of including these contracts in Babel/SIDL include…

- Better designs and documentation
  - Behavior and call ordering more explicit
- Improved debugging and reliability
  - Runtime checking of consistency between specifications and code
  - Runtime checking of client call ordering
- Better support for reuse
- Supported regardless of native support in the underlying implementation language
Future work focuses on adding and/or exploring additional features such as...

- **Terminals**
  - float and double complex
  - non-primitive SIDL types (e.g., arrays)
- **Operation:** $x^y$
- **Literal keyword:** old
  - Pre-method state?
  - Guarded postconditions associated with superclasses
    (old precondition) implies original_postcondition
- **Assertion exception policies**
- **Domain-specific features** – to be determined
An assessment of your level of interest and anticipated usefulness is needed.

- Is this capability of interest to you? Why or why not?
- Do you anticipate adopting this at some point? If so, within what context?

Thank You!