Contract-Checking Wrappers for C++ Classes

Stephen H. Edwards, Murali Sitaraman, *Member*, *IEEE Computer Society*,
Bruce W. Weide, *Senior Member*, *IEEE*, and Joseph Hollingsworth

Abstract—Two kinds of interface contract violations can occur in component-based software: A client component can fail to satisfy a requirement of a component it is using, or a component implementation can fail to fulfill its obligations to the client. The traditional approach to detecting and reporting such violations is to embed assertion checks into component source code, with compile-time control over whether they are enabled. This works well for the original component developers, but it fails to meet the needs of component clients who do not have access to source code for such components. A wrapper-based approach, in which contract checking is not hard-coded into the underlying component but is "layered" on top of it, offers several relative advantages. It is practical and effective for C++ classes. Checking code can be distributed in binary form along with the underlying component, it can be installed or removed without requiring recompilation of either the underlying component or the client code, it can be selectively enabled or disabled by the component client on a per-component basis, and it does not require the client to have access to any special tools (which might have been used by the component developer) to support wrapper installation and control. Experimental evidence indicates that wrappers in C++ impose modest additional overhead compared to inlining assertion checks.

Index Terms—Assertion checkers, binary components, design by contract, preconditions, postconditions, class invariants, coding techniques, debugging aids, specification.

1 Introduction

A fundamental goal of modern software engineering is to enable predictable and modular construction of software systems by assembling components. Bertrand Meyer's design-by-contract principles [20], [21] lay out a clear division of responsibilities between a component implementation and client code that uses it. A contract delineates what each party may assume and what each party is obligated to ensure. A ***contract violation*** occurs when one party does not live up to this contract. Being behavioral rather than syntactic in nature, a contract violation generally is not detected until runtime. When integration testing reveals that a component's contract with a client is violated, assigning responsibility for it can demand a substantial investment of debugging effort. Not detecting such a violation until after deployment can be even more expensive. Worse still, a system can behave properly on all test cases even as internal interface contracts—those where the component plays the role of client to its subcomponents—are violated. Failures resulting from such violations are revealed only in the form of accidents after deployment. The benefits of using runtime assertion checks to enable early detection of contract violations are, therefore, substantial and well-known [32], [35], [21], [10]. As component-based approaches gather momentum, runtime checking can address some of the risks faced by component clients [33].

Szyperski defines a component as "a unit of composition with contextually specified interfaces and explicit context dependencies only...[and that] can be deployed independently and is subject to third-party composition" [30], and this is the definition we use in this paper. However, to make the presentation of contract-checking concerns concrete, we consider only checking the behavior of classes in C++ rather than components in the more general setting of technologies such as CORBA, EJB, or .NET. Regardless, the most important issues involved in packaging and managing assertion-checking code for components remain the same. In particular, the critical concerns exposed when distributing components in compiled form noted by Szyperski [30] also arise in the case of C++ class libraries distributed in compiled form.

1.1 Contract Checking Revisited

The traditional approach to contract checking is to include assertion checks within the source code for component methods. This is useful for *component developers*—those who have direct access to component source code and can easily recompile it with alternative assertion-checking settings. Indeed, it is common when using assertion checking to enable checking code during development and testing, and to disable or remove it before delivery [21], [17], [18].

The primary limitation of this approach impacts not component developers, but *component clients*, when a component is distributed in compiled form only. When assertion checks are embedded directly within the component being checked, either the client must relinquish the benefits of assertion checks, or the client must pay some runtime penalty for the checks because there is no option to recompile without checks to eliminate this overhead.

- *S.H. Edwards is with the Computer Science Department, Mail Stop 0106, Virginia Tech, Blacksburg, VA 24061. E-mail: edwards@cs.vt.edu.*
- *M. Sitaraman is with the Computer Science Department, Clemson University, Clemson, SC 29634. E-mail: murali@cs.clemson.edu.*
- *B.W. Weide is with the Computer Science and Engineering Department, The Ohio State University, Columbus, OH 43210. E-mail: weide.1@osu.edu.*
- *J. Hollingsworth is with the Computer Science Department, Indiana University Southeast, New Albany, IN 47150. E-mail: jholly@ius.edu.*

Manuscript received 3 Mar. 2004; revised 22 Aug. 2004; accepted 1 Sept. 2004.
Recommended for acceptance by J.-M. Jezequel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0033-0304.

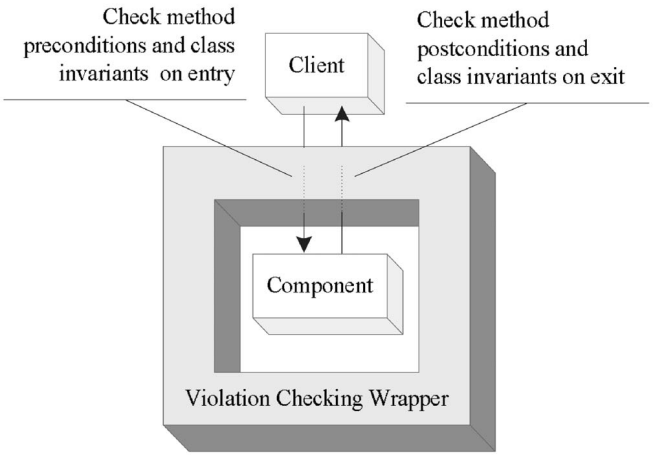

Fig. 1. A wrapper surrounds the component, implementing all necessary assertion checking.

The approach described in this paper provides runtime precondition, postcondition, and invariant checks by using ***contract-checking wrappers***. Rather than placing assertion-checking code inside the component, such a wrapper (also called a decorator [12]) isolates checks in a separate layer between the component and its client(s), as shown in Fig. 1. The wrapper's sole responsibility is to implement interface violation checks.

A contract-checking wrapper provides the same interface as the component it checks, and delegates the work involved in carrying out each operation to that underlying component. The wrapper performs runtime checks both before and after each method invocation in order to detect interface violations. The concept of using wrappers is a strategy that fully separates the assertion checks from *both* the client's code and the underlying component's code. The fundamental principles on which this approach is based are presented in the form of essential requirements for contract checking, in Sections 1.2 and 1.3.

1.2 Contract-Checking Requirements of Component Developers

To see why the traditional approach to contract checking is a good idea, yet still inadequate, it helps to identify the requirement it was designed to satisfy—that of a component developer.

Requirement #1: The contract-checking approach should allow checking code to be inserted or removed (e.g., for development or for deployment, respectively) without editing source code.

Meyer [21] argued persuasively against hard-coding assertion checks into code—either by incorporating checking code in the component *or* by incorporating it into the client. Hard-coding an assertion check inside a component forces the check to be executed, with its consequent performance penalty. Placing checking code in the client causes its proliferation (i.e., at the site of every call) as well as unnecessarily coupling the client to the component being checked.

Meyer's solution in Eiffel was to add linguistic support to allow checking code (automatically generated from an Eiffel executable assertion) to be automatically inserted into, or omitted from, the generated object code for the component at compile time, and at the developer's discretion. This strategy typifies virtually all mainstream runtime assertion checking techniques. Assertion checks are conditionally inserted into the component being checked based on a compile-time switch. Popular use of `assert()` macros in C++ and similar utility classes in Java are much the same. Such a strategy provides all the support necessary when one views assertion checks as a developer-only tool to be used before the commercial release of an application.

Requirement #2: At a minimum, the approach should allow categories of checks (e.g., preconditions, postconditions, or invariants) to be selectively enabled or disabled for each class individually.

This requirement provides fine-grained control over assertions during development. Many existing tools support selective enabling of assertions by allowing the developer to select which categories of assertions will be inserted into a given class and letting the developer recompile with alternate options to change such a decision. It is even possible to conceive of checking strategies that allow much finer control over assertion behavior, such as controlling individual assertions on individual methods to be enabled on a per-object basis. While it is feasible to implement such capabilities, there may be a point of diminishing returns where control at too fine a scale actually makes specifying which features are enabled or disabled more cumbersome than the benefits gained. As a result, Requirement #2 defines a minimum level of control that is comparable to that in other developer-oriented tools [25], including iContract [15], Eiffel [21], the low-level assertion facilities in the Java Virtual Machine and Microsoft's Common Language Infrastructure [32], and logging tools such as log4j [13].

1.3 Contract-Checking Requirements of Component Clients

Two results of the success of modern component technologies raise new requirements for a contract-checking approach. First, commercial components often are distributed in binary form [30], both for simplicity of deployment and for protection of intellectual property and trade secrets. Second, component clients often use components they acquire from others to build and deliver new components, not just end-user applications. A more comprehensive approach to contract checking should, therefore, seek to satisfy the needs of component clients, too.

Requirement #3: The contract-checking approach should not require recompilation of the component(s) being checked, the contract-checking code, or client code.

It is clear that a component client should not be required to recompile a component or contract-checking code that is not provided in source code form. Additionally, however, this client might be developing his/her own higher-level components that are not to be distributed in source code

form. Some methods of these new components might, in fact, be methods of the lower-level components (e.g., through inheritance). In other words, the contract-checking approach should scale up uniformly and naturally from lower-level components through higher-level components—implying that client code, too, should not require recompilation in order to enable/disable contract-checking features.

For contract checking to provide the greatest added value commercially, the component client also should be able to selectively control which assertion-checking features are in use and which are suppressed (Requirement #2) without recompilation. This will allow clients to benefit from checks where checking is appropriate, yet not incur any unwanted overhead where checking is not appropriate.

Requirement #4: The client should be able to control which action(s) are taken in response to detected contract violations.

Again, to provide the greatest added value to the component client, the developer should not hard-code a fixed response to failed assertion checks, even if that response is sufficient during component development. Instead, the client should have the flexibility to choose an action appropriate to his/her use of the checks. Currently, most runtime assertion checking strategies either halt a program with a diagnostic message or throw an exception. Halting the program provides no flexibility to the client. Throwing an exception, on the other hand, requires the client either to ignore the exception (typical result: halting the program), or to write exception-handling code within the client. Including assertion-specific exception handling code within the client makes it difficult to remove or disable assertion-related code from a final product unless conditional compilation or a similar technique is used (conflicting with Requirement #3). Ideally, the client should be able to choose among responses to a detected violation—e.g., throwing an exception, halting with a diagnostic message, popping up an informational dialog, transferring to a debugger, logging to a file or logging via an offsite connection and continuing execution, or any other response.

Requirement #5: The contract-checking approach should not require the client to use special support tools that might have been used by the component developer.

Among the many approaches to packaging and managing runtime assertion checks, most rely on a specific development tool. Certainly, special tools are powerful (and perhaps even essential) in effectively *producing* contract-checking code—e.g., automatically generating checks from behavioral descriptions. Once the checking code is compiled, however, the process by which customers receive, enable, disable, install, or remove runtime checking support should not depend on their use of such a special tool that might be available only to the component developer and not to every client.

1.4 Contributions

Together, these five requirements—which we show how to meet in the context of C++ components—capture the fundamental design goals of our contract-checking approach. At the same time, they help delineate how this approach differs from previous work. Many existing techniques [21], [17], [18] support Requirements #1 and #2. The benefits of Requirement #4 are explained in [3] and are supported by the Annotation Pre-Processor [26]. But, earlier techniques require recompilation to add or remove assertions (against Requirement #3), and use a special preprocessor or a separate phase of a compiler to conditionally insert checks in the underlying component (against Requirement #5). Moreover, although several researchers have used wrappers around individual methods [7], [18], [11], all such approaches have continued to place the checks in the original class rather than in a separate class where it can be managed more easily by clients. The added value of contract-checking wrappers as implemented here lies in their ability to meet Requirements #3-5 above, in addition to Requirements #1-2.

It is also worth noting a requirement that is *not* listed as a design goal here: one cannot retrofit new assertion checking capabilities onto binary-only, third-party classes or components that were not written to support this technique. Instead, we envision a developer who writes his or her own wrapper along with the component to be checked and then distributes it in binary form along with the underlying component as a value-added service for customers. A customer can then enable or disable the wrapper to dynamically check proper component behavior in a new composition context, supporting the detection and resolution of component mismatches, composition errors, and glue code bugs. While supporting the ability to retrofit assertion checks onto pre-existing components is a worthy goal, it does not appear to be technically possible in C++ without changing the checked component's implementation, which presumably would require recompilation.

Experience indicates that the modularity of our approach allows it to scale up to nontrivial C++ systems [13]. This paper explains how to build contract-checking wrappers for C++ components, and provides experimental evidence to show wrappers have minimal impact on performance. Without reservations, we make use of features specific to C++ in this paper. This is necessary to build and use wrappers elegantly in C++. However, it is possible to generalize and adapt the ideas to other languages, e.g., Java [31]. Together with previous work on specification [38] and dynamic checking [36] of C++ components, the present paper makes practical and effective use of component technology possible.

Section 2 presents the design and implementation of a C++ contract-checking wrapper through an example, together with the practical aspects of using such wrappers. Section 3 reports results from experience as well as an evaluation of performance impact. Section 4 contains a discussion of the limitations of the approach and directions for further research. Section 5 compares the proposed approach with related work and presents our conclusions.

```
// Represents categories of events for which one can register
typedef unsigned int Event_Category;

// The base type for an event hierarchy
class Event
{
public:
    Event_Category category();
    // Any other base features for events go here
};

// Any object registered for events must implement this interface
class Event_Listener
{
public:
    virtual void notify( Event& e ) = 0;
};

// The abstract interface for an event registration/broadcast service
class Notification_Center
{     // The abstract view of an object of this class is a set of pairs of the form
      //   ( cat : Event_Category,  listener : Event_Listener* )
      // Let "self" refer to this abstract model for current object
public:
    Notification_Center() {}
        // ensures self = {} (an empty set)
    virtual ~Notification_Center() {}
    virtual void register_listener(
        Event_Category  cat,
        Event_Listener* listener ) = 0;
        // requires (cat, listener) is not in self
        // ensures  self = #self union {(cat, listener)}
    virtual void unregister_listener(
        Event_Category  cat,
        Event_Listener* listener ) = 0;
        // requires (cat, listener) is in self
        // ensures  self = #self - {(cat, listener)}
    virtual bool listener_is_registered(
        Event_Category  cat,
        Event_Listener* listener ) = 0;
    // ... subsequent behavioral specs omitted for brevity ...
    virtual unsigned int num_listeners(
        Event_Category cat ) = 0;
        // ...
    virtual void notify( Event& e ) = 0;
        // ...
    // Other methods ...
};
```

Fig. 2. An event registration and broadcast interface.

2 The Practice of Contract-Checking Wrappers in C++

The design of a contract-checking wrapper is best introduced by example. Section 2.1 introduces an example class interface and implementation, while Section 2.2 shows how this class is used in client code. Section 2.3 presents a corresponding contract-checking wrapper and its internal construction. Section 2.4 describes how wrapper support can be installed without requiring source code changes or recompilation, while mechanisms to provide client-level control over wrapper options are discussed in Section 2.5. Finally, Section 2.6 addresses the creation of a wrapper for a subclass by simply extending the ancestor's wrapper.

2.1 An Example: The Notification Center

To illustrate the approach for detecting contract violations, Fig. 2 presents a simplified interface for a C++ "notification center"—a simple event registry and broadcast service. A separate class, called `Event`, is at the root of the event

```
class Notification_Center_Using_Map : public Notification_Center
{
public:
   Notification_Center_Using_Map() {}
   virtual ~Notification_Center_Using_Map() {}
   virtual void register_listener( Event_Category  cat, Event_Listener* listener );
   virtual void unregister_listener( Event_Category  cat, Event_Listener* listener );
   // Other methods ...
protected:
   typedef std::vector<Event_Listener*>        Listeners;
   typedef std::map<Event_Category, Listeners> Registry;
   Registry registry;    // The registry represents the set as a map that associates
                         // each event category with a vector of listeners.
   // representation invariant
   //     for all cat: Event_Category such that cat is in registry
   //         ( registry[cat] contains no duplicate values )
};
```

Fig. 3. An event registration and broadcast implementation class.

hierarchy. The `Event_Listener` interface must be implemented by any object that wishes to receive notification of signaled events. At any time, one can register a new listener for a specific category of events, unregister a currently registered listener, or signal an event—with the result that all currently registered listeners will be notified. The notification center described here is similar in spirit to the Observer design pattern [12], except that listeners (observers) can selectively register for specific events with the notification center, and those events may be triggered by any number of (unknown) source objects, rather that an explicitly identified object playing the role of the "observable" entity. Fig. 2 includes only a core set of methods for brevity and simplicity.

The `Notification_Center` interface shown in Fig. 2 also includes a behavioral description to explain to clients how the class operates. The notation used here is based on RESOLVE [28], though any other model-based specification language [37] would do as well. An informal behavioral description might also suffice, although the developer needs a precise idea of the required behavior when producing contract-checking code.

The behavioral description of the notification center presents an abstract model of its state as a set of pairs, each consisting of an event category and an associated pointer to an event listener interested in events of that category. The behavior of each method is then described in terms of this abstract model. Note that the comments in Fig. 2 use `self`(rather than `*this`) to refer to the abstract view of the current object, for readability. The constructor, for example, ensures that each newly constructed notification center is an empty set, containing no associations. The `register_listener()` method requires as a precondition that the listener to be added is not already registered for the given event category. When properly invoked, it promises in its post-condition to add the given category/listener pair to the notification center. Within postconditions in this behavioral description, the # prefix refers to the value of an object before a method is called (the prestate), while the same object name without a # prefix refers to the new value after the method has completed (the poststate).

Fig. 3 shows part of one possible implementation of the notification center interface. This implementation builds on the Standard Template Library (STL) [22], using a map to associate each event category with a vector of pointers to listeners. Adding or removing listeners involves simple map and vector operations. Signaling an event involves looking up the corresponding vector of listeners in the map and then using the `std::foreach()` algorithm to notify each listener.

Fig. 3 also shows a representation invariant [8]—a class invariant dealing with internal, implementation-oriented concerns rather than client-visible abstract state. As with the abstract model, the representation invariant is phrased in prose rather than in a particular formal specification notation because the focus of this paper lies on how runtime checks are packaged, rather than how they are generated. For tool-supported generation of assertion checks, an appropriate specification notation would be used instead. As expressed, however, the representation invariant states that no vector within the map contains the same listener multiple times. This representation invariant reflects (but should not be confused with) an abstract, client-level invariant property implicit in the specification—as a mathematical set, the abstract state model allows no duplicate pairs. Although it is simple, the notification center involves several key features that make it a useful example for illustrating contract-checking issues.

2.2 Client Usage

The contract-checking wrapper strategy is simple and easy to use from the point of view of client code. The critical question is: *How can one conditionally insert wrappers around checked objects without requiring source code changes (or recompilation)?* Since client code is coupled to the identity of the concrete class implementing an abstract interface only at object creation, a factory [12] is an ideal solution. The idea of using a factory to isolate client code from decisions about

```
class Notification_Center_Using_Map_Wrapper : public Notification_Center
{
public:
   Notification_Center_Using_Map_Wrapper( Notification_Center_Using_Map* w,
                                          Contract_Prefs&                p ) :
       wrapped( w ), prefs( p )
   {
       // Check invariant and constructor
       // postcondition, similar to Fig. 5
   }
   virtual ~Notification_Center_Using_Map_Wrapper()
   {
       // Check invariant and destructor
       // precondition, similar to Fig. 5
   }
   virtual void register_listener( Event_Category  cat, Event_Listener* listener );
   virtual void unregister_listener( Event_Category  cat, Event_Listener* listener );
   // Other methods ...
protected:
   // Internal methods embodying run-time checks to perform
   virtual bool check_invariant();
   virtual bool pre_register_listener( Event_Category cat, Event_Listener* listener );
   virtual bool post_register_listener(
       Event_Category  new_cat,       Event_Category  old_cat,
       Event_Listener* new_listener,  Event_Listener* old_listener,
       Notification_Center_Using_Map& old_nc );
   // Pre- and postcondition checkers for other methods ...
protected:
   Notification_Center_Using_Map* wrapped;
   Contract_Prefs&                prefs;
};
```

Fig. 4. A contract-checking wrapper for the implementation class.

which particular concrete class is used when creating new objects is standard practice for many object-oriented developers. Any reasonable factory approach is viable here, leading to client code similar to the following:

```
// Objects are created using a factory
Notification_Center* center =
  Notifier_Factory::create();
center->register_listener( my_category,
  my_listener );
center->notify( next_event );
```

Since construction of new objects is handled through the factory pattern [12], decisions about whether or not to use wrappers (or which checks to enable) can be controlled and localized elsewhere. Thus, client code does not require recompilation to switch between wrapped and unwrapped objects. Section 2.4 describes how wrappers can be "plugged in" as a link-time option without requiring any recompilation (simply by adding the wrapper's object file in the linking sequence), while Section 2.5 shows how wrapper creation can be enabled or disabled at runtime if the corresponding object code has been linked into an application.

In practice, a "smart pointer" template [1], [2] could be used to manage the heap-allocated notification center object automatically. In addition, the factory call could be encapsulated inside a template constructor on such a smart pointer—something that could be achieved with a simple extension to Batov's `Handle` template [2], for example.

2.3 A Contract-Checking Wrapper

Now that the client code perspective has been addressed, the issues of implementing a contract-checking wrapper can be explored. If the runtime checks are separated from the underlying component implementation, how will those runtime checks be implemented? A simple approach is to allow the checking code to have direct access to the internal data stored within the underlying component. Elsewhere, this approach has been compared to more sophisticated alternatives that do not require direct access to internal component state [9]. To simplify exposition, here we use the direct access approach.

Implementing a checking wrapper is then straightforward in principle: Simply move any contract violation checks one would normally place inside the methods of the underlying component into a new class, and ensure that within this new class, all internal data members in the underlying implementation are visible. In C++, this can be achieved by adding a `friend` declaration to the corresponding concrete class, such as the `Notification_Center_Using_Map` implementation in Fig. 3.

Fig. 4 outlines the contract-checking wrapper class for the notification center implementation shown in Fig. 3. Like

```
void Notification_Center_Using_Map_Wrapper::
register_listener( Event_Category  cat, Event_Listener* listener )
{
    // Check invariant on entry
    if ( prefs.enabled( Contract_Prefs::Entry_Invariant )  &&  !check_invariant() )
        prefs.handle( Contract_Prefs::Entry_Invariant,
                      "Notification_Center_Using_Map::register_listener" );

    // Check precondition
    if ( prefs.enabled( Contract_Prefs::Precondition )  &&
         !pre_register_listener( cat, listener ) )
        prefs.handle( Contract_Prefs::Precondition,
                      "Notification_Center_Using_Map::register_listener" );

    // Perform operation
    if ( prefs.enabled( Contract_Prefs::Postcondition ) )
    {
        // Save pre-state values for use in postcondition checking
        Notification_Center_Using_Map old_self     = *wrapped;
        Event_Category                old_cat      = cat;
        Event_Listener*               old_listener = listener;

        // Delegate to the wrapped object
        wrapped->register_listener( cat, listener );

        // Check invariant again before exit
        if ( prefs.enabled( Contract_Prefs::Exit_Invariant ) && !check_invariant() )
            prefs.handle( Contract_Prefs::Exit_Invariant,
                          "Notification_Center_Using_Map::register_listener" );

        // Check postcondition on exit
        if ( !post_register_listener(
                    cat, old_cat, listener, old_listener, old_self ) )
            prefs.handle( Contract_Prefs::Postcondition,
                          "Notification_Center_Using_Map::register_listener" );
    }
    else     // Skip saving the pre-state if postcondition isn't checked
    {
        // Delegate to the wrapped object
        wrapped->register_listener( cat, listener );

        // Check invariant again before exit
        if ( prefs.enabled( Contract_Prefs::Exit_Invariant ) && !check_invariant() )
            prefs.handle( Contract_Prefs::Exit_Invariant,
                          "Notification_Center_Using_Map::register_listener" );
    }
}
```

Fig. 5. Inside a wrapped method.

the underlying class, the wrapper implements the same abstract `Notification_Center` interface. The wrapper contains only two data members: A pointer to the wrapped object to which all the real work will be delegated and a reference to a "preferences" object that defines the options for controlling the behavior of this wrapper (Section 2.5). Values for both data members are passed as parameters to the wrapper's constructor.

In addition to implementing the public methods declared in the `Notification_Center` interface, the wrapper also declares a number of internal methods for use in performing its checks. Only `check_invariant()` and pre and post-condition check operations for `register_listener()` are shown in Fig. 4 for brevity. Each such helper operation is a Boolean-valued function that embodies a specific runtime assertion check to perform.

Each public `Notification_Center` method is overridden to wrap contract-checking actions around a call to the corresponding method on the wrapped object. Fig. 5 shows how such a method is implemented, using `register_listener()` as an example. The preferences object determines whether or not to perform each check, meeting Requirement #2. If a check fails, the preferences object also determines how best to handle the failure (Requirement #4).

In Fig. 5, first, the invariant is checked if desired. This is because, in modular reasoning, the representation invariant is required to be true at the beginning of every public method and guaranteed to be true at the end. If the invariant is false before beginning a method, then the object's state is corrupt.[1] The actual checking code does not appear inline in this method, but is instead placed inside `check_invariant()`, allowing the invariant to be strengthened or extended easily by wrappers for subclasses (Section 2.6). Second, the precondition for `register_listener()` is checked. A precondition failure signals a violation of the public contract by the client code invoking the method. If the precondition fails, the appropriate handling action is invoked and the remainder of the method is skipped in order to protect the integrity of the underlying object from the invalid client call. Third, the wrapper tests to see if postconditions are enabled. If so, fourth, it saves the state of the wrapped object and any method parameters that might be modified by the underlying operation. This step is generally needed for full postcondition checking, since a typical postcondition (including this method's) describes an object's new state or a method's return value in terms of the "old" values at the time of method invocation.[2] Deep copying (provided by `std::map`'s copy constructor) is used to capture a complete picture of the prestate before method invocation. No deep copy of the given event listener pointer is performed in Fig. 5 since the contract is specified in terms of the pointer's value rather than the value of the object to which the pointer refers (a full deep copy would be performed if it were required to check the postcondition). Fifth, the wrapper calls the corresponding method on the wrapped object. Sixth, the wrapped object's invariant is checked again, and seventh, the postcondition is checked. It is necessary to check the invariant at the end because every public method must guarantee that the invariant holds on exit. Proper ordering of invariant checks aids in detecting erroneous object states that could potentially cause other checks to crash. If postcondition checking is not enabled, an alternate branch allows the cost of copying prestate values to be avoided. This is much easier to achieve with minimal code duplication when both the checking code and the underlying implementation are in separate methods.

The internal checking methods are executable implementations of the corresponding behavioral descriptions in the component's contract. Ideally, an appropriate tool for the chosen behavioral description notation can be used to generate them, although the focus here is on how they are packaged and distributed. Fig. 6 shows the implementations for the invariant check and the `register_listener()` precondition check, as well as an outline for the `register_listener()` postcondition check. Pre and postcondition checking operations for the other `Notification_Center` public methods are similar. Here, it is worth noting that although `check_invariant()` sorts the listeners in each vector, it is not modifying the original map—instead, it is sorting an independent *copy* of each vector, created by copy construction when initializing the local variable `L`. This important step prevents the check from altering the map's internal state, which potentially could mask latent defects in the underlying notifier class.

With wrappers, the focus is on checking the contract "between" the client and the underlying component and, therefore, assertion checks are only performed on entry to and exit from the public methods. Since the underlying component makes no reference to the wrapper when calling other methods defined in the underlying class (or any of its ancestors), self-calls or super-calls within the implementation proceed without checking. Generally, this is appropriate because our error detection concerns clients without access to and without the knowledge of the internal code. Further, all contract violations visible to the client will be detected and reported. Nevertheless, the difficulty in checking self-calls in the wrapper approach is a limitation and it is discussed in Section 4.

For simplicity, exception handling issues are omitted from the wrapper design presented here. For behavioral description techniques that include exception specifications, the approach used by JML is suitable [4]. This involves creating a separate postcondition checker method for each class of exception that can be thrown from a method, placing the call to the wrapped object in a `try` block, and providing a `catch` handler for each exception class that invokes the corresponding postcondition checker. An additional handler to catch unexpected exceptions and signal a postcondition failure is also required. The invariant must be checked regardless of the manner by which control is transferred out of the method. Cheon and Leavens also provide a discussion of how one can handle exceptions that arise during the execution of assertion checks [4].

2.4 Installing Wrapper Support without Recompilation

Although the mechanics of contract-checking wrapper construction are straightforward, another issue remains (Requirements #1 and #3): *How can support for contract-checking wrappers be installed or removed from an application without requiring source code changes (or recompilation)?* This goal can be achieved by combining the factory concept used for object creation with careful use of dynamic binding. Fig. 7 shows a simple implementation of a notification center factory that demonstrates the concept. The `Notifier_Factory` provides a minimal public interface: just a `create()` method that can be called to create new objects. For wrapped objects that support multiple constructors, overloaded versions of `create()` are a natural extension. In a C++ design, the declaration for this factory class would normally appear in the same header file as the `Notification_Center` interface; together, these two classes provide all of the information necessary for client code to access and use notification center services.

Fig. 7 also shows the implementation for the `Notifier_Factory` class methods, which can be compiled separately. Internally, the factory's `create()` method first creates a new notification center object using the underlying concrete class. Next, it checks for an installed *wrapper*

1. Invariant checking at the beginning can be optimized out if we can assume that only exported methods can be used to alter the state of the object. Since this assumption is not true, in general, for C++ components, the check is included in both places here.

2. We expect wrapper generation to optimize out avoidable inefficiencies in the automation process. For example, if old values are not referred to in a postcondition, then there is no reason to include code to save those values.

```
bool Notification_Center_Using_Map_Wrapper::check_invariant()
{
    bool result = true;
    // For each category, look for any duplicate entries
    for ( Registry::iterator entry = wrapped->registry.begin();
          entry != wrapped->registry.end();
          entry++ )
    {
        Listeners L = entry->second;
        sort( L.begin(), L.end() );
        if ( adjacent_find( L.begin(), L.end() ) != L.end() )
        {
            result = false;
            break;
        }
    }
    return result;
}

bool Notification_Center_Using_Map_Wrapper::
pre_register_listener( Event_Category cat, Event_Listener* listener )
{
    // Make sure this pair is not already in the registry
    Registry::iterator entry = wrapped->registry.find( cat );
    return entry == wrapped->registry.end() ||
        find( entry->second.begin(), entry->second.end(), listener )
            == entry->second.end();
}

bool Notification_Center_Using_Map_Wrapper::
post_register_listener( Event_Category  new_cat,       Event_Category  old_cat,
                        Event_Listener* new_listener,  Event_Listener* old_listener,
                        Notification_Center_Using_Map& old_nc )
{
    // Pseudocode, for brevity:
• Use std::mismatch() to find entry in map where new and old registries differ; en-
  sure event category at this mismatch is the right one.
• Use std::mismatch() again on the vector at that location, to find listener where
  two vectors differ.
• Ensure difference is at the newly added listener; step over it in the new version
  of the registry.
• Use std::equal() to ensure the remainder of the two vectors are the same.
• Use std::equal() to ensure the remaining entries in the map are also the same.
}
```

Fig. 6. Examples of contract-checking methods.

creation function. Such a wrapper creation function can be registered by storing its address in the factory's static `wrap_fn` data member. If a wrapper creation function is registered, the newly created notification center object is passed to it for "wrapping." The resulting object is returned to the client. The factory does not define any wrapper creation function itself and will work seamlessly without one. Instead, it simply initializes the function pointer to be null. In effect, the `wrap_fn` pointer represents an explicit binding to a wrapper creation service that can be changed at runtime. Changes to this pointer value will affect object creation through the factory without being visible to the client and without requiring client code to be recompiled.

To complete wrapper installation, an appropriate wrapper creation function must be registered. Fig. 8 shows the `Wrapper_Factory`, which defines a wrapper creation function called `add_wrapper()` as a static method. The `Wrapper_Factory` class is a subclass of `Notifier_Factory` and can access the `wrap_fn` static variable to register `add_wrapper()`.

Fig. 8 also shows a static declaration for a `Wrapper_Factory` object. The sole purpose of this object is to cause the `Wrapper_Factory` constructor to be executed before the `main()` routine begins executing. C++ static variable initialization rules require that the constant data initialization of `wrap_fn` to null (in Fig. 7) occur before any constructors for global or static objects are executed [29], so the `Wrapper_Factory` constructor will execute after this initialization. This provides for clean, predictable wrapper registration before `main()` begins. It is worth noting that, if any global, statically allocated objects declared in a separate compilation unit

```
class Notifier_Factory
{
public:
    static Notification_Center* create();
protected:
    typedef Notification_Center* (*Wrap_Fn)( Notification_Center* );
    static Wrap_Fn wrap_fn;
};

// Method implementations can be placed in a separate file
Notification_Center* Notifier_Factory::create()
{
    Notification_Center* result = new Notification_Center_Using_Map();
    if ( wrap_fn )
        result = wrap_fn( result );
    return result;
}

// Declaration of the static pointer to the wrapping function
Notifier_Factory::Wrap_Fn Notifier_Factory::wrap_fn = 0;
```

Fig. 7. A simple factory class.

```
// This class extends the base Notifier_Factory. Ideally, it will be declared
// locally within the C++ source file defining the wrapper class's method
// implementations.  Only when the corresponding object file is linked to the
// executable  will this extension be enabled.
class Wrapper_Factory : public Notifier_Factory
{
public:
    Wrapper_Factory()
    {
        // When instantiated, install our wrapper creation function
        if ( prefs.enabled( Contract_Prefs::Install_Wrapper_Support ) )
            wrap_fn = &add_wrapper;
    }
private:
    static Notification_Center* add_wrapper( Notification_Center* kernel )
    {
        Notification_Center_Using_Map* inside =
            dynamic_cast< Notification_Center_Using_Map* >( kernel );
        if ( inside  &&  prefs.enabled( Contract_Prefs::Create_Wrappers ) )
            return new Notification_Center_Using_Map_Wrapper( inside, prefs );
        else
            return kernel;
    }
    static Contract_Prefs prefs;    // initialized below
};

Contract_Prefs Wrapper_Factory::prefs( "Notification_Center_Using_Map" );

// Create one instance at startup to "install" wrapper support
static Wrapper_Factory register_this_wrapper;
```

Fig. 8. Extending the factory to add wrapper support at link time.

call the `Notifier_Factory::create()` method in their constructor, the C++ language standard does not specify in which order the constructors will be executed. This means a `Notification_Center` object created as part of such a static initialization may potentially be unwrapped because it was created before `add_wrapper()` was registered. Otherwise, all objects created after the start of `main()` can be wrapped.

```
// This class provides an interface through which
// wrapper behaviors can be conditionally controlled.
class Contract_Prefs
{
public:
    enum Assertion_Category {
        Entry_Invariant = 0,
        Exit_Invariant,
        Precondition,
        Postcondition,
        Create_Wrappers,
        Install_Wrapper_Support
    };
    Contract_Prefs( char* class_name );
    bool enabled  ( Assertion_Category c );
    void Set_Pref ( Assertion_Category c, bool new_val );
    void handle   ( Assertion_Category c, char* method );

private: // Class implementation goes here
};
```

Fig. 9. The interface for a wrapper behavior preference class.

The code shown in Fig. 8 can be included in the same source file as the implementation of the wrapper class. When the wrapper code is linked into a program, its wrapper creation function will automatically be registered with the corresponding factory. When the wrapper's object code is omitted from program linking, its wrapper creation function will not be registered and only unwrapped objects will be used. The result is a simple scheme for installing wrapper support at link time without requiring any source code changes or recompilation.

2.5 Controlling Wrapper Options

Figs. 4, 5, and 8 all use a `Contract_Prefs` class to tailor runtime behavior, in support of Requirements #2 and #4. In spirit, this class simply stores Boolean-valued preferences for the various actions the wrapper can perform. Fig. 9 shows the interface of such a class, although other implementation choices are possible. The remainder of this section explains the design used for obtaining the performance data in Section 3.

The set of preferences are defined by an enumeration within `Contract_Prefs`. Individual settings are retrieved using the `enabled()` method. Fig. 8 shows one `Contract_Prefs` object being created for use with all `Notification_Center_Using_Map_Wrapper` objects. The preferences object is given the name of the class to be wrapped when it is initialized.[3] In addition to controlling which checks (invariants, preconditions, and/or postconditions) are enabled, the preferences object is also consulted when installing wrapper support at startup. If this preference is false, then no wrapper support will be installed at start up, and client code will operate just as if the wrapper class (and associated factory extension) were not linked into the program. Assuming the wrapper creation function is installed, the preferences object is queried on each object creation to see if the customer wishes for the underlying class to be wrapped.

The `Contract_Prefs` class also allows the customer to control the response action to failed checks. As shown in Fig. 5, when a contract check fails, the preference object's `handle()` method is invoked. The preference object thus decouples the response action from the wrapper. Internally, the `Contract_Prefs` object references a "violation handler" object. Different violation handler implementations with a common interface support different responses: throwing an exception, popping up an information dialog, or any other action. The desired action can be looked up using the same mechanism as for other preference settings, and user-defined handler actions can be supported.

The `Contract_Prefs` class separates choices regarding which features are enabled and how violations are handled from *both* the underlying component *and* the client code. Paired with a mechanism for adjusting preferences at runtime, the client has flexible control over wrapper behavior on a per-class basis. The result is a clean mechanism for addressing Requirements #2 and #4 without requiring source code changes.

2.6 Subclassing with Wrappers

The traditional approach of generating checks inside the methods being checked poses many problems in the face of subclasses. In particular, embedding the checks in the parent's methods makes it difficult for subclasses to reuse the checks without duplication, to extend the checks, or to call parent methods without performing unnecessary checks.

When using wrappers with subclassing, one can define a parallel hierarchy of public interfaces that captures the external contract of a parent class and its subclasses.

3. Instead, initial preference values might be read from a separate initialization file or a registry service. The preference object could even dynamically register itself with a separate manager object [6] so settings could be controlled dynamically through a GUI control panel—similar to iControl [15], but at runtime.

Further, a mirror hierarchy of wrapper classes can provide assertion checking support for each concrete class under consideration. A separate factory can then be provided for each concrete class so that client code can exercise control over exactly which type of object is created at any given point. This structure raises the question of how a wrapper for a subclass, which inherits from the superclass's wrapper, can define its own checks in a clean way.

Our approach is based on the technique implemented in JML. The JML technique [18] takes advantage of prior work [7], [11] and relies on renaming the original methods and augmenting the checked class with wrapper methods, rather than placing the checks in a separate class. However, JML places the code implementing the checks into helper methods in a manner similar to that shown in Figs. 4 and 5. As a result, when the externally visible contract (interface) for a subclass strengthens a postcondition or weakens a precondition on an overridden method, or strengthens a class invariant, the subclass's wrapper need only override the corresponding check. Further, the subclass wrapper's overriding definition for this check can easily reuse the parent wrapper's code for the same check by directly calling it—the checking code for the concrete parent class is no longer inline in the main operation.

For example, suppose we want to create a subclass of `Notification_Center_Using_Map` that eliminates the precondition on `register_listener()`. This new subclass would have a corresponding wrapper class that subclasses `Notification_Center_Using_Map_Wrapper`. The wrapper subclass would not have to redefine the `register_listener()` method; instead, it would only need to redefine the `pre_register_listener()` helper method corresponding to the precondition of interest.

The manner in which subclass checks are implemented depends on the interpretation given to a subclass's behavioral description, an issue orthogonal to the contributions of this paper. Some specification approaches interpret a subclass contract as an extension of its parent's contract, where additional precondition clauses in the subclass indicate additional prestates that are acceptable beyond those captured in the parent contract, and where subclass postcondition or invariant clauses indicate additional guarantees provided by the subclass. In this case, the wrapper for a subclass can extend checks by using disjunction to combine new precondition clauses with the parent's check, and by using conjunction to combine new postcondition or invariant clauses with the parent's check. Alternatively, one can consider the subclass's behavioral description to completely replace (rather than add to) its parent's. This may allow a subclass's behavioral description to conflict with that of its parent. Findler et al. [11] describe an approach to checking subclass contracts that can detect when such a subclass contract is not LSP-compliant from the perspective of the client code.

3 Experience and Evaluation

3.1 Practical Experience Using Wrappers

Contract-checking wrappers have been used successfully on a realistic scale and are an effective means of managing runtime assertions. A wrapper-based technique (a precursor to that described here) has been employed by Hollingsworth in commercial development [13]. The product uses about 250 components implemented in approximately 100,000 lines of C++ code. Since its introduction in 1993, the product has developed a customer base of about 2,000 sites.

Throughout product development, contract-checking wrappers supporting precondition checks were used during unit development, unit testing, integration, and system testing. The developers cited this practice as important in achieving such a high level of quality [13]:

> This approach detected almost all of the client [component]'s original defects during *unit* testing and system integration. Very few defects were revealed during *system* testing, and only two were detected postrelease. [...] Unit testing and system integration with checking components in place dramatically reduced the effort needed for system testing and debugging compared to a more traditional approach. [emphasis in original]

Hollingsworth's wrappers did not employ factories and thus required recompilation for removal. In this experience, primarily due to the C++ design strategies chosen, no special issues arose with regard to postcondition checking in the face of exceptions, or with regard to composing subclass checks with superclass checks in wrappers for subclasses. The developers reported that using wrappers provided a dramatic increase in the confidence they had in the correctness of their code.

In addition to this commercial experience, wrappers also have been used in the laboratory. Edwards has carried out an experiment using contract-checking wrappers to detect bugs while measuring the effectiveness of test suites [10]. Runtime contract-checking identified 100 percent of the artificially seeded defects triggered by the test data—the only undetected bugs were those that were not exercised by the test suites under examination.

3.2 Impact of Factory-Based Object Creation

Two questions arise about the performance cost of the wrapper approach: What is the impact of factory-based object creation, and what additional overhead is imposed by the wrappers themselves? To address these questions, a small experiment was conducted using the notification center example presented in Section 2.

Since the factory code is virtually identical for all wrappers, the additional overhead imposed will be approximately the same for every class for which wrapper support is installed. To measure this overhead, the actual running time of a sequence of 50 allocation and deallocation operations was collected for four alternatives:

1. Direct allocation of the underlying class using `operator new` (the baseline).
2. Factory-based allocation of an unwrapped object.
3. Factory-based allocation of a wrapped object, without testing the `Contract_Prefs`.
4. Factory-based allocation of a wrapped object including the cost of checking the corresponding `Contract_Prefs` setting at runtime.

Times were collected on a 1.2GHz Athlon processor running Windows 2000 Professional with 256MB of

TABLE 1
Running Times for 50 Allocation and Deallocation Operations

Condition	Time (μs)	Difference From Baseline
operator new (baseline)	160.7	—
Factory call, no wrapper	160.9	0.2
Factory and wrapper, no preferences	191.1	30.6
Factory with wrapper and preference test	191.7	31.0

TABLE 2
Running Times for a Sample Sequence of Method Calls

Condition	Time (μs)	Difference From Baseline
Raw, unwrapped	9.9	—
With in-line assert() checks, no wrapping	123.2	113.3
With wrapper	132.3	122.4
With wrapper, all checks disabled	13.9	4.0

memory, using code generated using GNU g++ 3.0.4 compiled with optimization (-O3). Times were measured using the approach described by Musser [22] for timing C++ library operations.

Table 1 summarizes the results. Using a factory to create unwrapped objects instead of directly allocating objects using `operator new` adds no measurable amount of overhead. This is due to the fact that the vast majority of the running time for such an operation is taken up in heap management activities, and the addition of a single function call (which can be inlined easily and optimized away) is negligible in comparison. Thus, the approach described here will have negligible performance impact on the final product when wrapper support is not installed.

Table 1 indicates that creating the wrapper in addition to the underlying object increases object creation time by less than 20 percent. This relative amount will vary depending on the cost of creating the underlying object, but on an absolute scale it should remain fixed for all wrappers because the wrapper allocation overhead is independent of the underlying class. Naively, one might assume that it will take approximately twice as long to create the wrapper together with the underlying object. However, the `Notification_Center_Using_Map` class includes an `std::map` that contains other heap-allocated structures internally. Creating such an object requires several heap operations, while creating the wrapper requires only one more. Table 1 also indicates that checking the preferences object to see if wrapper creation is desired on every factory call adds a negligible amount of overhead relative to the allocation cost.

3.3 Impact of Wrapper-Based Contract Checking

The impact on basic object method calls was also studied. A sequence of 100 notification center operations was timed, consisting of 5 percent `register_listener()` calls, 5 percent `unregister_listener()` calls, and 90 percent `notify()` calls. This sequence is intended to be a coarse approximation of notification center usage, where event broadcasts predominate. No assertion violations were triggered in this sequence of calls—the normal case for which performance (as opposed to defect detection and isolation) is an issue. The same code was timed against four separate notification center objects:

1. An unwrapped `Notification_Center_Using_Map`.
2. A modified `Notification_Center_Using_Map` class that uses the same contract checks placed inline using `assert()` macros.
3. A wrapped `Notification_Center_Using_Map`.
4. A wrapped `Notification_Center_Using_Map` where all checks are disabled using `Contract_Prefs`.

Table 2 summarizes the time taken for the subject code sequence. Wrappers impose some additional overhead beyond inline assertions, adding approximately 8 percent in this example.

3.4 Wrapper-Independent Overhead Costs

What is more striking is the large cost of executing the runtime checks at all, whether or not they are inlined. To explore this issue, additional timing runs were used to identify the relative contributions of the various parts of the contract-checking code to this running time. Fig. 10 shows the breakdown of the overhead for both inline and wrapper-based assertions. In Fig. 10, "delegation" is the cost of the virtual dispatch to the underlying component, "dynamic binding of checkers" is the cost of the four virtual calls to the checking helper methods, and "preference tests" is the cost of calling `enabled()` on the `Contract_Prefs`.

As Fig. 10 shows, the dominating contributors to assertion checking overhead are independent of the wrapper strategy. The majority of the overhead is devoted to one task: saving the prestate values before calling the checked method. While saving the values of incoming parameters is not expensive in this case, saving the value of the current notification center object is time consuming. The STL implementation provided with GNU g++ implements `std::map` as a red-black tree with heap-allocated nodes. Further, each node in this tree contains a pair: an event category together with a vector of listeners. The vector in each pair contains a heap-allocated array. As a result, copying the prestate value of a map involves numerous heap allocations.

Similarly, when the copy is finalized as it goes out of scope, corresponding deallocation operations occur. These heap operations dominate the overhead. This information highlights the use of two alternate branches in Fig. 5, one that saves prestate values when postconditions are being checked, and one that avoids this cost when it is not needed, resulting in improved performance when checks are disabled (Table 2). Further, when prestate values are not needed to check a postcondition, one branch can be eliminated entirely.

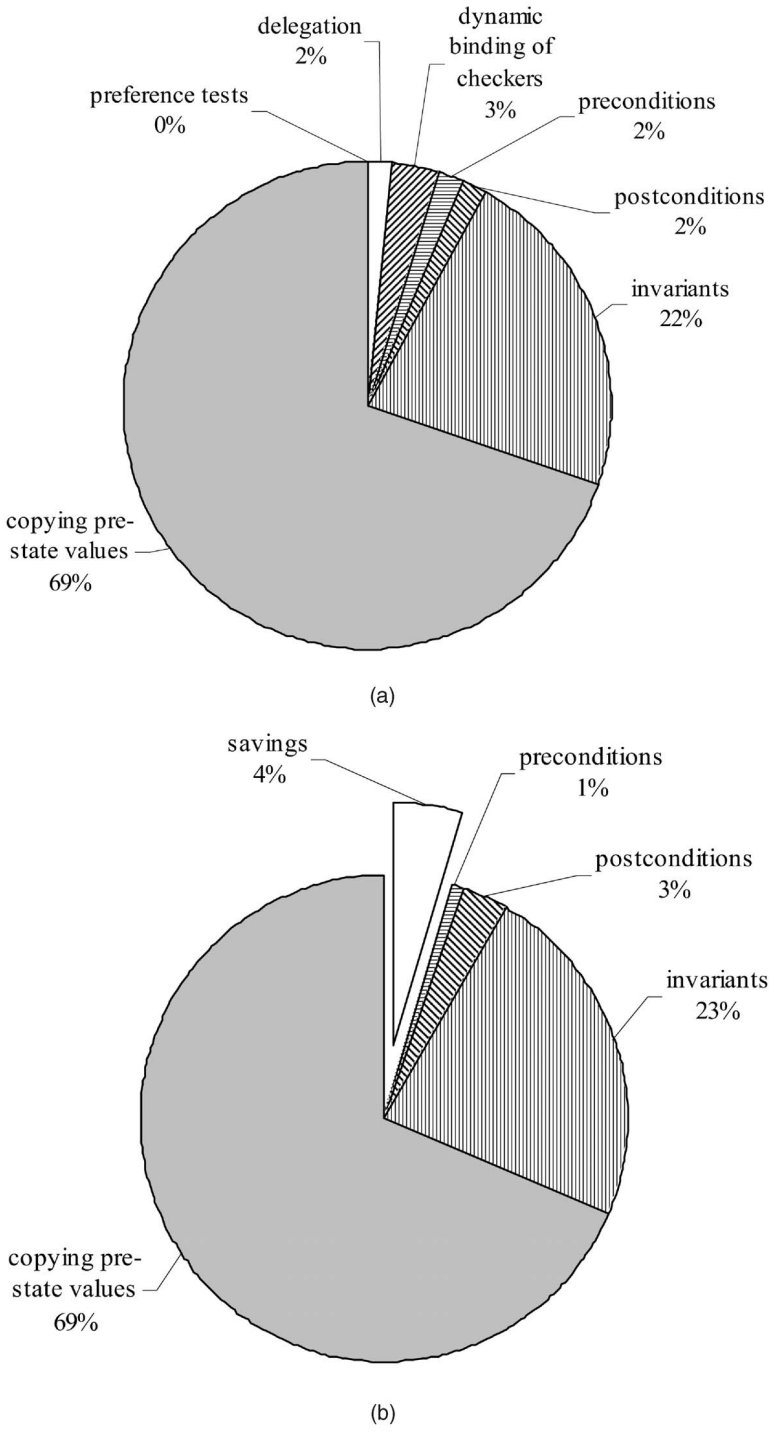

Fig. 10. Breakdown of assertion checking overhead (a) with wrappers and (b) with inline checks.

The second-largest contributor to the overhead is checking the invariant, the code for which is in Fig. 6. Although checking the invariant does not copy the entire map, it does copy each vector in the map before sorting it, involving more heap operations. In both cases, the extra copying is due in part to the value-oriented semantics of STL abstractions. However, in practice, this copying is critical; runtime checks must not modify the representation of the underlying object in any way. Any such modification might mask a defect by accidentally "erasing" its effect, or worse, it might introduce defective behavior of its own.

However, the performance issues with generalized heap memory management are well-known. Alexandrescu presents custom allocators that can significantly improve the situation [1]. A custom allocator using a simple free list reduced the overhead by two thirds, yielding a wrapper-based time of 53.9 μs compared to an `assert()`-based time of 47.1 μs, and further reductions may be possible. For example, the overhead imposed by dynamic binding of

checking helper methods can be eliminated in C++ by using template composition rather than virtual methods [1]. Template composition would allow subclasses to extend superclass assertions easily, while also allowing static binding (and potential code inlining) within wrapper methods [9].

4 Limitations and Future Directions

This section summarizes limitations of our current efforts and future directions. First, we outline and address key issues in transitioning the principles to other languages, such as Java. Then, we discuss more fundamental improvements to the core wrapper technology itself.

4.1 Wrapper Technology Transition to Other Languages

The core concepts behind the strategy discussed here—wrapper classes, delegation, and the factory pattern—can be applied in most object-oriented languages. The difficulty in applying this strategy to other languages lies in two of the implementation details: how wrappers gain access to the data within the underlying class, and how factory registration can be achieved. Because of language differences, unique solutions for these issues must be devised in different languages. For example, a similar wrapper design has been proposed and tested for Java [31]. It addresses field access by promoting fields to a protected level of visibility and including the wrapper in the same package as the wrapped class. Wrapper registration within a factory can be achieved using Java's dynamic class loading facility: The factory can search for the desired wrapper using the current class loader when it is initialized, and conditionally load and install wrapper support only if it is found. As part of future work, we plan to explore the remaining issues in transitioning this approach to languages other than C++.

4.2 Limitations and Directions for Improvement

While the wrapper-based approach to managing contract-checking code is a useful contribution toward addressing the needs of component clients, more work must be done to provide full contract-checking support. One of the most significant limitations is lack of support for checking self and super-calls. While contract violations by a self-call indicate issues that fall entirely "within the box" of the original component developer's responsibilities, intuitively one would prefer to check them if possible. Tan and Edwards [31] suggest one path toward a solution: include a "pointer to self" data member within the original class, and initialize this field via the factory. The field would point to the current object if it were unwrapped, or to the enclosing wrapper if one existed. The class could then be modified so that all self- and super-calls went through this data member. However, this approach requires much more invasive changes to the underlying component. While the concept of employing load-time bytecode transformations to achieve such changes automatically is enticing in language runtimes that support it (e.g., Java and .NET), the question of how best to address this issue in languages such as C++ is still open.

This limitation also suggests that subclassing in the face of wrappers is another issue needing further exploration. While Section 2.6 provides an outline of how a developer can provide a family of components related by inheritance, and how wrappers for subclasses can reuse the checks embedded in wrappers for their parent(s), other issues remain. In particular, if a client wishes to write new subclasses of such a commercial component, the lack of checking on self- and super-calls within wrapped objects may allow client-authored defects in subclasses to go undetected. This issue has already been addressed in several other assertion checking approaches, and adapting an appropriate technique for wrapper-based checks is an issue that must be addressed.

Another important consideration is that this paper does not address assertion generation (which has been well-covered by other researchers), but focuses only on packaging and managing the runtime checking code. We have experimented with automatically generating contract-checking wrapper support code [9], [27]. The next logical step is to combine a mature contract-based assertion generation tool with the wrapper-based assertion deployment strategy. We are currently working with the developers of JML to modify its assertion generator to package runtime checks in separate wrapper classes similar in design to those presented here [31].

We also intend to carry out more experiments in Java to explore the runtime costs of using wrappers by comparing wrapper-based assertions with prior JML assertions, iContract-style assertions, and assertions implemented using the new `assert` feature in the latest version of the Java Development Kit. This will require extending wrappers to handle thread-safe access to shared objects in a concurrent environment, a concern inherent to Java. In addition, we have also described how behavioral contracts described using abstract, model-based specifications can be checked at the abstract level using an "abstract value conversion" approach [9]—a strategy that boils down to using a program-version of an abstraction function to manipulate object state from an abstract view. This strategy naturally meshes with JML's model-based specification support. Further, although executing the programmatic version of an abstraction function might seem to impose a significant performance burden, it *replaces* the need for copying prestate object values. We plan to conduct an empirical evaluation of the impact of such an approach relative to pre-state copying to assess the performance trade offs.

To move from C++ classes to supporting a more sophisticated notion of software component, one could extend this approach to an object-based component model, such as JavaBeans. While all of the issues in client-oriented support of assertion checks arise in the simpler case of object-oriented classes, the benefits are even more compelling in the case of commercially popular component models. Fortunately, most component models already require the use of public interfaces for accessing behavior and factories for object creation to maintain compile-time independence between components. As a result, difficult issues in dealing with plain classes are addressed more easily in such component-oriented settings.

Finally, perhaps the most significant open problem worthy of investigation is: How can one retrofit runtime assertion checking onto an existing binary-only component

for which no source is available, when that component was never written to support such an addition? While this seems infeasible for C++, other languages may present different opportunities. In particular, bytecode editing capabilities in Java or the .NET framework may allow for such possibilities in other settings, as described in [31]. Extending wrappers in such a direction would be a notable advancement.

5 Related Work and Conclusions

The responsibilities for interface checks in our framework are probably best presented by Bertrand Meyer in the design-by-contract method [20]: Preconditions of operations are obligations on callers and postconditions are obligations on implementers who may assume that the preconditions hold at the time of invocation. The roots of this approach go back at least as far as Parnas [23]. Although some researchers have proposed alternate divisions of responsibilities [19], [24], Meyer's view is embraced by most contract-checking tools.

Eiffel's design-by-contract support [21] is exemplary of many existing tools. Assertions are expressed in the original component using annotations and the compiler can either generate corresponding checking code inline within methods, or omit such code, based on a compile-time option. iContract [17] and JML [18], [16] provide similar capabilities for Java, with variations in the assertion language used to describe behaviors. iControl [15] is an add-on tool for iContract that provides a GUI for selecting which categories of assertions are generated on a per-class basis, although changing these decisions still requires recompilation. Simple `assert()` macros in C and C++ are a primitive form of the same idea. Rosenblum's Annotation Pre-Processor (APP) [26] provides more sophisticated assertion generation capabilities for C. APP allows customizing the action taken when an assertion fails (Requirement #4), although this must be done by changing the text of the assertion and recompiling. APP also supports different levels of severity and allows runtime enabling or disabling of assertion checks based on severity level. Unlike our approach, these techniques all require recompilation to insert or remove checking code (against Requirement #3) and all require special tool support to do so (against Requirement #5).

Plösch summarizes and evaluates the assertion support in eight distinct tools targeting Java [25]. His evaluation criteria are primarily focused on the expressive capabilities of the notations used to express runtime-checkable contract properties, but the evaluation also includes criteria relevant to packaging and management design goals considered in the wrapper approach. In particular, Plösch considers flexibility in reporting violations, fine-grained control over enabling and disabling checks, and efficiency impacts when assertions are turned off. While many of the Java-oriented tools he evaluated address one or more of the five requirements laid out in Section 1.2, all of the tools discussed require either source component code to be available or utilize bytecode editing strategies at load time. As a result, while Plösch recognizes the need for several of our requirements, none of the tools included in the survey fulfill all of them in a way applicable to C++.

Several researchers also have used wrapping to add assertions to a component. JML, for example, augments the underlying component by renaming its original methods, and replacing each one with a "wrapper" method that performs checks before and after calling through to the now-renamed original. Findler et al. [11] employ a similar strategy; their focus is on assigning proper blame when assertions fail, particularly in the case where a subclass does not properly live up to the contract of its ancestor(s). Duncan and Hölzle [7] use the same technique in Handshake. While these three systems all wrap individual methods, none pushes the checking code into a separate wrapper class. Both JML and the Java contract compiler of Findler et al. require recompilation to add or remove checking code. Handshake goes further, however, by generating checking code fragments separately and then inserting them into checked classes when the corresponding Java bytecode is read from a file at load time. While this meets Requirement #3, it relies on techniques specific to Java's bytecode representation that cannot be applied in most other languages, including C++.

Runtime assertion checks can offer significant benefits to both component developers and clients. While existing approaches do an excellent job at allowing developers to insert, remove, enable, and disable checks without changing source code, our wrapper-based packaging and delivery strategy does this for C++ classes while also meeting important client-oriented requirements. It allows checking code to be added or removed at link time without requiring changes to source code or recompilation of *either* the underlying component or the client code. Further, the wrapper design presented here enables the client to selectively enable or disable various assertion features on a per-class basis, as well as to provide alternative "handler" responses to failed assertions without modifying the source code of the client. Finally, experimental evidence suggests that wrappers in C++ impose only modest overhead compared to inlining assertion checks.

Acknowledgments

The authors gratefully acknowledge the financial support from our own institutions, from the US National Science Foundation under grants CCR-9311702, CCR-0081596, and CCR-0113181, from the Defense Advanced Research Projects Agency under project number DAAH04-96-1-0419 monitored by the US 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 the NSF, the US Department of Defense, or NASA.

References

[1] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley, 2001.

[2] V. Batov, "Safe and Economical Reference-Counting in C++—Smart Pointers Keep Getting Smarter," *C/C++ Users J.,* vol. 18, no. 6, pp. 44-57, June 2000.

[3] A. Beugnard, J.-M. Jézéquel, and N. Plouzeau, "Making Components Contract Aware," *Computer,* vol. 32, no. 7, pp. 38-45, July 1999.

[4] Y. Cheon and G.T. Leavens, "A Runtime Assertion Checker for the Java Modeling Language (JML)," *Proc. Int'l Conf. Software Engineering Research and Practice (SERP '02),* pp. 322-328, June 2002.

[5] Y. Cheon and G.T. Leavens, "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way," *Proc. 16th European Conf. Object-Oriented Programming,* pp. 231-255, 2002.
[6] J.O. Coplien, *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.
[7] A. Duncan and U. Hölzle, "Adding Contracts to Java with Handshake," Technical Report TRCS98-32, Univ. of California at Santa Barbara, Dec. 1998, http://www.cs.ucsb.edu/research/trcs/abstracts/1998-32.shtml.
[8] S.H. Edwards, "Representation Inheritance: A Safe Form of 'White Box' Code Inheritance," *IEEE Trans. Software Eng.,* vol. 23, no. 2, pp. 83-92, Feb. 1997.
[9] S. Edwards, G. Shakir, M. Sitaraman, B.W. Weide, and J. Hollingsworth, "A Framework for Detecting Interface Violations in Component-Based Software," *Proc. Fifth Int'l Conf. Software Reuse,* pp. 46-55, June 1998.
[10] S.H. Edwards, "Black-Box Testing Using Flowgraphs: An Experimental Assessment of Effectiveness and Automation Potential," *Software Testing, Verification and Reliability,* vol. 10, no. 4, pp. 249-262, Dec. 2000.
[11] R.B. Findler, M. Latendresse, and M. Felleisen, "Behavioral Contracts and Behavioral Subtyping," *Proc. Eighth European Software Eng. Conf. and Ninth ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 229-236, 2001.
[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
[13] C. Gülcü, *The Complete Log4j Manual.* QOS.CH, 2003.
[14] J.E. Hollingsworth, L. Blankenship, and B.W. Weide, "Experience Report: Using RESOLVE/C++ for Commercial Software," *Proc. ACM SIGSOFT Eighth Int'l Symp. Foundations of Software Eng.,* pp. 11-19, Nov. 2000.
[15] "iContract Plus-Making iContract Friendly," iContract home page, http://icplus.sourceforge.net/, Aug. 2004.
[16] The Java Modeling Language (JML) Home Page, http://www.jmlspecs.org/, Aug. 2004.
[17] R. Kramer, "iContract—The Java Design By Contract Tool," *Proc. Conf. Technology of Object-Oriented Languages (TOOLS 26),* pp. 295-307, 1998.
[18] G.T. Leavens, A.L. Baker, and C. Ruby, "JML: A Notation for Detailed Design," *Behavioral Specifications of Businesses and Systems,* chapter 12, pp. 175-188, 1999.
[19] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development.* McGraw-Hill, 1986.
[20] B. Meyer, "Applying 'Design By Contract'," *Computer,* vol. 25, no. 10, pp. 40-51, Oct. 1992.
[21] B. Meyer, *Object-Oriented Software Construction,* second ed. Prentice Hall, 1997.
[22] D.R. Musser, G.J. Derge, and A. Saini, *STL Tutorial and Reference Guide,* second ed. Addison-Wesley, 2001.
[23] D.L. Parnas, "A Technique for Software Module Specification with Examples," *Comm. ACM,* pp. 330-336, May 1972.
[24] D.E. Perry, "The Inscape Environment," *Proc. 11th Int'l Conf. Software Eng.,* pp. 2-12, May 1989.
[25] R. Plösch, "Evaluation of Assertion Support for the Java Programming Language," *J. Object Technology,* vol. 1, no. 3, pp. 5-17, 2002, http://www.jot.fm/issues/issue_2002_08/article1.
[26] D.S. Rosenblum, "A Practical Approach to Programming with Assertions," *IEEE Trans. Software Eng.,* vol. 21, no. 1, pp. 19-31, Jan. 1995.
[27] G. Shakir, "A Systematic Generator for Detecting Interface Violations in Component-Based Software," MS Report, Dept. of Computer Science and Electrical Eng., West Virginia Univ., Morgantown, 1997.
[28] "Special Section: Component-Based Software Engineering Using RESOLVE," M. Sitaraman and B.W. Weide, eds., *ACM SIGSOFT Software Engineering Notes,* vol. 19, no. 4, pp. 21-67, Oct. 1994.
[29] B. Stroustrup, *The C++ Programming Language.* Addison Wesley, 2000.
[30] C. Szyperski, *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.
[31] R.P. Tan and S.H. Edwards, "An Assertion Checking Wrapper Design for Java," Technical Report #03-11, Dept. of Computer Science, Iowa State Univ., Ames, Iowa, pp. 29-34, Aug. 2003, http://www.cs.iastate.edu/leavens/SAVCBS/2003/papers/full-papers/tan-edwards.pdf.
[32] N. Tan, C. Mingins, and D. Abramson, "Design and Implementation of Assertions for the Common Language Infrastructure," *IEE Proc.—Software Eng.,* vol. 150, no. 5, pp. 329-336, 2003.
[33] P. Vitharana, "Risks and Challenges of Component-Based Software Development," *Comm. ACM,* vol. 46, no. 8, pp. 67-72, Aug. 2003.
[34] J.M. Voas, "Quality Time: How Assertions Can Increase Test Effectiveness," *IEEE Software,* vol. 14, no. 2, pp. 118-122, Feb. 1997.
[35] J. Voas and L. Kassab, "Using Assertions to Make Untestable Software More Testable," *Software Quality Professional,* vol. 1, no. 4 Sept. 1999.
[36] C. Wang and D.R. Musser, "Dynamic Verification of C++ Generic Algorithms," *IEEE Trans. Software Eng.,* vol. 23, no. 5, pp. 314-323, May 1997.
[37] J.M. Wing, "A Specifier's Introduction to Formal Methods," *Computer,* vol. 29, no. 9, pp. 8-24, Sept. 1990.
[38] J.M. Wing, "Using Larch to Specify Avalon/C++ Objects," *IEEE Trans. Software Eng.,* vol. 16, no. 9, pp. 1076-1088, Sept. 1990.

Stephen H. Edwards received the BS degree in electrical engineering from the California Institute of Technology, and the MS and PhD degrees in computer and information science from the Ohio State University. He is currently an associate professor in the Department of Computer Science at Virginia Tech. His research interests include software engineering, reuse, component-based development, automated testing, formal methods, and programming languages.

Murali Sitaraman received the ME degree in computer science from the Indian Institute of Science, Bangalore, and the PhD degree in computer and information science from The Ohio State University. He is an associate professor of computer science at Clemson University. He directs the Reusable Software Research Group (RSRG) at Clemson. He is a principal investigator of the RESOLVE research effort (www.cs.clemson.edu/~resolve). His research interests span foundational, practical, and educational aspects of software engineering. He served as the program chairman of the Fourth International Conference on Software Reuse in 1996. He coedited a book on foundations of component-based systems published by Cambridge University Press in 2000. He is a member of the IEEE Computer Society and ACM.

Bruce W. Weide holds the PhD degree in computer science from Carnegie Mellon University and the BSEE degree from the University of Toledo. He is a professor of computer science and engineering at The Ohio State University, where he codirects the Reusable Software Research Group. His research interests include all aspects of software component engineering, as well as teaching component-based software principles to beginning CS students. He and colleague Tim Long were awarded the IEEE Computer Society's 2000 Computer Science and Engineering Undergraduate Teaching Award for their work in the latter area. He is a senior member of the IEEE, and a member of ACM and CPSR. For details see http://www.cse.ohio-state.edu/~weide.

Joseph Hollingsworth is a computer science professor at Indiana University Southeast. His primary research interests lie in software component engineering where, most recently, he has applied research results in software component engineering to the development of commercial software. He has also introduced knowledge from research into classroom instruction and bases his junior-level data structures course on personal notes developed from software component research.