Techniques for Embedding Executable Specifications in Software Component Interfaces

Ross McKegney[1] and Terry Shepard[2]

[1] Queen's University & IBM Canada, Toronto, Ontario Canada
mckegney@cs.queensu.ca
[2] Royal Military College of Canada, Kingston, Ontario Canada
shepard@rmc.ca

Abstract. In this paper, we consider interface contracts as a possible mechanism for improving semantic integrity in component-based systems. A contract is essentially a formal specification interleaved with code and allowing a component or object to unambiguously specify its behaviour. The existing techniques that we survey are predominantly designed for object-oriented systems; we therefore investigate the extent to which they can be scaled up to the level of components, and embedded in interface specifications rather than code. We conclude that interleaved specifications are viable and useful at the level of components, but that future work is required to develop languages that can express the constraints that are important at this level of granularity.

1 Introduction

Software components are typically used/reused based on their syntactic interface descriptions and informal documentation of behaviour made available either embedded as comments or in a separate document. We view composition on this basis as being inherently problematic, because the informal descriptions are likely ambiguous and there is no mechanism to ensure that the implementation and its description remain consistent. If we formalize the external documentation then we alleviate the first problem, but formal specification is difficult and expensive [13]. Instead, we view as most promising those techniques that support interleaving formal specifications with source code, where the notation is a slight extension to the language(s) of development. It should be feasible to apply such techniques to component interface descriptions, allowing for the verification of conformance between implementation and specification using static or dynamic analysis.

This paper surveys techniques for embedding formal specifications in software component interfaces. We discuss two classes of techniques: Design by Contract based approaches, and Behavioural Interface Specification Languages. Both provide means for interleaving an executable specification of the abstract behaviour of a component with its syntactic interface description.

2 Components and Composition

We define the term software component as an executable unit of deployment and composition. The implication is that systems are built by composing components built by different groups of developers, based on interface descriptions and external documentation. Component technologies, including Corba, Enterprise JavaBeans^{TM}, and COM greatly simplify the composition process, by providing generic services and enforcing common communication protocols. Unfortunately, although component technologies have made the composition process much easier, they are limited to enforcement of the syntactic aspects of the interactions, and do little to aid with resolving semantic inconsistencies.

In order to ensure that components use each other in the way(s) in which they were intended (what is sometimes referred to as semantic integrity [3]) we need some mechanism to unambiguously specify component behaviour. Essentially, this amounts to understanding the assumptions made by the component developer and providing a mechanism whereby these assumptions can be conveyed to clients wishing to use the component. The range of assumptions that may be of interest is very broad, and would be a worthwhile research area in its own right, but we can state some of the common ones: type and range of parameters and return values; exception semantics; types and versions of dependencies; valid protocols of interaction; effects of operations on internal state; performance assumptions; concurrency assumptions; valid customization/parameterization; control assumptions; security assumptions; data structure semantics. These assumptions are all relevant to the correctness of the component; that is to say that if a client violates one of these assumptions then we cannot guarantee that the component will behave correctly.

Most software processes begin with some form of requirements document, which forms the basis for some design document, which is realized with some implementation, and tested against the requirements. This approach can be applied using informal notations (e.g. UML [4]) or formal notations (e.g. Z [31]). If formal notations are used, then we can sometimes apply a refinement calculus to the specifications as they are iteratively refined to code, verifying at each step that the properties which have been proved of the specification are not violated. The downside is that formal specification, proof, and refinement are all complex activities requiring significant training, knowledge, and experience. There is also the issue of the semantic gap between specification language and language of implementation, it is often non-trivial to map an implementation back to its specification. Finally, the formal specification is yet another work product that must be maintained throughout the development lifecycle, if benefits beyond simply aiding in reasoning about the system's design are to be realized.

An alternative approach, also relying on external formal specification of behaviour, is to use an executable specification language and run the model concurrently with the implementation to verify conformance. This is the approach taken by Microsoft for runtime verification using the Abstract State Machine Language [1] and by Bastide et. al. [2] for verifying a model of the Corba Event Service specified using Petri-Nets. As with the approaches that we will discuss in this paper, these both provide mechanisms for unambiguously specifying component behaviour, and for verifying conformance between specification and implementation.

We have selected to further investigate techniques for behavioural specification of components by interleaving executable formal specifications with source code. Ideally, such techniques would also work in conjunction with standard component technologies.

3 Design by Contract

The first techniques that we will discuss are based on Design by Contract^{TM}, as introduced by Meyer in [26], and implemented in the Eiffel programming language [27]. Design by Contract can be viewed as a merger of abstract data type theory and Hoare logic [15]. Classes are specified using contracts interleaved with source code, consisting of internal consistency conditions that the class will maintain (invariants) and, for each operation, the correctness conditions that are the responsibility of the client (pre-conditions) and those which the operation promises to establish on return (post-conditions). Because these conditions are to be expressed in the language of development, and interleaved with source code, they can be mechanically checked at runtime.

Design by Contract is typically implemented in terms of assertions, but it provides far more than this. Contracts represent a specification of the system that, through tool support, can be extracted and viewed as a separate artifact. The differentiation between pre-conditions, post-conditions and invariants allows for selective checking at runtime, based on the development phase. Specifications in terms of contracts can be used as the basis for unit test generation. Finally, contracts can be inherited from parent to child class, and refined.

In the remainder of this section, we further elaborate the Design by Contract programming style as implemented in Eiffel, followed by brief summaries of Design by Contract implementations for other programming languages.

3.1 Eiffel

Systems in Eiffel are modeled as a set of 'clusters', logical components (there is no physical construct for clusters in Eiffel, they are implemented as a group of related classes) with their specifications interleaved with the code. These clusters can be sequentially or concurrently engineered, using a micro-process of analysis, design, implementation, verification & validation, and generalization (this final step is similar to refactoring, and involves evaluating and preparing the cluster for general reuse). Throughout the process, all work is performed on a single product—the cluster source code—producing at the end a self-describing software component. Maintaining a high level of reliability from the outset is considered a core principle (above functionality); error handling and assertions are put in first, followed by the implementation of the methods.

Eiffel is a fully object-oriented language, with several interesting features that we will not discuss here (for a good introduction to the language itself, see [25]). What we will elaborate further is the implementation of Design by Contract as a core feature. We mentioned above that in Eiffel there is only one work product—the source code—and that Eiffel supports features for expressing the other work products that we would normally expect (requirements documents, design documents, etc.). The approach taken by Eiffel is to express this information in terms of contracts between collaborating components, unambiguously specifying each party's expectations and guarantees in terms of pre-conditions, post-conditions, and invariants on the services provided. Contracts permit the specification of semantic aspects of an object or component's functionality beyond what can be achieved through method signatures alone. Additionally, the executable nature of contracts assists in testing, since errors in the code may cause localizable contract violations rather than a failure later on. From a development perspective, contracts mean that servers do not have to check the values of inputs as they are received from clients (assuming that the server has properly specified its pre-conditions), and likewise the clients do not have to check the output values returned (assuming that the server has properly specified its post-conditions). The implication is that the specification defines what values are valid, and the underlying implementation can assume that it will always be working on valid data. If an error does occur, then it means either that there is a problem with the specification or that one of the components/objects has violated its contract. The downside of the approach is that the extra code for contract checking may slow down the system; as a result, Eiffel supports a variety of compiler options for the amount of contract code to generate. It is recommended that contracts be turned on to the highest level (all assertions) during development and testing, then switching to a lower level (no assertions or preconditions only) when the system is released.

In Eiffel, child classes automatically inherit not only the attributes and methods of the parent class—but also the parent's contracts. Just as Eiffel supports refining or extending the parent's class functionality, so does it support extending the contracts, using the require else (for weakening the original pre-condition) and ensure then (for strengthening the original post-conditions) clauses. As the level of nesting increases, the contracts can become complex—because they are spread across many classes. The solution provided by the Eiffel environment is to allow the viewing of a flat form and a flat contract form of a class. The flat form of a class is simply a view of the class augmented with everything inherited from its ancestors. The flat contract form is a contract version of this class. As with the contract form described earlier, these techniques allow various views of the system.

Despite some nice features that greatly aid in ensuring semantic integrity between components, the Eiffel programming language is unfortunately not widely used. It is object-oriented with support for distributed object computing, just like other more popular programming languages; its relative obscurity can possibly be attributed to relatively poor performance, and lack of support for the standard distributed object platforms.

3.2 DbC for Java^{TM}

iContract [19] is a prototype tool developed by Reliable Systems Inc. for embedding Eiffel-style contracts in Java code. Developers specify contracts using javadoc-style comment tags (@pre for pre-conditions, @post for post-conditions, and @invariant for invariants) on methods of classes or interfaces, and elaborate the contracts using an executable subset of the Object Constraint Language (OCL) [32]. The iContract tool is implemented as a source code preprocessor that instruments the code based on the contracts specified, creating assertions that can be checked at runtime.

Karaorman's jContractor [18] uses the reflective capabilities of Java and the Factory Method design pattern [14] to support the addition of contracts. When objects are instantiated through a special factory class, the factory adds to the new instance the instrumentation code. This approach uses only existing, standard Java constructs and calls. It also provides for retrying a method that throws an exception, and rolling back variables to earlier values. However, the user must specify as distinct private Boolean methods each precondition, postcondition, invariant, and exception handler, leading to a great proliferation of class members.

Biscotti [6], targets the Java Remote Method Invocation (RMI) distributed object infrastructure, and is implemented as an extension of the Java language and modifications to the Java compiler and Java Virtual Machine (JVM). Biscotti introduces six new Java keywords: invariant, requires, ensures, then, old, and result. These keywords can be used to describe pre-conditions and post-conditions on methods, and class and instance invariants on classes and interfaces. Biscotti also introduces assertion reflection, an extension to the standard reflection capabilities of Java so that contract information can be queried. Through the methods getInvariant(), getPrecondition(), and getPostcondition(), it is possible to find out the instance invariants, pre-conditions and post-conditions before making a method call on a distributed object. This is particularly useful when the intention is to leave contracts turned on even in deployed systems. The authors of Biscotti argue that it is unrealistic to rely on well-behaved clients in a distributed object system—which is justification not only for adding assertion reflection, but also for the general approach of requiring a proprietary compiler and JVM to compile and run programs with Biscotti contracts embedded. The downside of this design decision is that components written using Biscotti extensions can only be compiled and executed using the proprietary Biscotti compiler and JVM, and that there is currently no supported mechanism for turning the contracts off—which is often desirable once the component has been thoroughly tested.

3.3 DbC for Other Programming Languages

There have been a variety of research and industrial projects to bring the concepts of design by contract to other, more widely used languages. These include implementations for Ada (e.g. ANNotated Ada [23]), C++ (e.g. A++ [9,10]), Python [28], SmallTalk [5], and Lisp [16].

4 Behavioural Interface Specification Languages

Similar in spirit to the design by contract methodology is the work on behavioural interface specification languages (BISLs), proposed by Wing [33] as a mechanism for assigning formal behaviour models to interfaces. BISLs consist of two tiers, a mathematical tier where 'traits' (e.g. mathematical models of Sets, Bags, etc.) are defined, and an upper tier implemented as an extension to the language of development. These languages do not appear to be widely used in industry, but they are the subject of considerable academic research.

BISLs go beyond simply allowing the specification of valid and invalid behaviour, they also allow for the specification of exceptional behaviour. That is, the developer can specify under which conditions a method will terminate normally, which conditions the method will terminate with a particular exception, and under which conditions no guarantees are made on the implementation. BISLs also typically include the notion of a frame condition, stating which of the model variables the methods can update.

In this section we discuss two BISLs proposed for adding semantics to OMG IDL (Larch/Corba and IDL++), followed by a discussion of the Java Modeling Language, a BISL for Java.

4.1 Larch/Corba

Larch [12] is a formal specification language whose distinguishing feature is that it uses two "tiers" (or layers) [33]. The top tier is a behavioral interface specification language (BISL), used to specify the behaviour of software modules using pre-/post-conditions, and tailored to a specific programming language. The bottom tier is the Larch Shared Language (LSL), which is used to describe the mathematical vocabulary used in the pre- and post-condition specifications. Therefore, LSL specifies a domain model and the mathematical vocabulary, while the BISL specifies the interface and behaviour of program modules. BISLs have been written for a variety of programming languages, including Larch/CLU, Larch/Ada, LCL for ANSI-C, LM3 for Modula-3, Larch/Smalltalk, Larch/C++, Larch/ML, and VSPEC for VHDL (see [20] for pointers to the latest list of implementations). Each BISL uses the declaration syntax of the programming language for which it is tailored, and adds annotations to specify behavior. These annotations typically consist of pre- and post-conditions (typically using the keywords requires and ensures), some way to state a frame axiom (typically using the keyword modifies), and some way to indicate what LSL traits are used to provide the mathematical vocabulary (a trait in Larch is simply a theorem that can be checked with a prover, e.g. a trait used in post-condition might be that result > 0).

In his doctoral dissertation [30], Sivraprasad proposed Larch/Corba, a BISL for OMG IDL. A prototype implementation was produced, allowing for the specification of pre-/post-conditions, invariants, and concurrency in IDL interfaces. Larch/Corba is interesting for our evaluation because it combines a widely used interface definition language with formal specification techniques. Although we found this technique much more complex than the design by contract languages, it is also much more precise than some of the others—particularly because of the ability to specify concurrency and exception conditions.

4.2 IDL++

D'Souza & Wills propose the framework for a behavioural specification extension to OMG IDL in [10]. Their approach is similar in spirit to Larch/Corba; although their emphasis is on the ability to integrate the semantic specifications of objects with OOA/D tools. Their proposal comes as a response to a request for information (RFI) made by the OMG on adding semantics to IDL, and suggests making the behavioural specification of component interfaces as part of a larger process. Design by contract has similar ambitions, with the contracts being a work product that takes the place of design documentation and that aids in every phase of development, but the IDL++ approach takes this a step further by suggesting a customizable formal infrastructure on which components can be built.

As in Larch, semantics are divided into multiple tiers (or layers), but this time instead of two layers there are four. The lowest layer in the IDL++ model is called the 'Traits and Theories' layer, which is analogous to the Larch LSL tier and provides the mathematical foundation on which the remaining layers rest. This layer is given in terms of traits and theories, where a trait is a parameterizable specification fragment (e.g. result > 0) while a theory is a collection of related traits (e.g. the set of traits required to describe the commit/rollback behaviours of a transaction scheme). The second layer is called the 'Fundamental Object Model', and provides mechanisms for specifying objects and their design. It consists of formal models of objects and actions, where an object is a theory giving the generic behaviour of objects, including a list of queries that can be performed on the object and a mapping from an object identity to a state, while an action is simply a message or sequence of messages that causes some change to the object. In this sense, queries abstract away the internal structure of a design while actions abstract away the details of the dialogue.

The top two layers are the ones that developers will use, and include the 'Methodology Constructs' layer and the 'End-User Models' layer. The methodology constructs layer is important in this design because of the emphasis on integration with OOA/D and CASE tools. It should be possible to work on a design using diagrams that have a formal basis allowing for translation to lower-level constructs (e.g. code) for further refinement. The end-user models represent the top layer of the semantics model; this is where analysts and designers will apply the models formalized in the lower layers to the classes and interactions in their designs. Unlike other approaches to semantic extensions to interfaces, IDL++ allows for the specification of both incoming and outgoing invocations for each interface that an object supports, and even for the specification of protocols of interaction. Using these techniques, developers can formally specify patterns or frameworks, which can be reused throughout the system.

Using the proposed approach, developers would model their systems using graphical tools, and the IDL++ interfaces would be generated automatically. These specifications would be machine processable so that design tools would be able to search for, interrogate, select, and appropriately couple components at run time based upon their specifications. Although this is an interesting proposal, and successfully merges ideas from software process and formal specification, it has not been implemented to date.

4.3 JML

The Java Modeling Language (JML) [22] is a behavioural interface specification language (BISL) [33] developed as a collaborative effort between Compaq Research, the University of Nijmegen, and Iowa State University. Self-described as being more expressive than Eiffel, and easier to use than other BISLs such as Larch [12] or VDM [17]; JML uses a slight extension of the Java syntax to allow developers to embed specifications in Java classes and interfaces. The suite of tools surrounding JML is constantly evolving, and currently consists of tools for static analysis, verification, automated document generation, and run-time debugging (see JML website [21] for current tool list).

Like Larch, JML uses a two-tiered approach to specification, a lower level specifying the mathematical notation to be used in specifications and a domain model, and an upper level used to specify interfaces to components in terms of pre-/post-conditions and invariants. A fundamental difference, however, is that both layers are described in terms of the language of implementation (Java).

We found JML to be a much richer language for specifying contracts than the design by contract languages. In particular, the ability to specify not only constraints on valid behaviour, but also constraints on exceptional behaviour was very useful. As with the other BISLs, the ability to describe whether a method call affects the underlying model, and if so what changes are made, is tremendously useful when trying to determine valid ways of using the component. Although we found this technique very useful, we are concerned about issues of scale, as the complexity of the abstract representation of the component becomes greater.

5 Discussion and Opportunities

5.1 Contracts, the Ideal Formal Method?

The first test of the techniques investigated in this paper will be to compare them against the 10 ideal criteria set forth by Clarke and Wing in [7] for practical formal methods.

The first ideal is that developers should get ‘early payback’; they should see benefits of using the approach soon after beginning to use the new technology. Techniques for interleaving formal specifications with code achieve this objective in two respects: first, the process of formal specification itself is a worthwhile endeavor; second, by using these techniques the developer is relieved from writing code within the classes to check that input is valid. The second ideal is "incremental gain for incremental effort"; as developers become increasingly proficient with the tool, they should find increasing value. This is certainly true from the perspective that someone with experience writing formal specifications will be able to write them in a more elegant fashion than a novice. However, there are limits to what can be expressed through the techniques discussed in this paper, and complex conditions on concurrency, for example, cannot be expressed. The third ideal states that the method should support "multiple uses"; it should aid in multiple phases of the development cycle. This is one of the greatest benefits of DbC and BISLs, they not only aid in specifying the requirements of a system, they also aid in development (by formally specifying the components that are being assembled), unit testing (through tools that can automatically generate test cases from specifications), integration testing (by turning on contracts and verifying that none of the interactions in the system violate the contracts), maintenance (specifications are embedded in the code, and as such are less likely to become outdated), and reuse (components can be reused on the basis of their specifications rather than their implementations).

The fourth ideal is "integrated use"; methods and tools should work together and with common languages/techniques, with little requirement to buy into new technologies. DbC and BISL implementations exist for a variety of programming languages, meaning that it should be possible to find one to fit with the technologies being used. On the other hand, none of the tools integrate well with any common development environment, and few are suitable for production grade development. These technologies also carry with them a particular methodology, which must be bought into. The fifth ideal is "ease of use". Because these techniques are all implemented as slight extensions to the language of development, the learning curve to their effective use is minimal. However, tool support for most of these techniques is often poor and can be difficult to install and use. The sixth ideal is "efficiency"; tools should make efficient use of developers' time, turnaround time with an interactive tool is acceptable. As currently implemented, the tools are primarily run in batch mode and are not significantly less efficient compiling or executing test suites. If we want to build static checking capabilities on top of these notations that may change, but as currently implemented there is no efficiency penalty in this sense.

The seventh ideal is "ease of learning." With the exception of Larch/Corba, all of these techniques were easy to learn. The Larch Shared Language that underlies Larch/Corba and all of the other Larch BISLs is complex and requires training. The eighth ideal is "error detection oriented"; methods/tools oriented toward finding errors rather than proving correctness. This is precisely what these techniques are good for, aiding in finding bugs in programs before they manifest themselves as faults. The ninth ideal is "focused analysis"; methods/tools good at analyzing at least one aspect of a system well. This is at once a strength and a weakness of these approaches. These techniques are good at describing functional attributes of a system, but cannot easily be extended to describe other attributes of interest. The last ideal is "evolutionary development";

allow partial specifications that can be refined/elaborated. The only technique that explicitly supported refinement was JML. All of the techniques support partial description of systems.

5.2 Contracts for Components?

Contract-based design is generally perceived as being good practice, and is even a fundamental part of some object-oriented [26] and component-based development processes [11]. Nonetheless, given its purpose as a technique for improving semantic integrity between objects, it is worth discussing whether it can realistically be applied to components—which are not only at a different level of abstraction, but also possess some significant differentiating characteristics.

Unfortunately, although software contracts can be used to facilitate interface conformance checking (both on the part of the supplier of a service and the client, at compile-time and at runtime), alone they do not have the power and flexibility to resolve semantic integrity. Component-based systems are typically heterogeneous and distributed, meaning that a runtime contract evaluation tool will probably not be able to check all of the components. Further, since the goal in many component-based systems is to reuse, outsource, or purchase components off-the-shelf, it is unrealistic to expect that all of the interfaces in the system will be described in terms of the same contract technology (if at all). A potential solution is to use the wrapper design pattern [14] to wrap all components with contract-extended interfaces. The tradeoff to be considered in this case is homogeneity and the benefits of contracts vs. performance. An approach such as this would be particularly useful if the contract-extended interfaces included a reflection capability, such as is supported by Biscotti, which would enable components to inquire about the contracts of other components at runtime. Particularly if the system is composed dynamically, keeping the contracts active even for a final deployment may be an effective strategy.

Next is the issue of component granularity, which can vary significantly within and across applications. Contracts rely on being able to model the abstract type of a component or object, which may become infeasible as components increase in size and complexity. In the worst case, contracts may become unfeasible for everything except range and possibly type checking of parameters and return values.

The next issue is that components typically exhibit more complex interaction mechanisms than objects. Whereas it is reasonable to restrict the specification of protocols between objects to method calls, components often employ mechanisms such as event broadcasting and callbacks—which cannot be modeled succinctly using existing contract languages.

A final issue that must be reiterated is that techniques for embedding contracts in component interfaces typically do not consider non-functional attributes such as performance or security. Given that we are seeking to analyze component-based systems to ensure that assumptions made by a component's developers are not violated by other components, and that these assumptions may include these non-functional attributes, it is clear that contracts alone are insufficient.

The Real-time Corba specification [29] is the only mechanism we have found for quality of service contracts, allowing for the propagation of process priorities across a system and the specification of resource requirements, but this technique unfortunately makes assumptions about the configuration that are not realistic for most systems (e.g. that every host will be running a real-time Corba orb on a fixed priority real-time operating system). On the other hand, because contracts do allow us to express safety constraints, we can use them to check if two components are functionally equivalent—and if so select one or the other based on some non-functional quality attribute.

One of the other interesting properties of components is that they are often designed with some capability for customization and parameterization. We see the behavioural subtyping techniques as a promising mechanism for defining the inherent behaviour of a component, and for evaluating a given customized/parameterized version of the component for conformance. The one caveat is that we are not convinced that behavioural subtyping techniques will scale to large granularity components, for the same reason that design by contract techniques may not scale as the abstract type of the object becomes larger and more complex.

Our position at this stage is that component interface-level contracts are an important starting point for ensuring semantic integrity, but they must be either used in conjunction with other techniques (such as formal specifications or architectural description languages), or extended to support the types of constraints that arise when composing software from components.

5.3 Where Do We Go from Here?

We believe that the relationship between the contract techniques discussed here and some of the established techniques of software architecture description and formal methods could be very interesting. For example, to describe a system in terms of a formal specification or architecture description language then generate component interfaces with embedded semantics, then to implement components for the specification and use constraint checking to ensure that the implementation and interface are consistent. Inversely, given a component contract language that expresses the constraints that we want to express, we could use these as a basis for generating a formal specification that can be validated statically.

The most obvious area where research is needed is in fact the generation of such a contract language for components. It is unclear at this point whether such a language is feasible, given the issues of scale that arise even with the limited techniques currently available. However, we believe that it would be a worthwhile exercise to attempt such an extension to one of the BISLs (since they are much more expressive than the design by contract techniques). Issues that could be addressed include quality of service attributes or valid protocols of interaction. Another area of research that could prove quite fruitful would be to look at domain specific contracts. For example, to determine if there are particular types of constraints that are important to document and mechanically check in certain domains. One possible area to look at is the security aspect of components composed in e-Commerce systems.

In terms of tool development, we foresee many opportunities. For example, a tool that can generate a contract by poking and prodding a component through its interface(s) would be very useful. Another area where tools are useful is for contract repositories. Given a set of reusable components with specifications interleaved with their interface definitions, we could use such a tool to aid in component selection and integration based on requirements. A third area where tools may be useful is for the development of an environment supporting run-time composition of components, including contract negotiation at each binding. Technologies that should be investigated in this case would include web services, where we may wish to aggregate many services together, and there may be multiple choices for each of the constituent services.

There is also work to be done to investigate particular issues for some of the existing languages, such as scalability issues in BISLs or performance issues in DbC languages (particularly if we expect to leave contracts turned on after deployment). It would also be interesting to investigate whether any of these techniques can be applied hierarchically, i.e. with an overall contract description for the component, and another set of contracts on the underlying objects.

Of course the basis for all of this is that informal documentation in the form of comments is ambiguous. It would be interesting to study the extent to which this is true. This is perhaps more of a psychology experiment than computer science, but it would certainly be interesting to examine a wide range of existing components and determine the extent to which their interfaces are conducive to reuse.

We believe that syntactic component interface descriptions are insufficient, and that interleaved formal specifications are a practical and feasible solution. Although current techniques do not appear to be sufficiently expressive for the types of constraints that we care about when composing systems from components, we strongly feel that there is potential for valuable and interesting research in this area.

References

1. Barnett, M., and Schulte, W.: Spying on Components: A Runtime Verification Technique. In: Workshop on Specification and Verification of Component-based Systems, OOPSLA 2001, Technical Report #01-09a, Iowa State, October (2001) 7–13
2. Bastide, R., Sy, O., Palanque, P., and Navarre, D.: Formal Specification of CORBA Services: Experience and Lessons Learned. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (2000)
3. Blom, M.: Semantic Integrity in Programming Industry. Master's Thesis, Computer Science, Karlstad University (1997)
4. Booch, G., Rumbaugh, J., and Jacobson, I.: The Unified Modeling Language User Guide. Addison Wesley (1999)

5. Carillo-Castellon, M., Garcia-Molina, J., Pimentel, E., and Repiso, I.: Design by Contract in Smalltalk. In: Journal of Object Oriented Programming, November/December (1996) 23–28
6. Cicalese, C. D. T., and Rotenstreich, S.: Behavioral Specification of Distributed Software Component Interfaces. In: IEEE Computer, July (1999) 46–53
7. Clarke, E. M., and Wing, J. M.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys 28(4) (1996) 626–643
8. Cline, M. P., and Lea, D.: Using Annotated C++. In: C++ At Work 1990, September (1990)
9. Cline, M. P., and Lea, D.: The Behavior of C++ Classes. In: Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications, September (1990)
10. D'Souza, D. F., and Wills, A. C.: OOA/D and Corba/IDL: A Common Base. ICON Computing (1995) Available at: http://www.iconcomp.com.
11. D'Souza, D. F., and Wills, A. C.: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley, USA (1999)
12. Guttag, J. V., Horning, J. J., and Wing, J. M.: The Larch Family of Specification Languages. In: IEEE Software 2(5), September (1985) 24–36
13. Finney, K.: Mathematical Notation in Formal Specification: Too Difficult for the Masses? In: IEEE Transactions on Software Engineering 22(2), February (1996) 158–159
14. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
15. Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. In: Communications of the ACM 12(10), October (1969)
16. Hölzl, M.: Design by Contract for Lisp. Available at: http://www.pst.informatik.unimuenchen.de/personen/hoelzl/ tools/dbc/ dbc-intro.html
17. Jones, C. B.: Systematic Software Development Using VDM. In: International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J. (1990)
18. Karaorman, M., Hölzle, U., and Bruno, J.: jContractor: Reflective Java Library to Support Design-by-Contract. Technical Report TRCS98-31, University of California, Santa Barbara (1998)
19. Kramer, R.: iContract-the Java Design by Contract Tool. In: Proceedings of TOOLS 98 (1998)
20. Leavens, G. T.: Larch FAQ. Available at: http://www.cs.iastate.edu/~leavens/larch-faq.html
21. Leavens, G. T.: The Java Modeling Language Home Page. Available at: http://www.cs.iastate.edu/~leavens/JML.html
22. Leavens, G. T., Baker, A. L., and Ruby, C.: JML: A Notation for Detailed Design. In: Behavioral Specifications for Businesses and Systems, Kluwer (1999) 175–188
23. Luckham, D. C., von Henke, F. W., Krieg-Brueckner, B., and Owe, O.: Anna, a Language for Annotating Ada Programs: Preliminary Reference Manual. Technical Report CSL-TR-84-261, Stanford University, July (1984)
24. McIlroy, M. D.: Mass-Produced Software Components. In: Proceedings of the 1968 NATO Conference on Software Engineering, Garmisch, Germany (1969) 138–155
25. Meyer, B.: An Eiffel Tutorial. Available at: http://citeseer.nj.nec.com/meyer01eiffel.html
26. Meyer, B.: Object-Oriented Software Construction, 1st Edition, Prentice-Hall (1988)

27. Meyer, B.: Eiffel: the Language. Prentice-Hall (1992)
28. Plösch, R.: Design by Contract for Python. In: Proceedings of the Asia Pacific Software Engineering Conference (1997)
29. Schmidt, D. , and Kuhns, F.: An Overview of the Real-time Corba Specification. IEEE Computer, June (2000)
30. Sivaprasad, G. S.: Larch/CORBA: Specifying the Behavior of CORBA-IDL Interfaces. TR #95-27a, Department of Computer Science, Iowa State University, November (1995)
31. Spivey, J. M.: The Z Notation: a Reference Manual. Prentice Hall International Series in Computer Science, 2nd Edition (1992)
32. Warmer, J., and Kleppe, A.: The Object Constraint Language. Addison-Wesley (1999)
33. Wing, J. M.: Writing Larch Interface Language Specifications. ACM Transactions on Programming Languages and Systems 9(1), January (1987) 1–24