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

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.

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

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

<table>
<thead>
<tr>
<th>Clause</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preconditions</td>
<td>● Specify library’s requirements</td>
</tr>
<tr>
<td></td>
<td>● Obligations on callers</td>
</tr>
<tr>
<td>Postconditions</td>
<td>● Specify library’s guarantees</td>
</tr>
<tr>
<td></td>
<td>● Obligations on callee to provided preconditions were satisfied and no exceptions raised</td>
</tr>
</tbody>
</table>
### 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.

```idl
interface Vector {
    Vector axpy (in Vector a, in Vector x) {
        requires a != NULL;
        x != NULL;
        ensures return != NULL;
    }

double norm () {
    ensures return >= 0.0;
}
}
```

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.

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

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

    ...  
}
```

VectorWithOrdering.sidl

`Note: Sequencing constructs subject to change.`

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

```java
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.

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.

BaseException

g>Note(): string
setNote(String)
g>GetTrace(): string
add(Line)(string)
add(string, int, string)

RequiresViolation

EnsuresViolation

→ One or more requires conditions failed!

→ One or more ensures conditions failed!

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

<table>
<thead>
<tr>
<th>File</th>
<th>Change(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtds/SDL.dtd</td>
<td>Add elements for the assertion lists and conditions.</td>
</tr>
<tr>
<td>parsers/sidl/SIDL.jj</td>
<td>Add support for the new grammar productions.</td>
</tr>
<tr>
<td>parsers/xml/ParseSymbolXML.java</td>
<td>Add parsing for new structures from XML.</td>
</tr>
<tr>
<td>symbols/Method.java</td>
<td>Add support for assertion lists.</td>
</tr>
<tr>
<td>symbols/newclass(ee).java</td>
<td>New file(s) associated with the assertion list productions to support the lists.</td>
</tr>
</tbody>
</table>
The backends must also be modified to support the IOR and stub changes.

<table>
<thead>
<tr>
<th>Files</th>
<th>Change(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>backend/IOR.java</td>
<td>Add support for new built-in methods for dynamic switching and the new entry point vectors.</td>
</tr>
<tr>
<td>backend/IOR/IORHeader.java</td>
<td></td>
</tr>
<tr>
<td>backend/IOR/IORSource.java</td>
<td></td>
</tr>
<tr>
<td>backend/language/StubHeader.java*</td>
<td></td>
</tr>
<tr>
<td>backend/language/StubSource.java*</td>
<td></td>
</tr>
</tbody>
</table>

* 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?