Synthesizing Interface Stubs for Reusable Classes

Satish R. Thatté

Department of Mathematics and Computer Science
Clarkson University, Potsdam, NY 13699-5815
Tel: (315) 268-2395
E-mail: satish@sun.mcs.clarkson.edu
Fax: (315) 268-6670

Abstract

The problem of fitting together reusable components in spite of differences in protocol and representation choices is well-known. I discuss a novel approach to the protocol problem based on a formal logic of adaptability of interface types to *automatically* synthesize interface stubs to achieve an exact fit between the available "socket" for a reusable part and the actual part. The underlying notion of *adaptability* turns out to be related to but distinct from both subtyping and inheritance. The important related problem of data and object mobility in the presence of representation differences such as those between polar and cartesian representations of points appears to be amenable to a solution based on transformational knowledge in the form of equational laws used in conjunction with unification/matching techniques.

Keywords: class libraries, interface adaptation, object mobility, stub generation

Workshop Goals: Learning; networking; advance state of the art of class libraries and object interoperability.

Working Groups: reuse and OO methods, reuse and formal methods, tools and environments, reuse education.

1 Background

I have started to get involved in thinking and learning about reuse issues in the context of object-oriented systems over the last year or so. My background is in programming languages and type theory and I have recently been working on some type-theoretic problems that have applications in automatic stub generation for adaptation of object interfaces including data transformations based on knowledge expressed in the form of equational theories. I am particularly interested in exploring the methods involved in enhancing interoperability of classes and objects at both small and large grained levels in environments that are heterogeneous in some sense (ranging from hardware platforms to "ontology").

2 Position

This paper starts from the position that object technology is the most promising vehicle for realizing the goal of large-scale software reuse. This is now a common perception that provides much of the motivation for the current industry-wide shift towards object-oriented software design methods. An obvious potential bottleneck in the practical realization of this vision is the weakness of current component integration tools especially in the context of heterogeneous systems, where heterogeneity is broadly understood to mean anything ranging from different languages/hardware platforms to different sources for code components. This problem has been recognized and partly addressed, *e.g.*, in the CORBA [1] standard for distributed components[1], and in IBM's SOM [2] library management system for class libraries. However, neither CORBA nor SOM adequately deal with the inflexibility of the interfaces (signatures) offered by objects to the outside world. I believe that such inflexibility is at least a serious nuisance and could become a real problem whether the objects are connected statically by being compiled together or are dynamically connected using, say, CORBA's notion of a dynamic invocation interface (DII).

Focusing on the static (class library) case first, note that in the typical lifecycle of a large component-based system, there are likely to be many rebuilding episodes in which some of the old components are replaced by functionally equivalent (or superior) new components. For instance, in replacing an old library of container classes with a better one, or in accommodating an upgrade for a database system which also results in a rationalization of the API. In both cases, there is likely to be a protocol mismatch between the expected and actual interfaces for the new objects. The application programmer is now faced with the task of somehow adapting the new objects to bridge the difference between their expected and available interfaces—the obvious solution is to create stubs to connect the old interfaces to the new ones[2]. The task is not difficult in principle; in fact in most practical cases it is conceptually trivial. However, if a number of interdependent classes are involved, along with the complications of genericity, inheritance and overloading, the technical difficulty and opportunity for error are likely to be substantial. Conceptual triviality and technical complexity is a very uninviting combination for most people, and as such the task of synthesizing interface stubs is likely to be unwelcome and an obstacle to the optimal evolution of both class libraries and component-based systems. At the same time, its relatively mechanical nature makes it an inviting target for automatic inference.

[1] Although the main motivation for CORBA comes from distributed systems integration, CORBA-compliant ORB's may of course be used for integrating objects located on the same node or even in the same address space.

[2] Even when the functionality of a reused class needs to be extended to fulfill the requirements of the application, the problem can often be *factored* into a stub generation problem and an extension problem.

The stub generation problem also occurs in both the static and dynamic invocation interfaces of an ORB. In the context of static invocation, the interface repository is used in a way that resembles the use of a class (or stub) library and the remarks above about class libraries apply. In using the DII, the alternatives for an application are either to write complex code that can analyze and use an interface found in the repository in the form of what amounts to a data-structure at run-time, or to assume a generic interface for the required server and rely on the ORB to supply a dynamic stub-generation service to bridge the protocol gap[3], possibly with some interactive help. The latter is obviously the more attractive option.

Conceptually there are two somewhat orthogonal issues in stub generation. One is the formalization of "compatibility" between interface types that is a generalization of substitutability in the usual subtyping relation, and allows functionality preserving changes in method names, parameter order, etc. The other is both built-in and user-definable mechanisms for type isomorphism between parameter and result types of methods that would allow data and object interchange between client and server objects in spite of differences in representation and structure. I have made some progress in solving both problems. The main ideas and current state of this work are summarized below.

Formalization of Compatibility[4]

Compatibility is a new kind of conformance that can be formalized using an *adaptability* relation between interface types, which is related to but distinct from both subtyping and subclassing. As with the latter, the relation is quite simple except where the object (interface) types are recursive. Recursive interfaces are commonly found in small library classes and are rare for large active components like servers.

The main conceptual difficulty in formalizing the logic of adaptability is the misleading similarity of the concept with both subtyping and inheritance. Clearly, if a class A can emulate B, then an A-object can be used (with some disguise) where a B-object is needed. Treating the disguise as a coercion, this sounds exactly like the normal notion of subtyping. However, a formal application of this intuition leads to an impasse where obviously compatible interfaces cannot be shown to be related. The main insight needed to distinguish subtyping from adaptability is that adaptability is bijective whereas subtyping is injective, when both are seen as mappings realized by coercions. In other words, the set of objects belonging to a subtype typically forms only a part of the set of objects belonging to each of its supertypes, but if the interface of a class A is adapted to *implement* the interface specification of a class B, then every B-object is emulated by a hidden A-object, and every A-object can be converted to a B-object by attaching a suitable interface stub. These ideas (along with certain functionality preserving protocol transformations) can be formalized in a natural way in the form of a logic of adaptability and stub inference. The inference process can also be automated using an algorithm that modifies and extends the (complete) subtype inference algorithm of Amadio and Cardelli [4] for recursive types.

Some interesting issues arise in extending adaptability to generic classes. The main difficulty (assuming that their instantiation is user-specified) is the representation of their interface types since the type parameters of such classes are typically constrained by implicit assumptions about their functionality—for instance they may be used in the implementation as though they support certain operations. It has been found that representing such constraints with full generality requires

[3]Of course, there is theoretically the third option of hand-coding a stub interactively once the required interface is found, but this would require all users to be system experts.

[4]A detailed discussion of current results is given in [3].

F-bounded polymorphism [5] which is significantly more complex than ordinary bounded polymorphism [6]. However, since the reusable generic class and the actual parameters it is used with often come from different domains (the latter typically from the application domain), it is natural to use adaptability rather than subtyping bounds for the parameters and the proof-theoretic properties of adaptability permit the use of ordinary bounded polymorphism without loss of generality. It is worth noting that the interface adaptation of even a single generic class can be quite complex since the actual parameters for it often need stubs, besides the stub needed for the generic class itself.

Some important issues relating to adaptability have yet to be addressed. One is the extension to frameworks, which can be thought of as collections of classes related by subtyping and/or inheritance. The existing logic ensures that the application classes can be related by any *structural* subtyping relation consistent with the subtyping relations among the reusable classes used to implement them. However, there are many unresolved semantic issues. The entire framework is at present built around a simple core-OOL based on the λ-calculus extended with labeled records. I have not yet extended the logic or algorithms to account for the idiosyncrasies of any realistic language.

Type Isomorphism for Flexible Data Transport between Clients and Servers

This is a multifaceted problem with many existing approaches. An excellent survey of the current state of the art is given by Wileden, *et al* [7]. Most current systems support what one might call *data structure isomorphism* (DSI) which is used in data transport among components based on different platforms (languages, processors, operating systems). The typical solution is to define a universal data type definition notation (*e.g.*, IDL [1] or XDR [8]) along with language bindings to translate data to and from the standard format. Data transformations necessary for transport can then be automated—typical RPC stub generation techniques already automate such transformations during transport for distributed client-server interactions [8]. The main shortcoming of DSI is its lack of support for bridging the differences between alternative representations of abstract types, such as the polar and cartesian representations of points in a plane. We need *abstract type* isomorphism (ATI) to fully support data mobility in a heterogeneous environment when data formats differ in more than platform-specific idiosyncrasies. I am interested in exploring an approach where user knowledge about the equivalence of various representations can be expressed declaratively in the form of equational laws (seen as bijections with two-way coercions) between representation types. The problem of establishing equivalence between representation types then reduces to the word problem in algebraic theories for which there are well-established automated techniques based on canonical term-rewriting. There are good research tools available to experiment with these techniques [9, 10]. Inefficiencies due to multistage transformations can be eliminated by using techniques such as deforestation [11] if the transformations are themselves expressed declaratively using a functional language. Note that such transformations are of interest whether or not encapsulation of data-representation is important.

For a discussion of a closely related problem in the context of type reconstruction in functional languages, and the associated equational unification theory, see [12, 13].

3 Comparison

To my knowledge there has been no previous work on automated tools for generating interface stubs of the kind described above, but there are many frameworks and actual systems which provide a suitable habitat for such an extension. These are of two kinds: class library management systems and component connection and interoperability frameworks. A few of these, along with some alternative approaches, are described below.

IBM has implemented a class library framework called SOM [2]. SOM provides language neutral packaging and upwardly compatible binary libraries with support for dynamic linking which permits changes without recompilation of clients and even cross-language subclassing without access to source code. SOM also supports the use of object-oriented technology from procedural languages. The interface stubbing techniques described above clearly complement these strengths and would further enhance the reusability of classes in SOM libraries.

Wileden, *et al* [7], describe an approach to what they call specification level interoperability which is quite similar in spirit to the notion of abstract type isomorphism discussed above. However, their implementation appears to focus on pairs of language-specific stubs to ferry calls and results between static client-server pairs. They do not support knowledge-based automated exploration/selection of possible isomorphisms among representations. In fact, the prototype described in the paper does not support object mobility at all, although they do recognize the need for it as a future extension.

CORBA [1] is perhaps now the most widely accepted framework for interoperability of distributed components. It contains DSI support for data transport at a relatively high level of abstraction based on a type definition facility called IDL. The basic framework contains no notion of automated support for interface adaptation or abstract type isomorphism although both could be seen as "components" (in the CORBA terminology) that define extensions of the basic functionality. CORBA contains an interface repository which would be the natural habitat for such an extension.

Bart [14] is interesting as a concrete ORB (although it does not claim adherence to the CORBA standard) that exemplifies one currently popular approach to object mobility in heterogeneous distributed systems. Bart is realized as a single "bus" that carries (and stores) relational data and works mainly in broadcast mode. The translation of all shared objects to and from the relational form must be specified *explicitly* by the developers of attached components[5]. This allows great flexibility since the representations of objects can be mapped into tuples in arbitrary ways[6]. On the other hand, it also means a lot more work as opposed to automated matching of (say) IDL versions of the representation types by the bus. Note also that automated interface stub generation offers an alternative mode for object mobility in homogeneous contexts where code as well as data is mobile. This alternative respects object encapsulation absolutely. Such an alternative is not possible within the Bart framework.

Genesereth's facilitators [15] are much more ambitious ORBs that support a strong distribution model as opposed to Bart's weak one (there is no single shared bus—each component connects to others through its own facilitator). Facilitators are supposed to use an extension of first-order logic to exchange information about port specifications and to negotiate connections using theorem proving techniques. A similar but simpler model is described by Konstantas [16] where each distributed component consists of a "nucleus" that communicates with the outside world through a "membrane" (a kind of ORB) and the entire system has a "multicellular" structure. An interesting aspect of membranes is that they contain interface type managers that negotiate type compatibility with foreign membranes. This appears to call for stub generation techniques of the kind I propose.

[5] There is a striking similarity to the shared tuple-space of Linda.

[6] The flexibility is further enhanced in Bart by a Prolog-like language called SGL that allows the definition of new relations based on the extensionally represented ones.

References

[1] O. M. Group, "The common object request broker: Architecture and specification." OMG Document 91.12.1 Revision 1.1, 1992.

[2] I. Corporation, "OS/2 2.0 technical library system object model guide and reference." IBM Document S10G6309, 1992.

[3] S. R. Thatté, "Automated synthesis of interface adapters for reusable classes," in *to appear in the Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages (POPL'94)*, ACM Press, 1994.

[4] R. M. Amadio and L. Cardelli, "Subtyping recursive types," in *Proceedings of Eighteenth POPL Symposium*, pp. 104–118, ACM Press, January 1991.

[5] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell, "F-bounded polymorphism for object-oriented programming," in *Proceedings of Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89), London, U.K.*, ACM Press, Addison-Wesley, 1989.

[6] L. Cardelli and P. Wegner, "On understanding types, data abstraction and polymorphism," *Computing Surveys*, vol. 17, no. 4, 1985.

[7] J. C. Wileden, A. L. Wolf, W. R. Rosenblatt, and P. L. Tarr, "Specification-level interoperability," *Communications of the ACM*, vol. 34, pp. 72–87, May 1991.

[8] S. Microsystems, "Network programming guide." Part Number: 800-3850-10, 1990.

[9] D. Kapur and H. Zhang, "RRL: A rewrite rule laboratory," in *Proceedings of 9th Conference on Automated Deduction (CADE-9), Argonne, Illinois, USA*, Springer-Verlag, 1988. LNCS 310.

[10] S. J. Garland and J. V. Guttag, "A guide to LP, the Larch prover," Tech. Rep. 82, DEC Systems Research Center, 1991.

[11] P. Wadler, "Deforestation: Transforming programs to eliminate trees," in *Proceedings of Second European Symposium on Programming*, Springer-Verlag, 1988. LNCS 300.

[12] S. R. Thatté, "Coercive type isomorphism," in *Proceedings of the Fifth Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, pp. 29–49, ACM Press, 1991.

[13] S. R. Thatté, "Finite acyclic theories are unitary," *Journal of Symbolic Computation*, vol. 15, February 1993.

[14] B. W. Beach, "Connecting software components with declarative glue," in *Proceedings of the 14th International Conference on Software Engineering, Melbourne, Australia*, pp. 120–137, ACM Press, May 1992.

[15] M. Genesereth, "An agent-based approach to software interoperation," Tech. Rep. Logic-91-6, Stanford University Logic Group, 1991.

[16] D. Konstantas, "The implementation of the Hybrid cell," in *Object Frameworks* (D. Tsichritzis, ed.), Centre Universitaire d'Informatique, Université de Genève, 1992.

4 Biography

Satish R. Thatté is an Associate Professor of mathematics and computer science at Clarkson University. The focus of his current research is language design and software engineering issues connected with static and dynamic type systems. In the past he has worked in functional programming and symbolic computation including term rewriting systems and equational unification. He has previously taught at the University of Michigan and the University of California at San Diego. He received a Ph.D. in Computer Science from the University of Pittsburgh in 1982.