A Framework for Detecting Interface Violations in Component-Based Software

Stephen H. Edwards[†]
Gulam Shakir[‡]
Murali Sitaraman
Computer Science and Elec. Engineering
West Virginia University
Morgantown, WV 26506-6109
{edwards,shakir,murali}@cs.wvu.edu

Bruce W. Weide
Computer and Info. Science
The Ohio State University
Columbus, OH 43210-1277
weide@cis.ohio-state.edu

Joseph Hollingsworth
Computer Science
Indiana University Southeast
PS105-4201 Grant Line Rd.
New Albany, Indiana 47150
jholly@ius.indiana.edu

Abstract

Two kinds of interface contract violations can occur in component-based software: a client component may fail to satisfy a requirement of a component it is using, or a component implementation may fail to fulfill its obligations to the client. This paper proposes a systematic approach for detecting both kinds of violations, so that violation detection is not hard-coded into base-level components, but is "layered" on top of them, and so that it can be turned "on" or "off" selectively for one or more components, with practically no change to executable code (limiting changes to a few declarations). Among the salient features of this approach are its use of formal specifications, the ability to handle parameterized (i.e., generic, or template) components, and the automatic generation of routine aspects of violation detection. We have designed, built, and experimented with a generator of checking components for C++ templates.

***Keywords**: specification, error handling, wrapper, checking component, generator, design by contract, generic, template.*

1 Introduction

In component-based software, interfaces (or specifications) are separated from implementations and serve as contracts between users and implementers of components [9]. In practice, many failures in component-based systems arise because of interface violations among components—where one party breaks the contract. Specifically, the violations occur when

(a) A client component fails to satisfy a requirement of a component it is reusing, or

(b) A component implementation fails to fulfill its obligations to the client.

The objective of this paper is to describe a general approach for detecting interface violations within the context of component-based software, for cases where both the violating and violated components are software pieces. The proposed approach is general, and allows violation handling to be layered instead of being hard-coded into components. It can be applied to one or more components of a system and can be turned "on" or "off" selectively, for effective detection and isolation of violations. Among the other salient features of this approach are its use of formal specifications, the ability to handle parameterized (i.e., generic, or template) components, and the automatic generation of routine aspects of violation detection.

Section 2 describes the violation detection approach. Section 3 introduces a component interface example, presented as a C++ template. The interface permits alternative implementations, none of which is assumed to include code for handling pre-condition (or postcondition) violations. Section 4 explains how, through transformation of internal representations to mathematical models and vice versa, it is possible to understand and check violations in purely abstract terms. This technique allows the checks to be written even without access to the component's internal details (i.e., layered) by using the proposed set of transformation operations. Section 5 discusses a generator for C++ template components that mechanizes routine aspects of checking, and generates placeholders when the generation cannot be automated. Section 6 compares this approach with existing work, and Section 7 closes with a discussion of applications and future work.

[†] Stephen Edwards is currently with Virginia Tech, Computer Science Dept., Blacksburg, VA, 24060-0106.
[‡] Gulam Shakir is currently with Parametric Technology Corporation, Boca Raton, FL 33432.

2 Elements of a specification-based approach for detecting interface violations

2.1 Overview

Typically, two alternatives for checking violations of a component's interface are used in practice: incorporate the checking code as part of the client component or incorporate it as part of the reused component. Unfortunately, both of these choices have undesirable consequences. The ubiquitous principle of separation of concerns suggests that the process of checking for interface violations should be *separated* from the client component. At the same time, it should be *separated* from the component because otherwise expensive checks will be performed, even when they are not needed. This observation leads to a wider view of inter-component communication shown in Figure 1.

The idea expressed in Figure 1 is simple: encase the component under consideration in a "wrapper" (sometimes also called a decorator [5]), whose sole responsibility is to implement interface violation checks. This "checking wrapper" provides exactly the same interface as the component itself, and delegates the work involved in carrying out each operation to the component that it encases. The checking wrapper also has the opportunity to perform run-time checks both before and after each method invocation, in order to detect interface violations. If every software component in a component-based system is encased in a checking wrapper, then it becomes possible to detect and isolate interface violations.

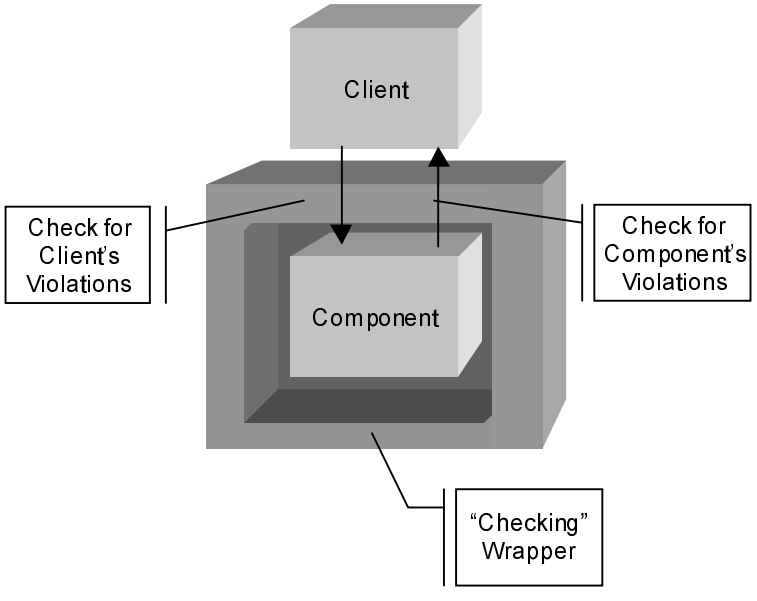

Figure 1—Insulating the Component with a Violation-Checking Wrapper

With this simple but expanded view in mind, attention falls on the structure of the checking wrapper. In particular, is it possible to systematically generate such a wrapper for any component so that the wrapper can be inserted and removed with ease for selected components of a system? Is it possible to automate the construction of such wrappers? Can wrappers check for both internal and external interface violations? Can wrappers implement checks without requiring direct access to the internal representation of the wrapped components—that is, without violating encapsulation? In this paper, we answer all these questions in the affirmative, by presenting a canonical architecture for checking wrappers with these properties. A key dimension of complexity in our solution to this problem results from the objective of handling parameterized template components [1, 7, 17], an essential element of complex systems.

2.2 The violation checking micro-architecture

At the heart of achieving these goals is a carefully designed architecture for the violation checking wrapper. This architecture raises a fundamental dilemma: On one hand, it is desirable to maintain encapsulation, indicating that the wrapper should not have direct access to the component's internal data. On the other hand, it is undesirable to invoke the component's operations directly, since they may repeatedly violate the interface's behavioral contract while the wrapper is in the process of performing its checks. There are three basic design alternatives for addressing this concern.

An obvious design choice is to create a wrapper by moving the implementation-dependent violation detecting code that normally might have been placed inside the base component into a new class. This new class would require direct access to the internal representation of the base component it will wrap. This choice has the obvious benefit of separating the violation detecting code from the base component. It also has multiple disadvantages. The wrapper must directly access the internals of the base component, violating encapsulation. Since the wrapper is representation-specific, it cannot be used for any other components that conform to the same interface. In addition, checks must be implemented in terms of the raw data representation, which might make the checking code itself more error-prone, especially when wrappers are developed by different programmers.

A natural alternative is to build an implementation-independent checking wrapper that is "layered" on the base component without requiring direct access. Instead, it can test for

violations by calling the exported methods of the wrapped component. Such a wrapper has three advantages: it separates the violation checking code, it respects encapsulation, and checking code can be written independent of how the component might be implemented. Unfortunately, since exported methods may themselves cause violations, it is not possible to pinpoint the source of a violation when one occurs. More importantly, using other exported methods to check the expectations of a method may cause state changes in the objects, possibly involving actions that are irreversible or expensive to roll back (e.g., causing files to be opened, physical devices to be controlled, or even nuclear reactor controls to be manipulated).

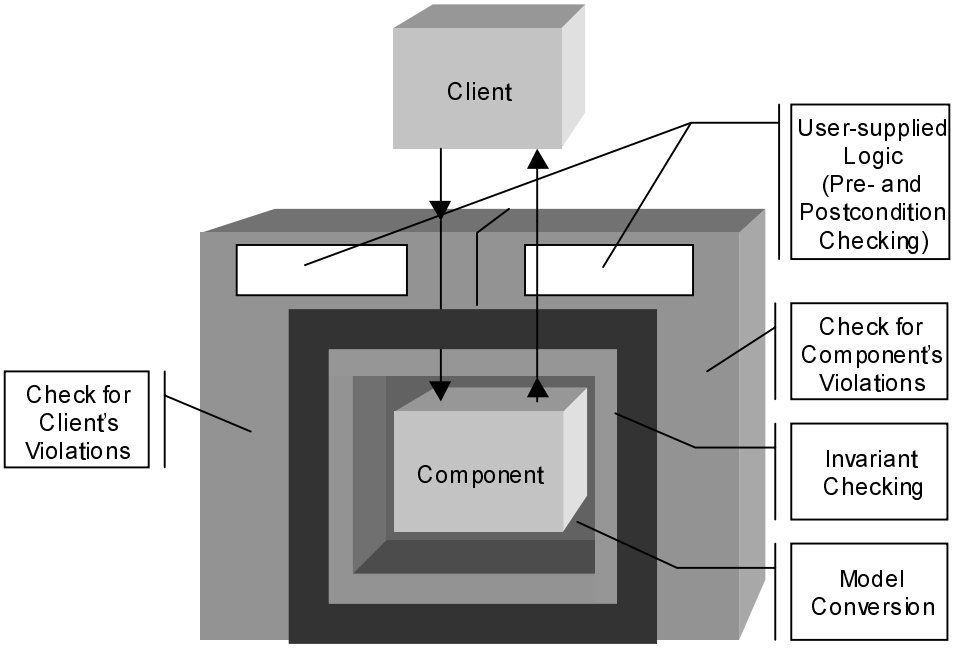

Figure 2-Detailed Wrapper Structure

A third design alternative provides the advantages of the previous two choices, and at the same time addresses their shortcomings. This solution relies on distinguishing abstract and concrete values of objects, and it performs checks by manipulating abstract values of objects, rather than directly working with the internal state stored in the object's implementation. Borrowing from model-based specification techniques, such as those summarized by Wing [19], we can "model" abstract values of objects using mathematical models such as sets, strings (or sequences), functions, tuples, and trees. The core of the scheme for violation checking wrappers is based on the ability to convert from an object's internal representation into an alternative form—one that directly captures the abstract view of the object's state, using the programmatic versions of the mathematical modeling types. The number of fundamental mathematical building blocks is small in some model-based specification languages, such as RESOLVE [16], which rely on composition of the basic models for abstract descriptions of more complex artifacts. As part of the RESOLVE/C++ component collection [7], interfaces and implementations have been provided for program analogues of these mathematical models. A structure of wrappers for this design alternative, in which abstract values and concrete representation values of objects are distinguished, is shown in Figure 2. Section 5.2 explains that this same wrapper structure also works for the earlier design choices, when abstract specifications are not available.

At a wrapper's core in Figure 2, there are two operations that translate an object's internal state to and from an alternative form based on the abstract picture of that state (usually provided as part of the object's specification). These two operations must have direct access to the object's internal state, and are thus provided by the encapsulated component. By converting the internal representation of the object under consideration into an alternative form, it is then possible for the wrapper to examine, manipulate, and test the internal state without violating encapsulation, and without relying on any of the (potentially violation-inducing) methods exported by the component under scrutiny.

The remaining layers shown in Figure 2 (from the inside out) allow for the abstract view of the object's stored state to be checked against any internal representation invariant conditions of the component, to be checked for precondition satisfaction, and to be checked for postcondition satisfaction. For each and every method exported by the wrapped component, the wrapper will follow the steps shown in Figure 3 to detect all possible interface violations. Each parametric object also needs to be converted to and from its model at the beginning and at the end, respectively, though this detail is not explicitly shown in the figure. An interface violation can be detected at any one of the four "check points" shown in Figure 3. The action(s) to be taken if an interface violation is detected can be as simple or as complex as desired,

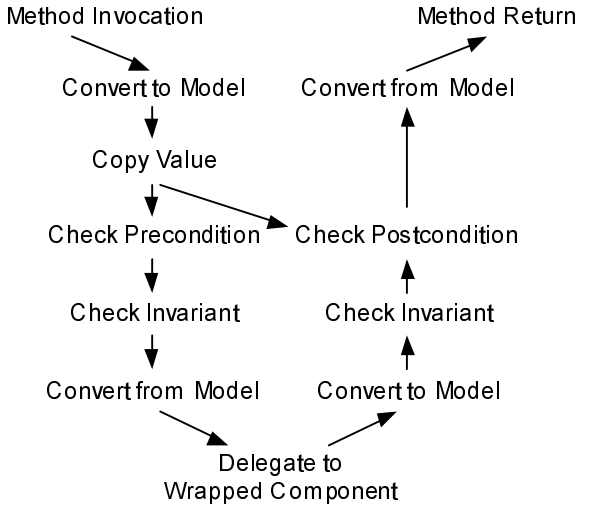

Figure 3—Wrapper's Method Invocation Strategy

and can also be separated from both the component under test and from the violation checking wrapper itself.

The wrapper is most powerful when component interfaces have formally specified mathematical pre- and postconditions for operations, though it is equally applicable when the descriptions are informal. To provide effective support, we have designed and built a generator for C++ components. The power of the generator in easing complexity is obvious when participating components are parameterized templates. Through the separation of abstract interfaces from concrete implementations, it becomes possible to generate checking code in parallel with actual development of the component implementation. When there are multiple implementations of the same abstract description, the same "scaffolding" is suitable for all of the different implementations.

3 An example

To illustrate the approach for detecting interface violations, we present a "prioritizer"—a component that orders a collection of items [3]. This component is parameterized by the type of entries to be placed in the prioritizer, the ordering used, and an upper bound on the

```
template <class Entry, int Max_Size>
// Class Entry should include a "Compare"
// operation that implements a total
// preordering R(x, y)
class Prioritizer {
  //! type family Prioritizer is
  //! modeled by (
  //!     entries : multiset of Entry,
  //!     insertion_phase : boolean
  //!   )
  //!   constraint |entries| <= Max_Size
  //!   initialization
  //!     ensures |entries| = 0 and
  //!             insertion_phase = true
public:

  Prioritizer () {};
  virtual ~Prioritizer () {};

  // swap operator to be added in each
  // implementation
  // virtual void operator &= (
  //     Prioritizer & rhs) = 0;

  virtual void Insert (Entry& x) = 0;
    //! requires  |entries| < Max_Size
    //!        and insertion_phase = true
    //! ensures   entries = #entries
    //!                     union {#x} and
    //!           insertion_phase = true
  virtual void Change_Phase () = 0;
    //! ensures   entries = #entries and
    //!           insertion_phase =
    //!              not #insertion_phase
  virtual void Remove_Next (Entry& x) = 0;
    //! requires  |entries| > 0 and
    //!           insertion_phase = false
    //! ensures   for all y: Entry where
    //!       (entries.occurrences(y) > 0)
    //!           ( R(x, y) ) and
    //!  #entries = entries union {x} and
    //!       insertion_phase = false
  virtual void Remove_Any (Entry& x) = 0;
    //! requires  |entries| > 0
    //! ensures   #entries = entries
    //!                     union {x} and
    //!           insertion_phase =
    //!                   #insertion_phase
  virtual Integer Size () = 0;
    //! ensures   Size = |entries|
  virtual Boolean Is_In_Insertion_Phase ()
                      = 0;
    //! ensures   Is_In_Insertion_Phase =
    //!                   insertion_phase
  virtual void Clear () = 0;
    //! ensures   |entries| = 0 and
    //!           insertion_phase = true
  static void Allowed_Max_Size () = 0;
    //! ensures   Allowed_Max_Size =
    //!                           Max_Size

private:
  // Assignment and copy construction
  // are prohibited
  Prioritizer & operator = (
               const Prioritizer& rhs);
  Prioritizer (const Prioritizer& m);
};
```

number of entries. The conceptual interface can be captured as a C++ abstract base class template called `Prioritizer`, as shown in Figure 4. Alternative implementations can be provided as additional C++ class templates that inherit from this interface (and that might have additional template parameters, as necessary).

Operation	Valid Inputs	Expected Outputs
Insert	p = ({2,4, 9}, true) x = 4	p = ({2,4,4,9}, true) x = ?
Change_Phase	p = ({2,4,4,9}, true)	p = ({2,4,4,9}, false)
Change_Phase	p = ({2,4,4,9}, false)	p = ({2,4,4,9}, true)
Remove_Next	p = ({2,4,4,9}, false) x = ?	p = ({4,4,9}, false) x = 2

Figure 5—Sample Inputs/Outputs for a Prioritizer

To minimize the cost to understand and reuse objects such as prioritizers, an abstract explanation of the concept's behavior must be provided for reusable C++ template classes. This can be done using a formal specification notation, such as RESOLVE [16], or it may be described informally. Briefly, a prioritizer is a bag of items that has a "remove next (in order)" operation—similar to a priority queue, except that it does not necessarily maintain the order in which items are added in case of ties. Further, it operates in one of two phases: an insertion phase, where items are being added, and an extraction phase, where items are being removed (Weide *et al* describe phase-based objects that embody algorithms such as sorting [18]).

More precisely, the contents of a prioritizer object can be viewed as an ordered pair: a *multiset* of entries contained within the prioritizer and a *boolean* flag that denotes whether the object is in insertion or extraction phase. Note that this is a *mathematical* model of the abstract value of any prioritizer object, irrespective of how that state might actually be recorded (i.e., in a heap, or in a binary tree, or in an unsorted list). Initially, a newly declared prioritizer object is empty and is in insertion phase. New items can be added to the collection, one at a time, using the `Insert` operation, which requires the object to be in the insertion phase. The `Remove_Next` operation requires the object to be in the extraction phase, and it can be called to extract items one at a time based on the specified ordering. The `Change_Phase` operation toggles the phase of the prioritizer. The `Remove_Any` operation removes an arbitrary item from the collection without regard for order, and can be called in either phase. The explanations of other operations are provided in Figure 4. The prioritizer interface in Figure 4 has been designed following the RESOLVE/C++ discipline [7]; inclusion of a swap operator (in C++, `operator &=` has been borrowed for this operator), and explicit prohibition of copy construction and assignment are among the standards for object design in this discipline.

Figure 5 contains sample inputs and outputs for a prioritizer that contains integers, an instance of a concrete class that conforms to the `Prioritizer` interface. These inputs and outputs respect the prioritizer interface conventions—no preconditions are violated by the client, and the expected outputs are those that would be produced by any conforming prioritizer implementation. Notice that the inputs and outputs have been expressed in terms of mathematical models of the objects (e.g., an ordered pair for prioritizers). In Figure 5, the value of `x` after calling `Insert` is not specified by the prioritizer interface; the question mark indicates that no specific value can be expected (or is required).

4 Model conversion operations

Alternative implementations of the prioritizer interface may use different internal representations such as arrays, heaps, lists, trees, or pointer structures. In addition, each implementation may use different conventions for ordering: one implementation might always keep its items in sorted order internally (or in a suitable heap), regardless of phase; another might perform a batch-style sort when `Change_Phase` is invoked to go from insertion to extraction phase; a third might never keep its items sorted internally, and simply find the next item in order only when `Remove_Next` is called. There are many possible design choices, and prioritizer interface was designed to allow for many alternative ways of spreading work (and amortizing cost) among the various operations [3, 18]. For detecting interface violations, fortunately, these implementation-dependent details can and should be ignored. Instead, internal representation values must be converted to the more abstract values in the interface specifications, and the satisfaction of pre- and postconditions on those abstract values can be checked before and after each operation call.

To facilitate conversion of internal representation values to abstract values, each class must be enhanced with two additional operations, `Convert_To_Model` and `Convert_From_Model`. In converting representations of generic objects such as prioritizers to their abstract models, objects of the parametric type `Entry` also need to be converted to their models, because these entries may be non-trivial objects. The `Prioritizer_With_Model` class that extends `Prioritizer` with the two model conversion operations is shown in Figure 6. This template has four parameters:

```
template <class Entry_With_Model,
      int Max_Size, class Model_Of_Entry,
      class Model_Of_Prioritizer>
class Prioritizer_With_Model :
    public Prioritizer <Entry_With_Model,
                        Max_Size> {
public:
    Prioritizer_With_Model () {};
    ~Prioritizer_With_Model () {};
    virtual void Convert_To_Model (
            Model_Of_Prioritizer& s) = 0;
    virtual void Convert_From_Model (
            Model_Of_Prioritizer& s) = 0;
};
```

Figure 6—C++ Interface for Prioritizer_With_Model

`Entry_With_Model` is the kind of item that will be placed in the prioritizer. As the parameter name suggests, this class should support both model conversion operations for entries.

Max_Size is the maximum number of items that can be placed in the prioritizer.

`Model_Of_Entry` is the programmatic version of the abstract model of an entry's value. The two conversion operations supported by `Entry_With_Model` map to and from this type.

`Model_Of_Prioritizer` is the programmatic version of the abstract model of a prioritizer—a pair containing a multiset and a boolean.

This interface can be generated automatically, though, in general, its implementation cannot be.

In developing an implementation of model conversion operations, "model" objects need to be created and manipulated within the program. To convert a prioritizer to its abstract value of a multiset of entries and a boolean, it becomes necessary, for example, to be able to insert entries into multisets. To facilitate this process, we have created C++ template implementations for standard models—integers, sets, functions, strings, relations, and tuples used in RESOLVE specifications [15]. For violation checking of Priortitizer_Template, the `Multiset_Model` template, which has programmatic operations such as `Union_Entry`, `Remove_Entry`, `Is_Element_Of`, `Are_Equal`, and `Copy_To` to manipulate multisets of entries, is used.

Assuming that a reliable set of implementations for standard mathematical modeling types is available, and that all objects are enhanced with reliable model conversion operations, most routine aspects of interface violation detection can be automatically generated as explained below. The complexities in the implementations and generation of the implementations mainly result from handling templates (parameterized components).

5 Generating wrapper components for detecting interface violations

To provide support for creating and using checking wrappers, we have designed and built a generator that uses RESOLVE-style component specifications and C++ template interfaces to produce one- and two-way checking components [15]. The underlying principles for creating such wrappers are independent of any particular specification technique or implementation language, and they can be readily extended to other languages.

The public, syntactic interface of a "checking" wrapper component is identical to that of the corresponding base component. Semantically, they differ in how they behave when either the pre- or postcondition of an operation is violated. In particular, where a regular component guarantees nothing if an operation is invoked under conditions violating its precondition, a checking wrapper instead guarantees it will perform a specific notification action. We call a wrapper that only checks for precondition violations a *one-way checking* wrapper.

Similarly, a *two-way checking* wrapper guarantees to:

1. Carry out its precondition notification action if the precondition does not hold, or
2. Establish that the postcondition is true upon operation completion, or
3. Carry out its postcondition notification action if the postcondition does not hold.

Both one-way and two-way checking wrappers are extremely useful. One-way wrappers correspond with the traditional notion of a "defensive shell" that protects a component from errant clients. Two-way wrappers, on the other hand, are more akin to "self-checking" or "self-verifying" components that confirm their own work as well as spotting erroneous client behavior. For brevity, we will focus on two-way checking wrappers in the remainder of our discussion; the corresponding features of one-way checking wrappers are easily extrapolated.

5.1 Two-way checking wrapper implementation

Essentially, wrapping a component inside its corresponding checking wrapper leads to reporting of both client and component interface violations in terms of abstract values. Figure 7 shows the interface of the `Two_Way_Checking_Prioritizer` template that is produced by our generator [15]. The generated interface includes in its template parameters the mathematical model classes of the object and its parametric entry to facilitate specification-based violation checking.

A `Two_Way_Checking_Prioritizer` object is layered on top of a `Prioritizer_With_Model` ob-

ject of the type passed in as its Wrapped_Prioritizer parameter, as seen from the template's private section. Templates of this form—which export one interface and take as a template parameter another component exporting the same interface—are called symmetric components in GenVoca-based systems [1].

Under violation-free conditions, the implementations of each operation in the Two_Way_Checking_Prioritizer just call through to the wrapped object's corresponding operation. To allow for violation detection logic to be separated from the wrapper itself, the generated template is parameterized by a class that contains the code for checking the pre- and postconditions of each operation. The class Two_Way_Checkers_For_Prioritizer contains operations such as Insert_Precondition and Insert_Postcondition for every operation, as well as Invariant for confirming that the class invariant holds. Given this basis, generating implementations for each of the two-way checking wrapper's operations is straightforward. The checking version of each operation *P*, makes the sequence of calls dictated by Figure 3 before calling through to (the non-checking version of) *P* provided by the wrapped component. Figure 8 contains the implementation for one operation provided by the two-way checking wrapper, Insert.

Just as the violation detection logic can be decoupled from the wrapper by means of a template parameter, the notification action can also be separated from the wrapper. The current version of the generator uses simple error message printing as the action to perform whenever a violation is detected, although in practice any user-defined action can be triggered. Using a template parameter to separate the action code from the wrapper is a convenient way to provide such flexibility while still maintaining a simple wrapper design.

Of course, before inputs or outputs can be validated, they must be converted to model values; after checking, they should be restored to representation values. In checking that outputs are valid, both inputs and outputs to the operation typically are needed, since postconditions normally describe the output values of parameters relative to their values before the call. For this reason, each two-way checking operation in the wrapper needs to make copies of the abstract parameter values before calling through to the operation in the base component.

For most components, checking each precondition is straightforward and can thus be automated. By using the model conversion approach, the majority of precondition statements can be converted to code by a simple transliteration process, using the operations provided by the programmable version of the mathematical modeling types. With the prioritizer interface, for example, all of the preconditions involve simple tests of the insertion phase flag and the multiset's size. As a result, it is often practical to

```
template <class Entry_With_Model,
  int Max_Size, class Model_Of_Entry,
  class Model_Of_Prioritizer,
  class Wrapped_Prioritizer,
  class Two_Way_Checkers_For_Prioritizer>
class Two_Way_Checking_Prioritizer :
  public Prioritizer<Entry_With_Model,
                     Max_Size> {
public:
  Two_Way_Checking_Prioritizer ();
  virtual
      ~Two_Way_Checking_Prioritizer();
  virtual void operator &= (
     Two_Way_Checking_Prioritizer& rhs);

  virtual void Insert (
      Entry_With_Model& x);
  virtual void Change_Phase ();
  virtual void Remove (
      Entry_With_Model& x);
  virtual void Remove_Any (
      Entry_With_Model& x);
  virtual Integer Size ();
  virtual Boolean
      Is_In_Insertion_Phase ();

private:
  Two_Way_Checking_Prioritizer&
    operator = (const
      Two_Way_Checking_Prioritizer&);
  Two_Way_Checking_Prioritizer (
    const Two_Way_Checking_Prioritizer&);

  Wrapped_Prioritizer       aPrioritizer;
  Two_Way_Checkers_For_Prioritizer check;
};
```

Figure 7—Generated C++ Interface for Two_Way_Checking_Prioritizer

automatically generate the code for a one-way checking wrapper (e.g., a One_Way_Checkers_For_Prioritizer class). Checking postconditions, on the other hand, is usually non-trivial. As a result, two-way checkers cannot be generated mechanically [2]. A future version of the generator is expected to include some automation of checking code.

Components that use prioritizer objects can opt to use an unwrapped prioritizer implementation, use a One_Way_Checking_Prioritizer, or use a Two_Way_Checking_Prioritizer. This choice determines whether or not client and/or component violations are automatically detected. Switching between these three alternatives only involves changing object declarations; no changes are needed to the code of the client components, as illustrated in Figure 9. Further, intelligent use of named subtypes (or typedefs) can easily provide a single point of control for this decision. Of course, instantiating one- and two-way checking components requires

```
template <class Entry_With_Model,
  int Max_Size, class Model_Of_Entry,
  class Model_Of_Prioritizer,
  class Wrapped_Prioritizer,
  class Two_Way_Checkers_For_Prioritizer>
void Two_Way_Checking_Prioritizer
  <Entry_With_Model, Max_Size,
   Model_Of_Entry, Model_Of_Prioritizer,
   Wrapped_Prioritizer,
   Two_Way_Checkers_For_Prioritizer>::
Insert (Entry_With_Model& x) {
   Model_Of_Prioritizer old_p, new_p;
   Model_Of_Entry       old_x, new_x;

   aPrioritizer.Convert_To_Model (old_p);
   old_p.Copy_To (new_p);

   x.Convert_To_Model (old_x);
   old_x.Copy_To (new_x);

   if (!check.Insert_Precondition (
            old_p))
      cerr << "Precondition for "
           << "Insert violated.\n";
   else {
      if (!check.Invariant (old_p))
         cerr << "Invariant violated "
              << "entering Insert.\n";

      aPrioritizer.Convert_From_Model(new_p);
      x.Convert_From_Model (new_x);
      aPrioritizer.Insert (x);

      aPrioritizer.Convert_To_Model(new_p);
      x.Convert_To_Model (new_x);

      if (!check.Invariant (new_p))
         cerr << "Invariant violated "
              << "exiting Insert.\n";
      if (!check.Insert_Postcondition (
             new_p, old_p, new_x, old_x))
         cerr << "Ensures Condition For "
              << "Insert Violated\n";
   }
   aPrioritizer.Convert_From_Model (new_p);
   x.Convert_From_Model (new_x);
}
```

Figure 8—Generated C++ Implementation for Two_Way_Checking_Prioritizer::Insert

additional template parameters that must be defined; these changes can be mechanized, however, though the current implementation of the generator does not do so.

5.2 Representation-dependent and representation-independent conversion and checking

Section 2.2 outlined three design alternatives for wrapper-based violation checking. Although our focus has been on the third design choice involving abstract and concrete values together with model conversion, we conclude this section by explaining how the alternative design choices can be applied using our framework.

To support the "model-based" construction of checking wrappers, it is essential to have components that permit the programmatic manipulation of common mathematical models used in the behavioral descriptions (specifications). To the extent that it is important to correctly identify the method that caused a particular violation, or to avoid unnecessary manipulation of the underlying object, the model conversion approach is the only effective choice. Where these issues are less critical, other choices for wrapper architectures are suitable.

To use the simpler "layered" approach to checking components (the second design choice discussed in section 2.2), violation checking code would need to directly invoke prioritizer operations such as `Remove_Any` to inspect the contents of a prioritizer object. Similarly, to restore the object's value after inspection, `Insert` and other prioritizer operations would be called.

Though the power of the approach presented in this paper stems from the ability to separate complex internal implementation details from more abstract interface violation checking, it is also possible to use the framework where such a distinction is absent. This is indeed the case with the first (and simplest) wrapper design choice in Section 2.2. Suppose that instead of abstract pre- and postconditions, assertions are expressed (formally or informally) directly in terms of the internal representation values of all participating objects, and that checks are performed directly on those internal values. In our framework, this situation can be viewed as the degenerate case in which the abstract and concrete representation spaces are identical and the model conversion operations are identity transformations. In this case, the approach reduces to representation-dependent violation detection,

```
// Change type declarations; used to be:
// typedef Prioritizer_1 < . . . >
//     Student_Record_Prioritizer;
typedef Two_Way_Checking_Prioritizer <
    Student_Record, . . . >
        Student_Record_Prioritizer;

// No change needed to declarations
Student_Record_Prioritizer p;

//No change needed to method invocations
......
p.Insert(r);
```

Figure 9—Changes to Calling Components

except that the generated wrapper components provide a mechanism for separating checking code from the component code. It is important to note that for template components, it is usually not possible to assume access to the internal representations of objects of a parametric type; in such a case, the framework can be used to combine representation-dependent and representation-independent strategies. In this sense, the present proposal for interface violations is quite general.

6 Comparison with previous work

The responsibilities for interface violations in our framework is probably best explicated by Bertrand Meyer as the design-by-contract notion [9]: pre-conditions of operations are obligations on callers and post-conditions are obligations on implementers who may assume that the pre-conditions hold at the time of invocation. We have long followed this approach, and its roots go back at least as far as Parnas [11]. Arguments for making pre-conditions the responsibility of the caller are compelling because the intent is to define the valid domain of inputs and allow implementers to assume that the conditions hold on inputs. However, this assignment of responsibilities is neither obvious nor universal. Liskov and Guttag [8] and Perry [13], for example, propose different assignments of responsibilities for handling interface violations between software modules, whereby implementers of modules are responsible for detecting some or all interface violations.

The contribution of this paper is a framework that allows interface violation checking code to be kept separate from, and therefore independent of, both the client's and the component's code. Using templates, the framework assigns the detection of violations to separate, highly reusable, software components which can easily be incorporated or not in plug-and-play fashion.

The assertion checking approach described by Rosenblum [14] and the Annotation Pre-Processor (APP) that implements the approach for C programs have much in common with the objectives of our framework. The APP allows assertions to be embedded in programs—including pre- and postconditions that need to be expressed in an implementation-independent form as executable C code—and combines separately provided code for checking those assertions to produce "checking" versions of code. The assertions can be assigned different levels of importance, and the tool can be instructed to include assertion checking at selected levels. Unlike our framework, however, the APP approach does not distinguish abstract and concrete values of objects and in this sense is an implementation of the first design choice presented in Section 2.2. In addition to the use of mathematical models, our framework also differs from and is complementary to the APP in that our focus is at the level of object-based components and that the violation checking code is separated through template parameters.

Eiffel provides another well-known approach for pre- and postcondition checking at runtime [10]. Unlike our generator, Eiffel also relies on the compiler to generate violation detection code directly from pre- and postconditions. To make this possible, Eiffel necessarily requires the captured conditions to be simple, at the risk of impreciseness. For example, in specifying the `Insert` operation of `Prioritizer` in Eiffel, the most likely postcondition is that the size of the prioritizer would be one larger after calling `Insert`. Missing in this description is the important part of the postcondition that the prioritizer after the operation contains all the items it had before plus the newly added item. Complex assertions such as these cannot easily be phrased in Eiffel (note the quantifier in Figure 4), and automatically generating code to confirm such behavior is beyond the capacity of that language. Also in Eiffel, assertions typically are expressed and checked using the second design choice discussed in Section 2.2; the limitations of this design decision include calling potentially incorrect operations in checking the correctness of other operations.

The checking wrapper approach and the framework presented here also significantly differ from approaches for specification-based testing of object-based and procedural software components in the literature [2, 4, 6, 12]. While few approaches for testing distinguish abstract and concrete values of objects, most of them do not address the issues of separating violation checking code or use of templates in handling such separation.

7 Status and conclusions

The interface violation detection strategy described here is applicable anywhere defensive components are appropriate. In particular, one-way and two-way checking wrappers facilitate both unit and integration testing, as well as debugging [15]. By encasing selected components inside checking wrappers, interface conformance can be checked dynamically at run-time, allowing interface violations to be systematically identified and reported. Further, using the strategy described here, it is possible to generate checking wrappers in parallel with (or even prior to) component implementation, since only the component interface is required. After verification and validation are complete, checking wrappers can be easily removed where appropriate.

Though we have concentrated on a single component in this paper, the significance of the approach becomes apparent when it is employed in component-based systems. When a new component is added to an existing system, it is encased in a two-way checking wrapper. Any defects in the interactions between the system and the new component are revealed by the wrapper. After a

suitable time has elapsed and the new component becomes more trustworthy, the two-way wrapper can be exchanged for a one-way wrapper to detect any misuses by additional components introduced in the future. This iterative process continues until the system is ready to be shipped, at which time all of the remaining wrappers are removed (except maybe for one-way checking wrappers for components that are in direct contact with users to report user violations.)

The generator discussed in this paper currently handles RESOLVE specifications as inputs and generates C++ template code as output. We have applied the generator on several layered RESOLVE/C++ template components. Since the underlying principles are independent of both the specification and implementation languages, similar tools can be easily developed for other languages. For the interested reader, the full text of the sample component presented here, as well as the wrapper templates created by the generator for single components and systems, have been made available on-line at `http://www.cs.wvu.edu/~resolve`/wrappers. Anecdotal evidence indicates that using these checking components during unit development, unit testing, system integration and system testing contributes significantly to revealing defects and to drastically reducing the effort required for system integration and system testing.

Modern programming languages such as Ada, C++, and Java encourage separation of interfaces from implementations and construction of layered, object-based templates. Typical systems built in these languages contain components developed by multiple vendors, bringing into question the reliability of the overall system. In this paper, we have described a systematic approach for identifying and isolating interface violations in such component-based systems. The proposed approach is designed to take advantage of situations when abstract interface specifications are available, but works equally well in their absence. Implementation and experimentation with a generator for producing violation-checking wrappers confirm the practical benefits of the approach.

Acknowledgements

We gratefully acknowledge the financial support from our own institutions, from the National Science Foundation under grant CCR-9311702, from the Defense Advanced Research Projects Agency under project number DAAH04-96-1-0419 monitored by the U.S. Army Research Office, and from NASA under grant NCC 2-979. Any opinions, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or the U.S. Department of Defense.

References

[1] D. Batory and B.J. Geraci, "Composition Validation and Subjectivity in GenVoca Generators," *IEEE Trans. Software Engineering*, Feb. 1997, pp. 67-82.

[2] B. Bennett and M. Sitaraman, "Validation of Results in Testing Abstract Data Types: A Method for Automation," *Proc. First Int'l Conf. on Software Quality*, Dayton, Ohio, Oct. 1991.

[3] D. Fleming, M. Sitaraman, and S. Sreerama, "A Practical Performance Criterion for Object Interface Design," *J. Object-Oriented Programming*, Jul./Aug. 1997, pp. 52-63.

[4] P. Frankl and R. Doong, "The ASTOOT approach to testing object-oriented programs," *ACM Trans. Software Engineering Methodology*, 1994, 3(2), pp. 101-130.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[6] D. Hoffman and P. Strooper, "The Test-graphs Methodology: Automated Testing of Classes," *J. Object-Oriented Programming*, Nov. 1995.

[7] J. Hollingsworth, S. Sreerama, B.W. Weide, and S. Zhupanov, "RESOLVE Components in Ada and C++," *ACM SIGSOFT Software Engineering Notes*, Oct. 1994, pp. 52-63.

[8] B. Liskov, and J. Guttag, *Abstraction and Specification in Program Development*, McGraw-Hill, 1986.

[9] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, New York, 1988.

[10] B. Meyer, *Object-Oriented Software Construction*, 2nd Edition, Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.

[11] D.L. Parnas, "A Technique for Software Module Specification with Examples," *Comm. ACM*, May 1972, pp. 330-336.

[12] A. Parrish and D. Cordes, "Applying Conventional Unit Testing Techniques to Abstract Data Type Operations," *Int'l J. Software Engineering and Knowledge Engineering*, Mar. 1994, pp. 103-122.

[13] D. E. Perry, "The Inscape Environment," *Proc. 11th Intl. Conf. On Software Eng.*, IEEE CS Press, May 1989, pp. 2-12.

[14] D. S. Rosenblum, "A Practical Approach to Programming with Assertions," *IEEE Trans. Software Engineering*, Jan. 1995, pp. 19-31.

[15] G. Shakir, *A Systematic Generator for Detecting Interface Violations in Component-Based Software*, M.S. Report, Dept. of Comp. Sci. and Elect. Engrg., West Virginia Univ., Morgantown, WV 26506, 1997.

[16] Special Section: Component-Based Software Engineering Using RESOLVE, Eds. M. Sitaraman and B. W. Weide, *ACM SIGSOFT Software Engineering Notes*, Oct. 1994, pp. 21-67.

[17] A. Stepanov, "The Standard Template Library," *BYTE*, Oct. 1995, pp. 177-178.

[18] B.W. Weide, W.F. Ogden, and M. Sitaraman, "Recasting Algorithms to Encourage Reuse," *IEEE Software*, Sept. 1994, pp. 80-88.

[19] J. M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, Sept. 1990, pp. 8-24.