

Integrating Interface Assertion Checkers into Component Models

George T. Heineman
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA
(508) 831-5502
heineman@cs.wpi.edu

ABSTRACT

Run-time enforcement of behavioral contracts has been studied extensively in procedural and object-oriented languages. This research has led to a better understanding of specific techniques, including pre-processing compilers or wrappers. However, component-based software engineering (CBSE) imposes additional restrictions and it is appropriate to consider how to extend these techniques when the software is decomposed into independently-developed third-party components. In this paper we identify some requirements for integrating run-time enforcement of behavioral contracts into the component model and illustrate a solution using a scaled-down component model and example. The primary result is that a standardized service should be added to component model implementations to enable application assemblers to enforce *local* properties as specified by the components in the application as well as *global* properties as specified by the application.

Categories and Subject Descriptors

F.3.1 [Logic and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs – *assertions, invariants, mechanical verification, pre- and post-conditions*. D.2.4 [Software Engineering]: Software/Program Verification – assertion checkers, programming by contract.

General Terms

Languages, Verification.

Keywords

Interfaces, Component Models, Run-time behavioral contracts.

1. INTRODUCTION

The full promise of component-based software engineering (CBSE) will only be realized when an efficient marketplace of software components is created. In such a market, component vendors must be able to sell software components with clear

specifications. Application assemblers must be able to locate software components that satisfy functional criteria and must be able to verify that the software component conforms to its specification. Ideally, as pointed out by Beugnard *et al.*, one should be able to determine at design time that a particular component assembly (i.e., a set of interacting components) can satisfy all required properties. In the context of this paper, a *property* is limited to a user-defined assertion that can be evaluated either before or after a method invocation as defined in an interface. As an intermediate goal, most software engineers would likely accept a practical solution that employs a run-time enforcement strategy to detect when properties are invalidated by a component assembly. For such an approach to become widespread it must be part of the component model otherwise, we would be faced with numerous *ad hoc* solutions, of varying quality, and the potential that the same functionality would be implemented countless times over.

The industrial-scale component models of EJB [7], COM+ [8], and CORBA CCM [9] have already matured to the point where they all support a core set of capabilities including transactions, persistence, and security. While these capabilities are important to ensuring the viability of the technologies, they do nothing to enhance the trust between application assemblers and component vendors. We argue that the time has come to apply existing techniques to validate the behaviors of software components at run-time. In this paper we outline requirements for having run-time enforced behavioral contracts fully integrated with a component model. We propose a solution that more closely involves the component model implementation than any found in the literature to date. We describe a running example using a small component model that illustrates the need for run-time enforcement and shows how it could be integrated. We close with some open issues.

2. REQUIREMENTS

Since CBSE involves several stakeholders, the requirements are divided by category.

2.1 Component Vendor

Standardized interfaces must be defined together with their appropriate behavioral contracts. This is an obvious requirement, but it highlights an important issue. The contracts for an interface can only contain *local assertions*, that is, assertions on the component providing (i.e., implementing) the interface. While it is desirable to be able to place restrictions on the environment within which the component will execute (consider the Interface

Automata of Alfaro and Henzinger [10], for example), this requires more powerful mechanisms than described in this paper and is left for future work.

The component source code must remain private to the vendor. This non-disclosure is essential from a business perspective and it immediately invalidates any attempt at run-time enforcement that requires the application assembler to pre-process or otherwise manipulate all source code for the application.

2.2 Application Assembler

The application assembler must be able to specify *global properties* to be enforced at run-time. We consider these to be global since they cannot be confined simply to one interface or even to one component. These properties may be compositional and only appear from the unique combination of components within an application. In general, these are most commonly associated with the way in which an interface is used. For example, a clipboard component could store arbitrary items for copy/paste, but in a particular application, no color images are ever copied to the clipboard. These user-defined properties are essential for troubleshooting the use of black-box components.

2.3 Component Model Vendors

Component model implementations (CMI) [5] are most often based on an agreed upon standard; in some cases, a CMI only has a single vendor and can become a *de facto* standard with enough users (witness Microsoft and COM). Many CMI vendors provide extra value-added services above and beyond what is expected from the standard. By analogy with database management system (DBMS) vendors, query optimization is not a defined part of the core relational database model; DBMS vendors, however, provide built-in query optimization because of the impracticality of expecting DBMS users to perform this task. The primary requirement placed on CMI vendors is that the service for run-time enforcement of behavioral contracts must not force the vendors or the application assemblers to alter their development processes or require developers to change the way they code.

3. OVERVIEW

To restate the problem we are investigating, an application assembler desires to enforce at run-time all the known behavioral contracts of the components within the application. In addition, new user-defined properties are incrementally discovered and added (possibly even removed) by the assembler. These properties are encoded as assertions that could be validated as pre-conditions (or post-conditions) to a method of an interface.

3.1 Component Model Example

The small block component model we use in this paper illustrates the need to involve the component model during the run-time validation of behavioral contracts. A block application is composed of block components. Each block component must explicitly declare its provided or required interfaces. A component application file (CAF) specifies the topology of the application (i.e., the interconnections of components). The block CMI provides a container that loads the CAF file, manages the life cycle of the constituent components, and launches the application. A sample CAF file for our running example is shown in Figure 1; GraphEditor requires the interface *text.IClipboard* which is

```
<APPLICATION>
<COMPONENTS>
<COMPONENT name="Editor" type="edu.wpi.cs.GraphEditor">
<COMPONENT name="Clipboard" type="edu.wpi.cs.Clipboard">
</COMPONENTS>
<CONNECTIONS>
<CONNECTION source="Editor" target="Clipboard"
interface="text.IClipboard">
</CONNECTIONS>
</APPLICATION>
```

Figure 1: Sample CAF example.

provided by Clipboard in this case. The connections in CAF are analogous to cross-references in Enterprise Java Beans (EJB) [7]. In EJB, the *ejb-jar* deployment file encodes the dependency information in similar fashion.

3.2 Component Substitutability

It is certainly reasonable to allow an application assembler to include an enhanced clipboard component that provides an interface extending *text.IClipboard*; in this circumstance, it is a natural question to ask whether the composition is valid, that is, whether all behavioral contracts are satisfied.

The principles of covariance and contravariance have been applied to objects as follows: Consider the following interface with a behavioral contract based on pre- and post-conditions:

```
interface text.IClipboard {
    insert (Item i) : void
    pre (i null)
    post (size() = size() @ pre + 1) and (i=retrieve() @ next)

    retrieve () : Item
    pre (size() >= 1)
    post (IClipboard =IClipboard @ pre)
}
```

Here, *retrieve()* is valid when at least one item is in the clipboard and makes no changes; *insert(i)* increases the size of the clipboard and ensures that *i* is the next object retrieved.

The interface *text.IClipboard* is standardized and the vendors of GraphEditor and Clipboard have agreed to the standard. The GraphEditor is unaffected when an enhanced Clipboard component, ClipboardGUI, is substituted for Clipboard under the following conditions: (1) *covariance* – ClipboardGUI provides more than is required by *text.IClipboard* and ClipboardGUI may return more specified types (in this case, retrieve could return a further refinement of Item); (2) *contravariance* – ClipboardGUI expects less than demanded by *text.IClipboard* and it may allow more general types as input (in this case, insert could accept a more general Item). In the absence of more rigorous, formal approaches, the application assembler would like to validate that the component replacement satisfies both conditions. In this example, the new interface, *text.IClipboardGUI*, extends *text.IClipboard* and there are known techniques for properly assigning blame at run-time when a *hierarchy extension* fault occurs (see Findler *et al.* [1]).

3.3 Active Interfaces

In prior work [11], we describe *active interfaces*, a technology for creating black-box adaptable software components. We can use this technology to directly support the run-time enforcement of behavioral contracts. An active interface decides whether to take action when a method is called; there are two phases to all interface requests: the “before-phase” occurs before the component performs any steps towards executing the request; the “after-phase” occurs when the component has completed all execution steps for the request. These phases are similar to the Lisp advice facility described in [13]. Instead of adapting the behavior of the component, the active interface will enforce its behavioral contracts at run-time. Active interfaces are different from wrappers in the following ways:

- Active interfaces are compiled into a black box component and do not suffer the overhead associated with wrappers. In addition, there are complications when multiple components must communicate with each other while they are each contained within some form of wrapper [15].
- All methods, whether private or public, can be augmented with an active interface. A wrapper approach will be unable to interpose its logic when the method cannot be directly invoked externally. This could be useful, for example, to check conditions when a subclass overrides a protected abstract method.
- Wrappers fail to capture the internal invocations that the encased component makes on its own methods; active interfaces will be capable of enforcing contracts whenever a method is invoked, regardless of whether the callee is internal or external to the component.

Without additional tool support, the vendor would have to expose its source code to enable the application assembler to insert the active interface into the components; this would violate a core requirement as described in Section 2. We have built an Active Interface Development Environment (AIDE) tool that automatically instruments source code to contain the appropriate *hooks* at the before- and after-phase of a component’s methods [14]. These hooks do not actually perform any enforcement checks; they only make it possible for the CMI to do so at run-time. The vendor simply instruments the component before releasing it as a black-box component. The application assembler then can add properties of her own to be checked at run-time.

3.4 Specification Language

The object constraint language (OCL) is the language used within a UML diagram to define precise constraints over the methods of interfaces and classes [18]. While OCL is a powerful language it is not widely used within industry; in addition, there are skeptics who propose simpler languages that are equally powerful, such as Alloy [19]. For the purpose of this paper, we have focused solely on the need to declare pre- and post-conditions for class methods. Within a post-condition for an operation, there is a need to refer to a value that was present before operation was executed (i.e., to ensure that a value was properly incremented). In OCL and other similar languages, a post-condition for a method might specify “count = count @ **pre** + 1” which declares that the value of count after the method is invoked is one larger than before. We have developed for this paper two new concepts: **next** and **past**.

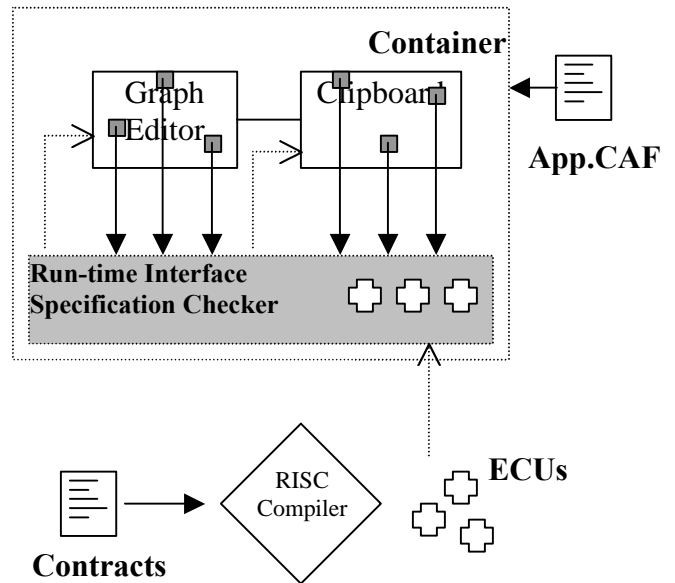


Figure 2: Run-time Interface Specification Checker.

In the same way that “@ **pre**” refers to the value of a state attribute at the time the method was invoked, “@ **next**” refers to the value returned by a method of the interface the next time it is invoked (if ever); as such, this is akin to an obligation that the interface places on itself in the future. Similarly, “@ **past**” refers to the value returned by a method of the interface at some time in the past. There will naturally be extra state overhead for the @**pre**, @**next**, and @**past** constructs but we have not yet carried out any through evaluation.

4. PUTTING IT ALL TOGETHER

4.1 Building the component

The component vendor is free to operate as before with one additional step prior to deployment. The vendor may choose to release multiple “binary” versions of its component, one with and one without the requisite contract checking code. This decision is analogous to deciding whether to compile with the `-g` debugging flag enabled. These instrumented components are packaged and deployed as stipulated by the component model.

4.2 Compiling contracts into code

Given the full interface examples as outlined in Appendix A, the Run-time Interface Specification Checker (RISC) compiler converts the pre- and post-conditions of each interface method into executable checking units (ECUs) as shown in Figure 2. These units are registered with the RISC monitor when the components are instantiated (this interaction is shown as dashed lines from the RISC into the individual components). The component vendor could carry out this task only for the constraints on the component’s interfaces. If the application assembler has other constraints that must also be checked, these constraints can also be provided to the RISC compiler before executing the component-based application.

4.3 Enforcing behavioral contracts

As the application executes, the instrumented black-box components make outgoing calls to RISC to ensure that the contracts are satisfied. Note that we separate the policy of how the contracts are enforced from the built-in mechanism that requests

the checks on demand when the interface is invoked at run-time. This separation enables us to incrementally increase the expressive power of the contracts whose enforcement is assured by RISC. When a contract is invalidated, RISC contacts CMI, which can then alert the responsible party (in similar fashion to the approach outlined by Findler *et al.* [1]) and possibly take other preventative or recovery measures. In this way we once again separate the reaction to the invalidation of a contract from the detection of that invalidation.

The solution outlined above is incremental, and can accommodate a growing (or shrinking) set of properties to be validated without forcing the recompilation of the original components or even the application; only the ECUs need to be regenerated and compiled.

The component model plays an important role in defining standards for how components should react to these run-time error situations. It is simply not sufficient for a component to throw an exception when a client incorrectly invokes its interface. For example, the component could log the error situation and reset itself to its initial state for future interactions. We assume that the component model will ultimately provide the ability for the component to declaratively specify its error handling policy.

As described in Section 3.2, a new interface, *text.IClipboardGUI*, has been developed by extending an existing interface, *text.IClipboard*. In Figure 2, the Graph Editor component requires some component to provide *text.IClipboard* (in this case, a default Clipboard Component (DCC) performs this logic). We are tasked with determining whether an advanced Clipboard Component (ACC) can be substituted in place of DCC. There are two situations that can be detected by our assertion checker. First, the implementation of ACC may not actually satisfy the post-conditions as declared in the *text.IClipboard* interface. Naturally, one would expect that the vendor has ensured the proper execution of its component, but from a practical perspective, we can only increase our confidence in external code by validating its proper function. Second, we can ensure that the vendor has properly built ACC by only weakening its pre-conditions and strengthening its post-conditions.

4.4 Evaluation

We have developed a parser for the constraint language outlined by example in Appendix A. We are currently investigating the overhead of the RISC approach, especially with regards to the computational overhead of the active interfaces instrumented in the components. AIDE already has the capability of selectively enabling (or disabling) the methods of an active interface; this capability can enable the application assembler to tune the performance of the assertion checkers in a manner similar to how the latest version of the Java™ virtual machine enables fine-grained control over assertions [21]. We will also investigate the computational overhead of maintaining state information for components to verify post-conditions.

5. RELATED WORK

Findler *et al.* describe weaknesses in existing run-time contract checkers for Java programs and proposes a sound solution based on the behavioral subtyping [1]. Rather than introducing a more complicated contract checker, Findler *et al.* generate interface-checking code and embed wrapper-like methods within the source code. They are able to properly assign blame in the face of complex class hierarchies and interface extensions. Interestingly,

the original methods of the classes are still available for invocation and do not therefore have any behavioral checks at all. Their pre-processor rewrites calls to the original methods into calls to the appropriate wrapper method, so this validation process affects the original client. Since the generated interface checkers contain the predicates to be validated, this approach will only work for local assertions; the violation of global properties cannot be detected by this mechanism.

Several approaches for testing components rely on the ability to decorate (or otherwise wrap) a software component to insert hooks for validation. Edwards has developed a built-in-test (BIT) capability that adorns components with hooks for self-checking and self-testing features [2][3]. The actual component is encased within the wrapper. BIT wrappers are transparent to the encapsulated component; however, the wrappers are only transparent to the client code invoking the component if the client maintains only indirect access to the underlying component, through a factory for example. If, for example, the client directly accesses the constructor, then the wrapper is bypassed. In addition, if the encapsulated component invokes one of its own methods, the BIT wrapper is not invoked; for local assertions this is of no consequence, because one could assume that the component can safely use its own methods at any time. However, if there were a desire to validation extra-component or other global properties at every access to the component, the wrapper approach would be unable to perform its duties. Finally, BIT performs its operations on copies of the encapsulated component's abstract state, which imposes additional overhead; this may be unavoidable when ensuring post-conditions that refer to prior state, as we also encounter with RISC.

Aspect-oriented programming (AOP) provides a vision of programming that separates concerns [20]. One can view the separation of constraints from components as being inspired by the AOP community. The AIDE tool in particular instruments components in a way similar to how aspects and ordinary code are "weaved together" in AOP. It would be worthwhile to evaluate using an existing AOP tool to weave aspects into Java code to achieve the same results as AIDE.

6. FUTURE CHALLENGES

As described earlier, an interesting avenue for future work is to capture in an interface the restrictions that are to be placed on the environment within which a component providing that interface will execute. It may be that these restrictions are more appropriately associated with a specific component realization than the abstract interface; for example, an application assembler may be willing to select a component which provides an interface but has a long list of restrictions.

After the initial success of our prototype, we are now investigating expanding the specification power to support a subset of the Bandera Specification Language (BSL) properties [16]. We need to investigate the tradeoff between selecting the minimal features of a specification language and the costs of implementing these features. We will also need to acquire more experience in the types of constraints that application assemblers wish to write.

The benefit of using active interfaces is that the approach is language independent and does not depend upon language features to be supported. To date, we have used active interfaces

with C and Java programs so these results are immediately applicable to numerous component-based systems.

7. ACKNOWLEDGMENTS

The author would like to thank Kathi Fisler, Shriram Krishnamurthi, and Robert Findler for their comments on the ideas that led to this paper. Paul Calnan [17] is helping implement RISC as part of his MS thesis. This work is sponsored in part by National Science Foundation grant CCR-9733660 and by the Defense Advanced Research Project Agency under DARPA Order K503 monitored by Air Force Research Laboratory F30602-00-2-0611.

8. REFERENCES

- [1] R. Findler, M. Latendresse, and M. Felleisen, “Behavioral Contracts and Behavioral Subtyping”, *Proceedings, Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Vienna, Austria, Sep. 2001, pp. 229—236.
- [2] S. Edwards, “A Framework for Practical, Automated Black-Block Testing of Component-Based Software”, *Journal of Software Testing, Verification, and Reliability*, Vol. 11, No. 2, 2001, pp. 97—111.
- [3] S. Edwards, G. Shakir, M. Sitaraman, B. Weide, and J. Hollingsworth, “A Framework for Detecting Interface Violations in Component-Based Software”, *Proceedings, 5th International Conference on Software Reuse*, June 1998, pp. 46—55.
- [4] C. Szyperski, *Component Software*, Addison-Wesley, Boston, MA, 1998.
- [5] G. Heineman and W. Councill, Editors, *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, Boston, MA, 2001.
- [6] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, “Making Components Contract Aware”, *IEEE Computer*, Vol. 32, No. 7, June 1999, pp. 38—45.
- [7] Sun Microsystems, Enterprise JavaBeans Specification, v2.1, java.sun.com/products/ejb/docs.html.
- [8] T. Ewald, *Transactional COM+: Building Scalable Applications*, Addison-Wesley, Boston, MA, 2001.
- [9] N. Wang, D. Schmidt, and C. O’Ryan, “Overview of the CORBA Component Model”, in [5], Chapter 31.
- [10] L. de Alfaro, T. Henzinger, “Interface Automata”, *Proceedings, Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, Vienna, Austria, Sep. 2001, pp. 109—120.
- [11] G. Heineman, A model for designing adaptable software components, *Proceedings, 22nd International Conference on Computer Software and Applications Conference (COMPSAC)*, Vienna, Austria, Aug. 1998, pp. 121—127.
- [12] N. Medvidovic, D. Rosenblum, and R. Taylor, “A Language and Environment for Architecture-Based Software Development and Evolution”, *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999, pp. 44—53.
- [13] GNU Emacs Lisp Reference Manual, Chapter 17: Advising Emacs Lisp Functions, www.gnu.org/manual/elisp.
- [14] P. Gill, *Probing for a Continual Validation Prototype*, MS Thesis, WPI Computer Science Department, Aug. 2001.
- [15] Urs Hölze, Integrating independently-developed components in object-oriented languages. In O. Nierstrasz, editor, *Proceedings ECOOP’93, LNCS 707*, Kaiserslautern, Germany, Springer-Verlag, July 1993, pp. 36—56.
- [16] J. Corbett, M. Dwyer, J. Hatcliff, and Robby, “A Language Framework For Expressing Checkable Properties of Dynamic Software”, *Proceedings of the SPIN Software Model Checking Workshop, Lecture Notes in Computer Science*, Springer-Verlag, Aug. 2000.
- [17] P. Calnan, *Extract: Extensible Transformation and Compiler Technology*, MS Thesis, WPI Computer Science Department, expected May 2003.
- [18] J. Warmer and A. Kleppe, *The Object Constraint language. Precise Modelling with UML*, Object Technology Series, Addison-Wesley, 1999.
- [19] D. Jackson, I. Shlyakhter, and M. Sridharan. A Modularity Mechanism. *Proceedings of the joint ACM SIGSOFT Conference on the Foundations of Software Engineering (FSE) and European Software Engineering Conference (ESEC)*, Vienna, Austria, Sept. 2001, pp. 62-73.
- [20] A. Lai, G. Murphy, and R. Walker, Separating Concerns with Hyper/JTM: An Experience Report, *Workshop Proceedings: Multi-dimensional Separation of Concerns in Software Engineering*, Limerick, Ireland, 2000, pp. 79—91.
- [21] Sun Microsystems, Java 2 Platform, Standard Edition (J2SE), java.sun.com/j2se/1.4.1.

Appendix A

In this example, there is an interface for a component that provides standard clipboard functionality. A more powerful version of the clipboard is available, complete with its own GUI interface, as shown in Figure 3.

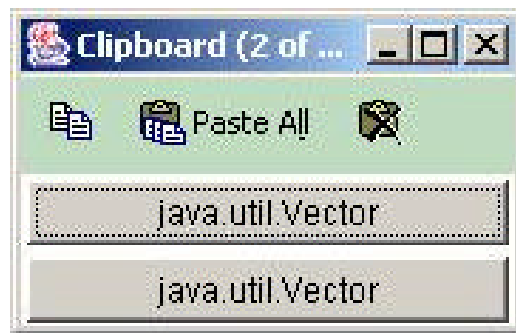


Figure 3: Sample GUI for Clipboard

The enhanced clipboard component provides the IClipboardGUI interface, which extends the previously defined IClipboard interface.

```

interface IClipboard {
  clear () : void
    pre (none)
    post (size() = 0)

  insert (IItem i) : void
    pre (i != null)
    post (size() = size() @ pre + 1) and (i=retrieve() @ next)

  newDataTypeInfo (String s) : IDataTypeInfo
    pre (s != null) and (Class.forName(s) != null)
    post (none)

  newItemInstance (IDataTypeInfo type, Object o) : IItem
    pre (type = newDataTypeInfo()) @ past
    post (none)

  retrieve () : IItem
    pre (size() >= 1)
    post (none)

  retrieveNth (int n) : IItem
    pre (size() >= n) and (n >= 1)

```

```

    post (none)

  types () : Iterator
    pre (none)
    post (none)
}

interface IClipboardGUI extends IClipboard {
  setAccessor (IClipboardAccessor ica) : void
    pre (ica != null) and (accessor = null)
    post (accessor = ica)

  showClipboard (boolean flag) : void
    pre (accessor != null)
    post (none)

  // weaken the pre-condition to accept any
  insert (IItem i) : void
    pre (none)
}

```

All exceptions thrown by methods in these interfaces are omitted for space reasons. This example is formatted for readability.