Toward Multi-Language, Multi-Component Interface Contract Enforcement

[Extended Abstract]

Tammy Dahlgren
Lawrence Livermore National Laboratory
Computer Applications and Research Department
dahlgren1@llnl.gov

Irina Abramova
University of Southern California
Viterbi School of Engineering
iabramov@usc.edu

ABSTRACT

This paper describes current work aimed at helping scientists gain confidence in software built from emerging component technologies through the automated enforcement of interface contracts. These contracts consist of assertions required to hold before and after interface methods are executed. Runtime contract enforcement is a well-known technique for enhancing testing and debugging, but is typically considered too expensive for deployment. Prior work investigated strategies intended to retain an application’s high performance while enforcing contracts during plug-and-play component deployment. The associated studies involved single-component implementations in C and C++. Current efforts seek to expand and harden the Babel middleware toolkit’s contract support. The emphasis is on enforcement across all supported languages. Example clients and implementations have been added to the Babel source code distribution for nearly every supported language combination. Preliminary work on programs utilizing multiple classes and Common Component Architecture-compliant components has also been performed.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—programming by contract

Keywords

Babel, Common Component Architecture (CCA), Interface Contracts, Programming Language Interoperability, Scientific Interface Definition Language (SIDL)

1. INTRODUCTION

This paper describes work aimed at helping scientists gain confidence in software built from emerging component technologies through the automated enforcement of interface contracts. Component technologies are based on independent software units, called components, with interface specifications describing their use [8]. Interface contracts make explicit the obligations on the caller and all implementations of the specified methods. Current efforts focus on hardening, extending, and demonstrating support for contract enforcement for applications involving multiple Common Component Architecture (CCA)-compliant components, including implementations in different programming languages.

Runtime contract enforcement is a well-known technique for enhancing testing and debugging but can also help ensure proper functioning of software during third-party component deployment. The interface contracts are formed from precondition, postcondition, and/or class invariant clauses belonging to the interface specification not the underlying implementation(s). Precondition clauses consist of assertions on properties required to hold prior to method execution. Postcondition clauses contain assertions ensured to hold upon method completion. Class invariants apply before and after execution of all defined methods. For example, Figure 1 shows the contract for a product method that

Figure 1: Interface contract, written in the Scientific Interface Definition Language (SIDL), for the vector product method. Preconditions require the caller provide a non-null, 1-dimensional array (u). Postconditions ensure all implementations of the method return a non-null, 1-dimensional array (result), of the same size as the incoming array, u.

array<double> vuProduct(in double a, in array<double> u) throws

   sidl.PreViolation, sidl.PostViolation;
   require /* Preconditions */
   not_null: u != null;
   u_is_1d: dimen(u) == 1;
   ensure /* Postconditions */
   result_not_null: result != null;
   result_is_1d: dimen(result) == 1;
   result_correct_size: size(result) == size(u);

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CBHP'09 Nov. 15-16, Portland, OR, USA
Copyright 2009 ACM 978-1-60558-718-9/09/11 ...$10.00.
takes a double value and a 1-dimensional array of doubles and returns a 1-dimensional array of the same size.

Prior work built the underlying infrastructure for component interface contract enforcement and studied its effects for individual C and C++ implementations [2–5]. Extensions have since been made to support enforcement for additional programming languages and to prepare for studies involving applications built of multiple CCA-compliant components. This paper summarizes these efforts.

2. MOTIVATION

The CCA Forum [1] seeks to bring the benefits of component-based software to the high-performance scientific computing community. The goal is to facilitate the development of increasingly ambitious applications seeking to leverage significant advances in today’s hardware. The inherent complexity of new applications — including legacy software written in different programming languages and the ability to swap components at runtime — present challenging environments for debugging and testing. Third-party use of deployed components exacerbates the problem in part because the software may experience unanticipated usage patterns and/or unexpected input combinations.

Runtime swapping of components implementing the same interface but written in different programming languages raises concerns regarding the consistency of checks associated with and enforcement of contracts. Programming languages used to write legacy scientific software generally lack native support for contracts or even basic assertion mechanisms. Software teams and organizations typically follow different development practices. Manually ensuring the software operates and produces valid results under these conditions is impractical.

Another common concern, especially for legacy codes, is the need to preserve the independence of the original software. This is accomplished by decoupling contracts and their enforcement from the underlying implementation. This feature is especially important for codes used in both component- and non-component-based applications.

The approach taken here to addressing both the consistency and decoupling issues leverages scientific component middleware technologies. Specifications written in a programming language-neutral form can include contracts. When included, the corresponding checks are generated in the middleware of each implementation of the interface; thereby providing contract enforcement consistency and decoupling from component implementations.

3. CONTRACT ENFORCEMENT

The contract enforcement infrastructure is integrated into the Babel [6] toolkit. The toolkit consists of an interface specification language, a compiler, and a runtime library. Scientific Interface Definition Language (SIDL) is the specification language that has been adopted by the CCA. The Babel compiler transforms SIDL specifications, including interface constraints, into interoperability middleware for components written in any of the following programming languages: C, C++, FORTRAN 77, Fortran 90, Java, and Python. Checks associated with contract clauses are generated in ANSI C in the Intermediate Object Representation (IOR) layer. Figure 2 illustrates the paths taken when enforcement is enabled and disabled through the middleware for a call to a method with one or more contract clauses in its specification. Finally, the SIDL Runtime library provides object-oriented middleware management including features for controlling and monitoring contract enforcement.

Much of the research so far focuses on addressing a common perception that checking assertions and contracts generally takes too long for deployment runs. The Babel Runtime library provides options for more than a dozen different enforcement strategies, ranging from disabling checking to filtering by contract clause to (basic and performance-driven) sampling to enforcing all contract clauses. Experiments conducted with single-component programs indicate the overhead of full contract enforcement is negligible if there is sufficient work performed within component methods [2,3]. Furthermore, sampling based on performance criteria can mitigate some of the overhead of enforcing contract clauses taking more time to execute relative to the time spent in interface methods.

4. EXPANDED PROGRAMMING LANGUAGE SUPPORT

The Babel toolkit now includes demonstrations of contract enforcement across all but one supported programming language, as illustrated in Figure 3. The examples use and implement a vector utilities interface consisting of thirteen methods, including one nearly identical to that shown in Figure 1. Client programs are implemented in C, C++, FORTRAN 77, Java, and Python. Each client invokes all of the methods specified in the interface (at least once) for each implementation in C, C+++, FORTRAN 77, Fortran 90, Java, and Python. Work on the final client, written in Fortran 90, is in progress.

Figure 2: Babel-generated interface contract enforcement middleware [3], illustrating an external component or library. Control passes from the program, through the stub layer and, if contract enforcement is enabled, through a contract enforcement routine (written in ANSI C) in the intermediate object representation layer before passing through Babel’s skeleton then implementation layers.
5. MULTI-COMPONENT CONTRACT ENFORCEMENT

Preliminary work hardening multi-language, multi-component contract enforcement started with the implementation of two classic computer science graph algorithms: Breadth-first and Depth-first search. Their designs and implementations are based on published algorithms [7].

Breadth-first search attempts to find the shortest path to all reachable nodes from a given starting node. The result of this search is represented in a list of reachable graph vertices supplemented with the required number of hops from the start node.

Depth-first search targets topological sorting, which attempts to determine an acceptable ordering for carrying out tasks that depend on one another. The algorithm determines the correct precedence for vertices in a directed graph. The ordering, or precedence, of the vertices is not necessarily unique.

Figure 4 lists the ports associated with the main CCA components for this example. The algorithms were initially implemented using the Babel interface and classes before being refactored to work within the CCA framework. Key classes were redeclared as components using or providing services, or ports. The Driver component has a standard CCA “go” port for fetching all declared “uses” ports. Once the handles are obtained for GraphSourcePort, InitGraphPort, and GraphSearchPort, methods are called on the providing port components; namely, GraphCFactory, InitGraph, and BreadthFirstSearch, respectively.

The underlying graph is a set of SIDL classes implemented in C. Interface contracts were added to the associated interface and class specifications. For example, Figure 5 shows the contract associated with the method for inserting a vertex into the graph. Callers of the insVertex method are expected to provide an existing vertex identified by the Data argument. Given a valid vertex, all implementations of the interface are obligated to ensure the vertex exists in the graph upon completion of the method’s execution.

```cpp
int insVertex(in Data d) throws
   sidl.PreViolation, sidl.PostViolation;
require
   not_exist: (not vertexExists(d));
ensure
   vert_exist: vertexExists(d);

bool vertexExists(in Data d);
ensure
   no_side_effects: is pure;
```

Figure 5: Contracts for inserting a graph vertex using the insVertex method and ensuring it exists with the vertexExists method. The is pure assertion is actually a non-executable annotation indicating the method can safely be used within a contract clause (since it has no side effects).
Figure 6 illustrates a screen shot of the CCA Graphical User Interface (GUI) for the graph search components. In this case the BreadthFirstSearch component is being executed. Since both BreadthFirstSearch and DepthFirstSearch components are declared as using the same search port, they could be swapped at runtime.

6. SUMMARY AND FUTURE WORK

The goal of this research is to help scientists gain confidence in software built from emerging component technologies through executable interface contracts. Previous work demonstrated contract enforcement for single-component programs written in C and C++. The Babel toolkit has been modified to harden, extend, and demonstrate contract enforcement for all supported languages, though work remains on the Fortran 90 example client.

Preliminary work has also been performed to demonstrate contract enforcement in multi-component applications. Babel-only examples of breadth-first and depth-first search check interface contracts on the graph implementation. Contract enforcement for CCA-compliant graph search components is undergoing testing.

The next phase of this work involves: implementing the algorithms and components in additional languages; developing additional algorithms and components; and conducting a study to measure contract enforcement effects with different enforcement strategies.

7. ACKNOWLEDGMENTS

We thank Boyana Norris, Tom Epperly, Ben Allan, and Rob Armstrong for their help with the CCA tools. Special thanks to the anonymous reviewers for their helpful feedback.

This work was supported by the Office of Advanced Scientific Computing Research via the Scientific Discovery through Advanced Computing (SciDAC) initiative [9], Office of Science, U.S. Department of Energy, under Contract DE-AC52-07NA27344.

8. REFERENCES


