Toward Reflective Metadata Wrappers for Formally Specified Software Components

Stephen H. Edwards
Virginia Tech, Dept. of Computer Science
660 McBryde Hall
Blacksburg, VA 24061-0106 USA
+1 540 381 3020
edwards@cs.vt.edu

ABSTRACT

Abstract behavioral specifications for software components hold out the potential for significantly improving a software engineer's ability to understand, predict, and reason soundly about the behavior of component-based systems. Achieving these benefits, however, requires that specifications be delivered along with components to the consumer. This paper considers the question of what is the best way to package specification and verification information for delivery along with a component. Rather than distributing specifications in "source" form, an alternate solution based on reflection is presented. A reflective interface that supports program-level introspective access to behavioral descriptions is proposed. By embodying this interface in a wrapper component, it becomes possible for the reflective interface to also support services for contract violation checking, self-testing, and abstract value manipulation, even when the underlying component technology does not have built-in reflection capabilities.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, programming by contract, assertion checkers, class invariants*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*specification techniques, pre- and post-conditions, invariants, assertions*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object-oriented programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*.

General Terms

Design, Verification.

Keywords

Formal specification, reflection, design by contract, representation invariant, wrapper class, unit test, integration test.

1. INTRODUCTION

Component-based software development (CBSD) is becoming more prevalent every day, carrying with it the hope for greater productivity and software quality. Indeed, off-the-shelf components should be well-seasoned, well-tested, and more reliable than newly written code. An even greater benefit potentially can be provided by well-designed software components, however: they—or more correctly, the abstract specifications that explain their behavior—can help software engineers understand, predict, and reason soundly about the dynamic behavior of component-based software systems. Efforts at formally specifying the behavior of software components aim at maximizing this effect.

To achieve this benefit for a commercial component, naturally the component's (formal) specification must be distributed along with the component itself. For most commercial component technologies, including COM and its derivatives, CORBA, JavaBeans, ActiveX, and .NET, components are distributed in a binary form. Indeed, a central issue in reasoning about component-based software is the problem of correctly reasoning about composite behaviors when source code is unavailable. If the component provider is going to deliver a behavioral specification as well, in what form will it be delivered?

This position paper explores the question of how best to package and deliver a component's formal specification, as well as associated non-code information, along with the component's implementation. Section 2 outlines the problem, while Section 3 sketches a possible solution: delivering a wrapper component that provides access to a wide variety of information, including specification details, through a standardized, reflection-based interface. Section 4 explores the various kinds of metadata and services that may be appropriate to provide through such a reflective wrapper. Section 5 summarizes the limitations of the approach, Section 6 outlines relations with previous work, and Section 7 summarizes the issues covered.

2. PROBLEM AND SIGNIFICANCE

The problem under consideration, as introduced in Section 1, supposes that a component provider also wishes to provide a formal specification (perhaps along with verification information) when a component is delivered:

> *What is the best way to package specification and verification information for distribution to clients along with a component?*

One of the primary driving factors in binary packaging of commercial components is protection of proprietary information or trade secrets embodied in the component's implementation. This concern does not arise with specifications, of course. The client cannot receive many of the benefits of having a component's behavioral specification unless the specification is completely accessible. "Hiding" a specification clearly is at direct odds with the value added by distributing it with a component in the first place. This leads to the naïve view that a formal specification should be delivered in human-readable "source" form. Even something as

simple as a text file containing the specification in a suitable formal notation should suffice.

This simple approach to distributing specifications treats them the same as traditional documentation, which is distributed most frequently in printed form, as plain text, as HTML, or in a platform-specific help file format. Besides simplicity, source distribution of specifications has another strength: it highlights and reinforces the fact that specifications are designed for *communication* with software engineers—they are written for other people to read. In essence, isn't a formal specification the ultimate in rigorous documentation?

While the value of reading specifications cannot be underestimated, treating specifications as "plain old documentation" misses the opportunity to appropriately leverage those specifications in development tools, during component composition, and during automated reasoning or verification tasks. It is certainly possible to force every CBSD tool to pick some specification notation, support parsing/internalization of that notation, and maintain its own representation of the relations between specifications and components. Unfortunately, this strategy implies a huge duplication of effort among CBSD tool implementers, conflicting choices made in different tools, and a number of other inefficiencies. As a result, in asking what is the "best" way to package specifications for distribution, this paper is aiming to support both human and tool consumption and use of specifications.

The significance of this problem to researchers in formally specified components is apparent: a component's client cannot reap many of the benefits of a specification unless the specification is delivered with the component. Similarly, the specification should be delivered in a form that conveniently supports all of the activities the client may wish to perform, including both people-oriented and tool oriented tasks.

3. A POSSIBLE SOLUTION: REFLECTIVE METADATA WRAPPERS

If providing specifications (and other related information) as traditional documentation has disadvantages, what alternatives are available? Consider the history of software components. Providing specifications in source form is analogous in some ways to the "old days" when reusable components were subroutine libraries shared as source code files among programmers. While the client could always refer to the code, the critical interface information (the name, parameter profile, description, and usage of each subroutine) was typically provided in embedded comments or as separate documentation.

Component packaging and distribution has evolved enormously since subroutine libraries first came into use, however. While traditional documentation typically is provided for commercial components, most commercial component technologies, including COM, CORBA, JavaBeans, ActiveX, and .NET all provide some API for "inspecting" a component's interface. In effect, a component "knows" what it exports, and a client can use a well-defined interface to "ask" what operations are available, how many parameters of what type are needed, what properties are provided, and so on. Such an interface is a wonderful boon to component packaging. It naturally allows any development environment to immediately manage and support any newly installed component, it supports "plug-in"-style integration of new features in applications, it supports automatic checks for (syntactic) interface compatibility during composition, and it can even support general-purpose component-level scripting tools in some cases.

3.1 Reflect for a Moment

The ability of a component to respond to queries about its own structure, behavior, or implementation is the cornerstone of **reflection** [24, 11]. Reflective software is capable of representing (and thus operating on at run-time) some aspect(s) of itself. **Computational reflection** is the activity of a computational system when computing about or operating on (and thus potentially altering) its own computation [17]. This concept arose in the programming language arena, and has seriously impacted object-oriented programming language design.

A reflective component can provide two different forms of reflection services: **introspective** capabilities provide read-only access for inspecting component properties, while **intercessory** services allow one to modify a component or alter its behavior in some way [11]. While intercessory protocols are at the heart of computational reflection and metaprogramming [11, 16], the more restricted introspective protocols supported by most component technologies are still powerful tools. In effect, "interface inspection" APIs supported by most commercial components are simply scaled down introspective interfaces.

If a typical component (say, a JavaBean) already supports a standardized interface for reporting on the syntactic properties of its exported features, how difficult can it be to extend that interface to include access to specification-level descriptions? Perhaps the more interesting question is how far can this strategy be taken?

Many OO languages that support reflection, including Java, do so by associating each object with a metaobject that encapsulates information about how that object is structured and how it behaves. Often in class-based OO languages, an object's metaobject is a singleton object representing its class. This class object supports methods to determine the name and parameter profile of the methods supported by instances of the class, the name and type of instance and class-wide data members, the name and number of superclasses, and so on. A class object may also offer intercessory capabilities, such as changing the way method dispatch is supported. Normally, supporting intercessory capabilities requires the metaobject approach to be built-in to the language.

By analogy, it is possible to turn a behavioral specification into a stand-alone component that provides a standard interface for inspecting all facets of the specification. Whereas a traditional OOPL "class" object represents a class' structural or syntactic interface, a specification object instead represents a specification's structural *and behavioral* description. All of the normal reasoning and structuring techniques applied to abstract class collections and hierarchies could also be applied to specification objects. In effect, this strategy turns a behavioral specification into *another operational component* that can be delivered alongside the original component it describes.

Now providing specification information for components appears to be a simple matter: simply design a metaobject system similar to that used in a class-based OOPL, except that metaobjects model and allow access to formal behavioral descriptions instead of simply syntactic interfaces. If we wish to limit ourselves only to introspection, this approach may be satisfactory. However, intercessory services cannot easily be added through specification

objects alone if one is working using an existing component technology where metaobject-based reflection is not built-in to the component model.

3.2 Decorating With Wrappers

Specification objects are a powerful idea, particularly for providing introspective capabilities and for sharing specifications among behaviorally interchangeable components. Introducing them requires little more than adding some kind of `GetSpecification()` method to a component's interface. On the other hand, can intercessory services (that allow changes to component-level behavior) be added to components that are implemented using a non-reflective language or component technology?

It is possible to add some (but obviously not all) intercessory capabilities to any component with the correct design. The **decorator** pattern [7] suggests a simple approach that is suitable to the situation at hand: add the reflective interface by packaging the new operations in a wrapper component. This wrapper should conform to the original specification, but will delegate all of the work involved in the original operations to the component it wraps. We can call such a component a **reflective specification wrapper** (or simply reflective wrapper). At a minimum, this reflective wrapper provides the `GetSpecification()` access to a stand-alone specification object. Further, by interposing a separate processing layer between the client and the underlying component, it becomes possible to add or remove features before or after component operations. This supports a degree of intercessory customization—here, changing the behavior of the wrapper by turning some features on or off, rather than modifying the behavior of the underlying component. Appropriately exploiting this customization from the point of view of behavioral specifications is discussed in Section 4.

While neither the use of wrappers nor the use of reflection is a new idea, the novelty lies in combining the two to provide program-level access to specification information. First, the specification information is clearly turned into another component that can be delivered alongside the original. Further, rather than placing more operations and data inside the underlying component, such a wrapper isolates these features in a separate layer between the component and its client(s) (which now may include a host of development tools in addition to other application code). This approach, which is more in-line with object-oriented design, separates the added features from the underlying code in a way that can be made completely transparent to the remainder of the application, that supports easy insertion or removal of the added capabilities, and that naturally fits with conventional component distribution techniques.

Placing specification-oriented reflection features in a separate class or component is a simple idea, but it refocuses attention with dramatic results. It elevates the reflection features from the level of one or two methods in a component interface up to the level of a separately useful component abstraction. This elevation shifts attention to the question of exactly what "meta" data or services should be provided by a reflective specification wrapper.

3.3 Summarizing the Proposed Solution

The position espoused in this paper is that a component's specification (and other supporting information) should be provided *as another component* (or set of components), distributed in the normal fashion. This position is founded on three insights:

1. Reflection supports both human-readable and tool-based access to and application of the needed information. Reflection naturally supports standardized smart browsing tools and other document navigation aids for human understanding [6], while it also supports uniform automated services that rely on specification data without requiring the duplication of effort necessitated by source code distribution.
2. Wrappers can be used to transparently add features to a component without affecting the underlying entity. Further, they add the ability to support limited forms of intercessory reflection, even when such features are not directly supported in the underlying component technology.
3. If component instances are created using factories [7], client code is completely insulated from dependencies on the concrete implementation used for each instance. This can encapsulate and even parameterize wrapping decisions, so that reflection services can be employed when needed or stripped out when unnecessary without altering clients.

While the position presented here is founded on a wide-ranging collection of prior research, reflective specification wrappers in the form described here have not yet been implemented. Instead, this paper explores the issues and possibilities arising from the proposed approach, both to highlight the problem of packaging and distribution of specifications and to suggest a potential solution for exploration.

4. WHAT METADATA AND SERVICES ARE NEEDED?

If one wishes to provide specification (and verification) information through a reflection interface embodied in a wrapper component, the next issue to face is the question of what data and/or services to support through this interface. As is traditional with reflection, the component information we are concerned with here is truly *metadata*, in the sense that it describes the nature of the component and how it behaves, in contrast to the *data* that the component computes with or transforms. But exactly what metadata or intercessory services should be supported?

4.1 A Component's Formal Specification

The most obvious metadata to provide is some representation of a component's formal specification. Just as a conventional metaobject protocol provides introspective access to an object's class, its methods, and its fields, a reflective specification wrapper should provide introspective access to all aspects of a component's formally specified **exported interface**. This is the role of the specification objects introduced in Section 3.1.

For a model-based component specification [28], the specification object could provide access to the object's **abstract model**, to the **pre- and postcondition for each operation** or method, and to the object's **abstract invariant**. If an algebraic specification approach were used, access would instead be oriented toward axioms and algebras. Beyond the basics, access to publicly available fields or properties, exception behavior, and relationships to other specifications (such as inheritance, perhaps) also need be considered.

However, as Szyperski notes in his definition, there is more to a component than just an exported interface: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only" [25]. This perspective is also shared by the 3C model [15]. As a result, it is clear that in addition to the exported interface, a reflective wrapper should also provide introspective access to a component's **imported interface**: that is, the explicit context dependencies it places on its environment. While the exported interface captures the contract between a component and its client, the imported interface forms the contract(s) between a component and the other, lower-level components on which it is built.

Taken together, providing program-level access to a component's import and export interfaces seems like the bulk of the problem when it comes to packaging and distributing a component's formal specification. On the other hand, elevating the reflection interface to a separate component focuses attention on the other information and services that can be provided through such an object.

4.2 A Component's Verification History

While one's initial concern will necessarily be with distributing specification information, in the long term, consideration of verification information will also be useful. Was a component formally verified? By hand? With tool assistance? Was model-checking used instead? Or was a testing-based approach used? Is a proof or proof fragment available? Upon what assumptions is the verification based?

The extent and quality of verification performed on a component is clearly of interest to the client. In many cases, this information is most useful before making a component purchasing decision. Further, in the ideal situation where local certifiability (also called the modular reasoning property) [26] is ensured by all components in an application, there would be little need for verification details by the client after purchase. However, without local certifiability, verification details are important in supporting application-level verification of component compositions. Further, if formal verification is not used systematically throughout an application, component-level verification information may be useful in localizing defects during testing.

4.3 Violation Checking Services

Component-based development highlights the differing needs and perspectives of the component-provider and the component-user [8]. It is important to provide powerful capabilities for establishing a component's quality to the component-provider. The component-user, on the other hand, must also be provided with the services and information necessary to test her application in combination with the component.

One approach to addressing both concerns is checking interface contracts for violations. In addition to simply providing access to specification and verification information, a reflective wrapper can also provide contract checking features. Because of the way the wrapper is interposed between the client and the component, it is easy to add any or all of the following run-time checks:

- Precondition checks
- Postcondition checks
- Abstract invariant checks
- Representation invariant checks

This idea, originally proposed by the author and colleagues [3], has been used with some success [9, 2]. Postcondition and invariant checking are extremely useful to the component provider during development [1, 2, 19], while precondition checks are useful to the client during component integration. Postcondition and invariant checking can also be useful to the client in defect localization during application testing.

A carefully designed reflective wrapper could allow each category of checks to be enabled or disabled, perhaps on a per-operation basis. Similarly, a component might even offer different levels of checking for some conditions—fast, but less rigorous checks versus slow but tediously thorough checks, for example. An interface that provides a systematic way to query the wrapper for the checks it can provide as well as enable or disable them at designated levels would allow component composition environments to directly support such services in a uniform way.

4.4 Self-Testing Services

Component-based approaches to software construction highlight the need for detecting failures that arise as a result of miscommunication among components. In an invited paper at the 22nd International Conference on Software Engineering, Mary Jean Harrold laid out a roadmap for the future of software testing research and identified testing techniques for component-based systems as one of the fundamental research areas ripe for exploration [8]. Violation checking services address some aspects of testing-based component verification, but additional testing support can be critical in supporting an application develop in the process of verifying an application in combination with a component.

It is possible for a reflective wrapper to provide self-testing capabilities in addition to violation checking services. For example, a selection of component developer-provided test suites (from short and simple to long and thorough) could be embodied in the reflective wrapper. Self-testing can then be performed by executing a selected test suite on the wrapped component—perhaps while also enabling interface violation checking.

Such a testing approach provides a natural, incremental approach to application integration. If each component comes prepackaged with test data (and with violation checking services acting in the role of test oracle), a component's own self-test becomes an ideal "real world" test for the lower-level components on which it depends.

Such an approach could even be expanded to support the integration of client-written test suites into the self-testing scheme. Bruce Weide has also suggested that such a wrapper could potentially be augmented to provide operation call/parameter record and playback capabilities [27].

At this point, the benefit of intercessory services from the point of view of component specifications becomes clear. An appropriately structured reflective wrapper can provide for changes in its own behavior—it can allow one to enable or disable specific actions that occur immediately before or after it delegates calls to the wrapped implementation. Although this does not support intercessory actions on the underlying component, simply adding or removing certain actions before and after delegating to the

wrapped component supports many powerful capabilities oriented toward component composition and testing-based verification.

4.5 Abstract Value Manipulation

The component wrapping scheme previously proposed for interface violation checking [3] uses a novel approach to implementing checks before and after operations. Instead of implementing checks in terms of the concrete implementation values inside the underlying component (and thus violating encapsulation), the component is required to provide the computational equivalent of an abstraction function (or abstraction relation). Program-level classes that correspond to the various mathematical modeling types used in defining the state model and pre- and postconditions for the component are used to represent abstract (specification-level) values. The result is that the wrapper asks the component to "project" an abstract value of its current state as a separate object. All checking and analysis is then done on this object, which is designed to mimic the corresponding mathematical abstraction.

This abstraction relation approach can be co-opted for a reflective specification wrapper to provide additional intercessory capabilities. If a component were to provide both an abstract-relation-based projection function ("convert-to-abstract-model") and a corresponding injection function ("convert-from-abstract-model"), then it would be possible for a development environment or other tool to directly access and manipulate an abstract representation of the state of a component. Further, manipulations of that abstract representation could then be "pumped back down" into the component itself. This approach works naturally for components where the abstraction relation is a one-to-one mapping. For many-to-one or many-to-many mappings from representations to abstract values, practical convert-from-abstract-model injection functions are not always possible, and so such a feature cannot be required for all reflective wrappers.

While such an interface can be used for certain kinds of metaprogramming, in the context of component-based software, greater impact is likely to accrue from using such a capability within a development environment. All components would now have a standard interface for plugging into state visualization tools, for prototyping and testing use, and for interfacing with model-checking tools.

4.6 Documentation?

To come full circle, one can also consider incorporating program-level access to component documentation through a reflective interface, as opposed to providing specification information through traditional documentation. JavaDoc and other embedded documentation strategies push the documentation down into the source code in a way that allows tools to extract, format, package, and navigate it for human readability. In the same manner, one can imagine the specification object obtained from `GetSpecification()` providing component-level, per-operation-level, and per-parameter-level documentation strings in a form suitable for compilation into on-line documentation, use in a component property browser, or use in a documentation search database. As with current components, it is likely that printed documentation will be needed for some purposes, but providing program-accessible documentation through a standard interface may have unexplored benefits.

5. LIMITATIONS

There are a number of drawbacks to explore when considering the wrapper approach proposed here. One immediate concern is that this approach may lead to code bloat in the final application, since each component would now be accompanied by one or more supplementary classes to provide its wrapper, testing support, abstract value manipulation, and other services. The critical aspect of the wrapper approach is that all of these services are designed for use during *development*, when specification information about components is most valuable, rather than after delivery, when component specifications are of little or no use to the end user. Because these additional services wrap the underlying component during development, it is a simple matter to remove them for final release builds, without requiring access to the component's source code.

Another concern is whether or not this approach will demonstrably lead to better quality components. However, the position in this paper is that reflective wrappers are designed to provide a mechanism to deliver and later access a component's specification (particularly by development tools). Solving this problem is necessary to allow developers of component-based software to leverage the formal specifications created by component developers. One should not make the mistake of presuming that any solution to the specification packaging and distribution problem will, by itself, be sufficient to guarantee an increase in software quality.

A more significant concern is the question of how specifications will be communicated through the wrapper interface. A program-manipulable representation of specification features is necessary for this strategy to work. One possibility is to use the Extensible Markup Language (XML) [30] to represent specification information, which would require the development of one or more appropriate Document Type Definitions (DTDs). Such a representation must be generally acceptable in order for tools to support it. It is difficult to imagine that one representation could work for the myriad of specification approaches and notations available today. Instead, such a representation would most likely require difficult choices about the specification approach to be used.

Finally, it is clear that packaging and delivery of specifications using the wrapper approach will require additional work by component developers above and beyond simply creating the specifications. Generation of wrapper boilerplate and implementation of many wrapper services, including representing and accessing specification details, can be automated so that no additional work is required of component developers. However, fully supporting all of the features described in Section 4 will require some manual effort. The primary services that may require additional developer-supplied code are:

- Precondition, postcondition, and invariant checks for violation-checking services.
- Selection (and perhaps even construction) of test suites for self-testing services.
- Abstract model projection and injection functions to support abstract value manipulation.

To make this approach practical, it is clear that a "sliding scale" of wrapper functionality, where a given wrapper provides only some subset of the reflective services described in Section 4, is desirable. Component developers who wish to devote the resources necessary for implementing more comprehensive wrapper features

might then have an advantage in competing for more demanding customers.

6. RELATED WORK

The wrapper approach to providing access to specification information was initially inspired by a prior framework for run-time behavioral contract checking [3] and a larger strategy for end-to-end, automated, specification-based testing [2]. This prior research is also related to formal specification and to verification, as well as specification-based testing and parameterized programming. Because of the sweeping nature of the position advocated here, it is related to and has been influenced by a wide variety of existing work across a selection of topics in design, programming languages, formal specification, testing, and software reuse.

Reflection has a 20-year history in programming languages [24], and has been widely discussed at OOPSLA, at the Annual Workshops on Object-Oriented Reflection and Metalevel Architectures, and more recently at the International Conference on Meta-Level Architectures and Reflection. Kiczales has provided one of the most influential discussions of the subject in the context of CLOS [11]. Ferber described alternative approaches to supporting computational reflection in class-based OOPLs [5]. The proposal in this paper adds nothing new to the realm of reflection—instead it aims to take what has been learned about reflection in the design of object-oriented languages and reapply those insights to a new problem: packaging and providing access to specification information and related services. The primary difference in the approach proposed here is that many useful reflection capabilities can be provided within a framework that does not support reflection simply by using wrappers (although general computational reflection cannot, of course). Past work involving formal specification and reflection has primarily focused on reflection as a specification technique, or on how to specify reflective behavior [14, 23].

Within the reuse community, the issue of providing program-level access to specification features has received little attention. The question of how and what to describe in relation to a component's verification history, however, has been discussed widely under the topic of "component certification" [20, 4, 29, 21, 10]. Lessons from the reuse community provide much insight into what kinds of information may be useful to clients in this regard.

The interface violation checking approach described here [3] naturally meshes with Bertrand Meyer's view of design by contract [18]. A violation checking wrapper is intended to provide run-time checking of such contractual obligations while separating such checks from either of the parties involved. The value added by the wrapper approach results from separating the checking code from both the client and the base component and promoting it to a separately manageable class. This addresses concerns about clutter, expression of more complex conditions, and detracting from the focus of the underlying implementation, while allowing one to easily include or exclude checks on a per-component basis in a plug-and-play fashion.

Alternative approaches to run-time assertion checking include Eiffel [19], iContract [13], Rosenblum's Annotation Pre-Processor (APP) [22], and Kiczales' AspectJ [12]. Eiffel supports compiler-generated run-time checks based on user-provided Boolean assertions phrased in terms of publicly exported class features. iContract provides services similar to those of Eiffel, but for Java programs. APP allows separately defined checking operations to be compiled into or out of C code for assertion checking, but makes no distinction between values at the abstract, specification level and the concrete, implementation level. AspectJ allows one to create a separate aspect containing checking code and then choose whether or not to weave this cross-cutting decision into a non-checking implementation at build time. In many ways, AspectJ is philosophically closest to the approach advocated here.

The self-testing concepts folded into the reflective wrappers proposed here have been most heavily influenced by current research in automated, specification-based testing [2]. The idea of providing self-testing capabilities through a standard interface is orthogonal to the approach(es) used to generate test suites and the approach(es) used to assess correctness. As a result, virtually any testing approach could be integrated into the wrapper strategy.

7. SUMMARY

For clients to receive the benefits provided by formal specifications, those specifications must be distributed to clients along with the components they describe. The discussion presented here explores some of the issues surrounding the question of how best to package and deliver such specification information.

The position taken in this paper is that a component's specification (and other supporting information) should be provided *as another component* (or set of components), consisting of a reflective specification wrapper and associated specification objects. This position is based on three insights: reflection supports both human-oriented and tool-oriented access to specifications; a wrapper approach cleanly allows the addition of interface functionality and opens up powerful reflection capabilities, even when the underlying component technology does not support reflection; and the whole scheme can be implemented in a manner transparent to client code.

Given this position, potential introspective and intercessory services for reflective specification wrappers were explored. In addition to specification and verification information, reflective wrappers could potentially be used to provide services for violation checking, self-testing, and even abstract value manipulation. Even documentation could be accessed programmatically through a reflective wrapper. Although such reflective specification wrappers have not been implemented in the form proposed here, their potential deserves further investigation.

The position taken in this paper only outlines one possible mechanism for packaging and delivering specification information, however. While many of the services and capabilities suggested in this paper require significant research issues to be addressed before one can capitalize on the wrapper approach, the services are orthogonal enough that progress can be made incrementally. Nevertheless, the cornerstone of the approach involves capturing and representing "plain old specifications." First and foremost, this research issue must be solved for the position espoused here to be viable. A program-manipulable representation of specification features is necessary for this strategy to work—perhaps one based on XML. Such a representation must be generally acceptable in order for tools to support it. It is difficult to imagine that one representation could work for the myriad of specification approaches and notations available today. Instead,

such a representation would most likely require difficult choices about the specification approach to be used.

8. ACKNOWLEDGMENTS

Bruce Weide, Murali Sitaraman, and Joseph Hollingsworth have all contributed to the basic approach proposed in this paper; their contributions are greatly appreciated. In addition, we gratefully acknowledge financial support from Virginia Tech and from the National Science Foundation under grant CCR-0113181. Any opinions, conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of NSF or Virginia Tech.

9. REFERENCES

[1] Edwards, S.H. Black-box testing using flowgraphs: An experimental assessment of effectiveness and automation potential. *Software Testing, Verification and Reliability*, Dec. 2000; 10(4): 249-262.

[2] Edwards, S.H. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, June 2001; 11(2).

[3] Edwards, S., Shakir, G., Sitaraman, M., Weide, B.W., and Hollingsworth, J. A framework for detecting interface violations in component-based software. In *Proc. 5th Int'l Conf. Software Reuse*, IEEE CS Press: Los Alamitos, CA,1998, pp. 46-55.

[4] Edwards, S.H., and Weide, B.W. WISR8: 8th Annual Workshop on Software Reuse: Summary and working group reports. *ACM SIGSOFT Software Engineering Notes*, Sept./Oct. 1997; 22(5): 17-32.

[5] Ferber, J. Computational reflection in class based object-oriented languages. *ACM SIGPLAN Notices (Proc. OOPSLA'89)*, Oct. 1989; 24(10): 317-326.

[6] Foote, B., and Johnson, R.E. Reflective facilities in Smalltalk-80. *ACM SIGPLAN Notices (Proc. OOPSLA'89)*, Oct. 1989; 24(10): 327-335.

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

[8] M.J. Harrold. Testing: A road map. In *The Future of Software Engineering*, A. Finkelstein, ed., ACM Press, New York, NY, 2000, pp. 61–72.

[9] Hollingsworth, J.E., Blankenship, L., and Weide, B.W. Experience report: Using RESOLVE/C++ for commercial software. In *Proc. ACM SIGSOFT 8th Int'l Symposium on the Foundations of Software Engineering* (San Diego, CA, November 2000), ACM, pp. 11-19.

[10] IEEE. *Supplement to IEEE Standard for Information Technology—Software Reuse—Data Model for Reuse Library Interoperability: Asset Certification Framework*. IEEE Std 1420.1a-1996, Apr. 3, 1997.

[11] Kiczales, G., des Rivieres, J., Bobrow, D.G. *The Art of the Metaobject Protocol*. MIT Press, 1992.

[12] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W.G. Aspect-oriented programming with AspectJ. Available on-line at http://www.aspectj.org.

[13] Kramer, R. iContract—the Java design by contract tool. In *Proc. Technology of Object-Oriented Languages, TOOLS 26*, IEEE CS Press, 1998, pp. 295-307.

[14] Kurihara, M. and Ohuchi, A. An algebraic specification of a reflective language. In *Proc. 15th Annual Int'l Computer Software and Applications Conf., COMPSAC '91*, IEEE CS Press, 1991, pp. 231-236.

[15] Latour, L., Wheeler, T., and Frakes, B. Descriptive and predictive aspects of the 3Cs model, SETA1 working group summary. *Ada Letters (Proc. 1st Symp. Environments and Tools for Ada)*, Spring 1991; 11(3): 9-17.

[16] Lee, A.H. and Zachary, J.L. Reflections on metaprogramming. *IEEE Trans. Software Engineering*, Nov. 1995; 21(11): 883-893.

[17] Maes, P. Concepts and experiments in computational reflection. *ACM SIGPLAN Notices (Proc. OOPSLA '87)*, Dec. 1987; 22(12): 147-155.

[18] Meyer, B. Applying "design by contract." *Computer*, Oct. 1992; 25(10): 40-51.

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

[20] Poulin, J., and Tracz, W. WISR'93: 6th Annual Workshop on Software Reuse: Summary and working group reports. *ACM SIGSOFT Software Engineering Notes*, Jan./Feb. 1994; 19(1).

[21] Rohde, S.L., Dyson, K.A., Geriner, P.T., and Cerino, D.A. Certification of reusable software components: Summary of work in progress. In *Proc. 2nd IEEE Int'l Con. Engineering of Complex Computer Systems*. IEEE CS Press, 1996, pp.120-123.

[22] Rosenblum, D.S. A practical approach to programming with assertions. *IEEE Trans. Software Eng.,* Jan. 1995; 21(1): 19-31.

[23] Saeki, M., Hiroi, T., and Ugai, T. Reflective specification: Applying a reflective language to formal specification. In *Proc. 7th Int'l Workshop on Software Specification and Design*, IEEE CS Press, 1993, pp. 204-213.

[24] Smith, B. Reflection and semantics in a procedural language. Technical Report 272, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, 1982.

[25] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[26] Weide, B.W., Heym, W.D., and Hollingsworth, J.E. Reverse engineering of legacy code exposed. In *Proc. 17th Int'l Conf. Software Engineering*, ACM, Seattle, WA, Apr. 1995, pp. 327-331.

[27] Weide, B.W., Heym, W.D., and Ogden, W.F. "Modular regression testing": Connections to component-based software. In *Proc. 9th Annual Workshop on Software Reuse*, Jan., 1999.

[28] Wing, J.M. A specifier's introduction to formal methods. *IEEE Computer*, Sept. 1990; 29(9): 8-24.

[29] Wohlin, C., and Runeson, P. Certification of software components. *IEEE Trans. Software Engineering*, June 1994; 20(6): 494-499.

[30] www.xml.org.