A design pattern based approach to generating synchronization adaptors from annotated IDL *

H.-Arno Jacobsen
Humboldt University zu Berlin
Institut für Wirtschaftsinformatik
D-10178 Berlin, Germany
jacobsen@wiwi.hu-berlin.de

Bernd J. Krämer
FernUniversität
Lehrstuhl für Datenverarbeitungstechnik
D-58084 Hagen, Germany
bernd.kraemer@fernuni-hagen.de

Abstract

Middleware platforms such as CORBA and DCOM provide standard component interfaces, interaction protocols, and communication services to support interoperability of object–oriented applications operating in heterogeneous and distributed environments. General–purpose services and facilities foster re–use and help reduce development costs. Yet the degree of automation of the software development process is limited to the generation of skeleton and stub code from component interface specifications given in a common interface definition language (IDL). This is mainly due to the fact that the expressiveness of current IDLs is limited to the specification of type and operation signatures. Important properties of crucial components of security–, safety–critical or reactive applications such as object behavior, timing, or synchronization constraints cannot be documented formally, let alone checked automatically.

In this work we continue developing solutions for adding specifications of semantic properties to component interfaces and automatically synthesizing code that instruments corresponding semantic checks. Independently, from the concrete syntax and semantics of such specification elements, we present a collection of design patterns that allow the designer to seamlessly integrate the synthesized code with the code frames generated by standard IDL compilers. We study these approaches along the concrete example of extending CORBA IDL with synchronization constraints and evaluate several implementations, solely based on standardized features of the CORBA standard.

1 Introduction

Several distributed computing platforms and component architectures have come to age over the past few years. The more common ones include CORBA [OPR96], DCOM [Mic96], JavaBeans [Eng97], DCE [RKF92], and ANSAware [ANS93]. They all aim at insulating distributed applications from the underlying proprietary infrastructure to achieve interoperability across disparate hardware platforms, network protocols, and operating systems. A common interface definition language (IDL) serves to package heterogeneous component implementations with uniform interface specifications. Thus server components are made accessible to clients written in virtually any programming language. Many IDLs such as CORBA IDL, ODL, or ANS.1, typically specify component interfaces in terms of module names, interface names, structured types, and signatures of operations, have been developed. A signature defines the operation name, return type, argument mode and type, and possibly an exception type.

This simplicity and generality ensures that IDL is applicable to a wide range of application domains and can be mapped to a large variety of implementation languages. They include, for example, C, C++, Ada, Smalltalk, Cobol and Java in the case of CORBA IDL. The price for this generality is that:

- semantic properties of application objects such as their functional and dynamic behavior, timing and synchronization constraints, or quality of service requirements cannot be formally documented in the object interfaces,
- specifications cannot be analyzed for consistency and behavioral properties such as liveness, safety, or fairness,
- correctness and conformance of object implementations cannot be verified, and
- automatic synthesis of code from specifications is limited to the generation of header files, skeleton, and stub code providing some degree of communication and location transparency.

Compared with state–of–practice specification and design languages such as OMT with its structural, dynamic, and functional views on object specifications [RBP+91] and related support, and code generation tools such as Rhapsody [HG97], IDLs are expressively weak. The consequence are growing number of

*This work was partly carried out while the authors were at the International Computer Science Institute in Berkeley. The first author is supported by the German Research Society, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316). The second author's work is sponsored by the Federal Minister of Education, Science, Research and Technology under grant KAN 018/97 INF.

proposals for IDL extensions, application–specific IDL conventions, and supplements including:

- The development of the component definition language (CDL) to express the interaction of business objects at the meta-level by the OMG. OMG's adoption of the Unified Modeling Language (UML) for analysis and design;
- the inclusion of real–time [SLM97, WBTK96], quality of service [ZBS97], behavioral [Zad97], and quality assuring [AJ96] annotations into IDL;
- annotations invisible to the IDL compiler that impose synchronization constraints on the operations visible at the object interface [Krä98].

A crucial aspect of all these proposals is the question of how they are implemented. Certainly, the simplest way is to wait for the standard and its implementation by a CORBA vendor. But this may take some time and applications exploiting such IDL extensions are not portable as long as they depend on individual vendor platforms. This is likely to happen in the case of UML. But it is rather unlikely that language constructs for specifying real–time requirements, synchronization constraints, functional or dynamic behavior will ever be included in future versions of IDL. The TAO approach to associate real–time semantics with predefined IDL types lacks portability as object implementations rely on the existence of real–time object adapters and suitable precautions in the object request broker (ORB) [HOLS98]. In general, such modifications to proprietary CORBA platforms are not feasible as the standard lacks sufficiently detailed middleware API specifications and leaves a wide range of design decisions to CORBA vendors (cf., e.g., [Jac97]).

To escape this trap, the approach described in [Krä98] proposes to include semantics annotations as comments in IDL interface definitions and separately compile these annotations into code implementing corresponding sanity checks.

In this paper we further explore this idea by searching for design alternatives that exploit different features of the CORBA standard to seamlessly integrate synthesized synchronization code with manual implementations of the object's functionality. These solutions are developed into a suite of design patterns for implementing IDL extensions that co–exist with standard IDL compilers. Prototype implementations of the proposed design patterns, which are ongoing, serve to empirically investigate their pros and cons.

Our approach is based on the current CORBA specification for which several proprietary and public domain implementations exist. In Section 2 we briefly review the CORBA standard and its interface definition language to the extent necessary for understanding the design solutions. Then we elaborate means to associate synchronization constraints with IDL interfaces along a simple example. In Section 4 we develop a collection of design patterns to synthesize portable code instrumenting these constraints in terms of before– and after–tests. We also identify problems resulting from the immaturity of the current state of the CORBA specification and propose extensions which would allow for a more extensible distributed system infrastructure.

2 CORBA: Distributed Object Computing Middleware

In an open distributed computing world we are confronted with a constantly changing infrastructure. It typically consists of a diversity of proprietary hardware and software components, protocols, operating systems, programming languages, and development tools. For distributed applications operating in such a heterogeneous computing environment, service and information discovery, and client/server interoperability are key issues [Vin97].

2.1 The Common Object Request Broker Architecture

The Common Object Request Broker Architecture (CORBA) is a standard for distributed computing which has been developed by the Object Management Group (OMG, [OMG91]). CORBA aims at providing a uniform communication infrastructure for building distributed applications. It provides mechanisms, protocols, and services that allow application developers to integrate software components operating on different hardware platforms and operating systems into a coherent logical entity. CORBA has also been designed to support programming language interoperability. This is to allow for full flexibility in application design and development, as well as, to facilitate the integration of legacy systems and legacy code into distributed applications.

Interoperability is achieved by packaging all component implementations with uniform interface specifications using CORBA's interface definition language (IDL). IDL is a descriptive, non–algorithmic 'lingua–franca' following a C++–like syntax with added features for expressing distributed processing [Sie96]. Interface specifications are compiled into stub code written in the component's implementation language. The stub code is linked with hand–written code implementing the actual application semantics and with CORBA library components implementing infrastructure services. The stub code handles communication with remote machines via the Object Request Broker (ORB). This includes argument packaging, data marshaling, and un–marshaling. To realize interaction and communication between distributed components, a broker mechanism is deployed. It finds remote components, possibly activates them, invokes the requested operation, and returns eventual results to the invoking client object. All this is fully transparent to the interacting components.

2.2 CORBA IDL

CORBA IDL is a simple descriptive interface definition language designed to be easily represented by a large range of programming languages. An IDL language mapping describes the representation of IDL statements and expressions in the target programming language. Mappings for C, C++, Smalltalk, Java, Ada, and COBOL have been standardized so far. IDL allows one to define component interfaces by listing

their operation signatures, types, and attributes. Each signature contains an operation name, a return type, a list of typed formal parameters including a mode indicating for each parameter whether the actual value is passed from client to server, from server to client or both (in, out, inout, respectively). The signature may also include exceptions to be raised by the declared operation.

The concrete IDL specification of a simple bounded buffer object providing two operations `put` and `get` that allow independent client objects to deposit and remove items in and from a buffer, is depicted in Fig. 1. The read–only attribute `bufsize` models the maximal capacity of the buffer. This example will serve us throughout the rest of the paper as a running example.

```
interface BoundedBuffer {
  // size of the buffer
  readonly attribute short bufsize;

  // method for extracting an element                    5
  short get();

  // method for inserting an element
  void put(in short value);                          };
```

Figure 1: IDL definition of a bounded buffer interface.

2.3 Generic IDL Compilation Framework

The CORBA standard precisely defines the language mappings supported and the interfaces of client–side stubs and server–side skeletons into which IDL specifications are compiled. The interfaces between stub and ORB, skeleton and object adapter, object adapter and ORB, however, are proprietary and therefore generally not open to manipulation by middleware users (cf. Fig. 2).

To describe the integration of object implementations and formalize design solutions implementing the proposed IDL extensions, we use a design–pattern–like notation. This allows us to capture universal concepts apt for different implementations [GHJV95]. Supporting code samples are presented in a C++–like syntax.

In Fig. 3 we use the OMT notation to illustrate how an object implementation is integrated into the distributed computing platform. This figure depicts the inheritance relationship that holds among the ORB, the CORBA objects, and the developer's object implementation (the service implementation).

3 Annotating Object Interfaces

The services and facilities coming with a CORBA implementation support re–use and thus help reduce development costs. But the degree of automation of the software development process is limited to the generation of skeleton and stub code for the language mappings supported. This is mainly due to the fact that CORBA IDL, like other IDLs, only allows the

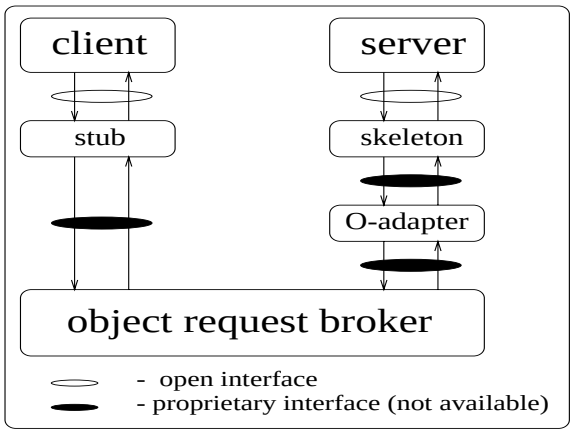

Figure 2: Open and proprietary interfaces to the ORB.

formalization of syntactic properties of object interfaces but lacks the specification of semantics. This is different for object modeling techniques such as OMT or UML, formal description techniques such as SDL or Lotos, or Meyer's principle of design–by–contract [Mey92]. They capture semantic properties such as an operation's functional semantics (specified, e.g., in terms of pre–, post–conditions, and invariants or algebraic equations) or an object's dynamic behavior (expressed, e.g., in terms of state–charts or process specifications). Such elaborate specifications bear the potential for detailed semantic analyses and the automatic synthesis of code implementing the specified behavior or checks verifying the observance of specified properties.

3.1 Synchronization Constraints at Object Interfaces

In [Krä98] we proposed means to include synchronization constraints as part of interface definitions. To be compliant with the CORBA standard, these constraints occurred as comments to the IDL specification. The language constructs used are interpreted by logic expressions characterizing partially ordered executions of interface operations.

Inheritance anomalies that were subject of discussion for a long time in the literature on object–based concurrent programming languages (cf., e.g., [BY87]), are avoided. It was also shown that the given set of synchronization operators can be implemented by means of state variables keeping track of the history of operation executions per object implementation. The design of a tool that compiles augmented IDL interfaces into code implementing corresponding sanity checks was sketched.

In the following section we extend this work by developing a collection of design patterns and implementation techniques for including synchronization constraints in object interfaces. These solutions can be adapted to develop other extensions to interface con-

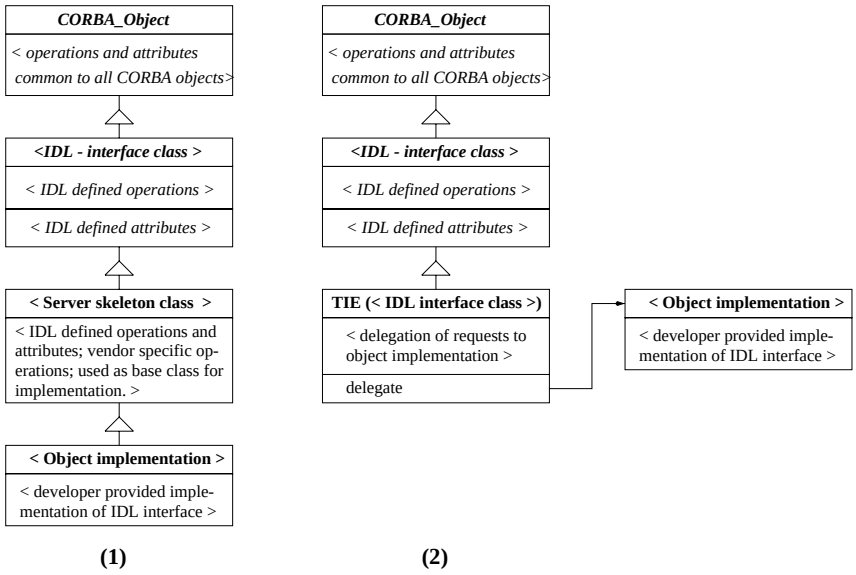

Figure 3: Integrating an object implementation with code generated by IDL compilers. (1) Inheritance–based: Object implementation inherits from server skeleton; (2) TIE–based: Object implementation and server skeleton are *tied* together via method delegation

tracts, if the extensions can be implemented by activating suitable checking code before and after an operation invocation is executed. Candidates are functional behavior expressed in terms of pre–, post–conditions, and invariants [Mey92] or dynamic behavior specified by associating interface operations with state transitions [HG97]. To focus the discussion, we illustrate our design alternatives for the case of synchronization constraints. The bounded buffer example, introduced in the previous section, serves to illustrate our solutions. This example is further elaborated in the following subsection.

It should be noted that this example serves to illustrate the inner workings of a generic framework that allows the application developer to seamlessly include her favorite specification dialect and corresponding checking code into a standard CORBA based development environment. Our objective was not to propose yet another specification or assertion checking technique as they are discussed by the specification language and software architecture communities.

3.2 Bounded Buffer Revisited

A bounded buffer acting in a distributed environment gives rise to several kinds of synchronization constraints:

- *mutual exclusion*: different invocations of the `get` operation must not be executed concurrently; the same holds for different invocations of the `put` operation;
- *capacity limitation*: no `bufsize` more `put` than `get` invocations are allowed in any computation to avoid buffer overflow;
- *precedence constraint*: there must never be more executions of the `get` than there are executions of the `put` operation to prevent underflow.

In addition we might want to add a *strong fairness* condition requiring that any client c_i must not execute the `get` operation k times more often than any other client c_j, provided c_j has tried to invoke the `get` operation, at all; a similar fairness requirement exists for the clients of the `put` operation. Priorities among competing invocations of interface operations also express synchronization needs by requiring the selection of execution candidates among a collection of currently invoked operations.

Fig. 4 presents the IDL of the bounded buffer example annotated with synchronization specifications as presented in [Krä98]. Synchronization specifications act like negated guards [Frø92]. The intuitive meaning of the $mutex(m, n)$ predicate is that an operation invocation m is not executed if another invocation n is currently executed, and vice versa. Invocations of some operation m occurring in a $dist(m, n, k)$ predicate are disabled if the difference of the number of executions of m is equal to the number of executions of $n + k$ (buffer is full), while invocations of n are disabled if the number of executions of m and n are equal (buffer is empty). Conceptually, such conditions can be verified by reference to a record of the server object's history of execution events. This idea will be exploited in the following section.

```
interface BoundedBuffer {
  readonly attribute short bufsize;
  // the buffer takes at most 'bufsize' elements
  //--sc: dist(put,get,bufsize)
                                                    5
  short get();
  // get invocations must be processed in sequence
  //--sc: mutex(get,get') for get =/= get'

  void put(in short value);                         10
  // put invocations must be processed in sequence
  //--sc: mutex(put,put') for put =/= put'     };
```

Figure 4: IDL specification with synchronization annotations.

4 Design Patterns for Implementing Synchronization Constraints

In this section we describe various techniques and design patterns that implement the synchronization constraints previously defined. Code implementing the IDL annotations is synthesized and automatically integrates with the object implementation of the application developed. The strength of our approach is its reliance on standardized CORBA features only, we strictly avoided exploiting proprietary extensions.

4.1 Synchronization Constraints at the Code–level

Context: In a distributed computing environment no assumptions can be made about a specific order in which the operations of a component interface are invoked because different clients of a shared component act concurrently. But unsynchronized accesses to shared components are likely to cause inconsistencies in the components' states. They include an over– or under–flow of limited resources, overwriting of information due to concurrent write updates to the same partition of a repository, unfair uses of a shared resource, or illegal execution orders. To maintain the consistency of state, the developer often has to take precautions that synchronize concurrent invocations at component interfaces.

Synchronization can be achieved in many ways. Locking is a traditional mechanism to maintain the consistency of a resource in the presence of concurrent accesses. The concurrency control service of CORBA provides a locking mechanism to mutually exclude accesses of concurrently executing transactions or non–transactional threads of control. A drawback of such services is that they provide programming solutions only. The locking requirements are not documented at the component interface. This lack of contractual information prevents decent analyses of the proper interworking of concurrent components to detect potential deadlocks, blockings, and other forms of unfair uses prior to constructing and testing executable code. Further the observance of such properties by the actual implementation cannot be rigorously verified. Moreover, mutual exclusion is only one way to synchronize the execution of a set of concurrent operation invocations. There may be other causal dependencies among the operations of a component interface that require the developer to:

- impose a precedence on certain operation executions,
- defer executions to prevent violations of capacity constraint such as over– and under–flow of limited resources, or
- guarantee the fair use of a shared component by multiple clients.

Problem: CORBA IDL offers no means to specify synchronization constraints. At best, synchronization constraints are hidden in object implementations. This complicates design, validation, maintenance, and evolution of distributed applications. Further, developers cannot be sure whether a new object implementation conforms to the behavior of the one it is going to replace [Sch98].

Solution: First augment server interfaces by synchronization constraints suggested by the application semantics based on the language constructs defined in [Krä98]. To enable checks of synchronization constraints in the server implementation define two variables $\#start\_m$ and $\#end\_m$ for each operation m involved in a synchronization constraint. Prior to activating the code implementing the functionality of some operation m, verify one or more the following conditions — depending on the constraint expression in which m occurs — and prevent the execution of m while either of the corresponding conditions given below hold:

mutex(m,n): $\#start\_m > - \#end\_n \neq 0$ (n is active);

alt(m,n): $\#end\_m - \#end\_n = 1$ (m must begin and is at most one step ahead n);

alt(n,m): $\#end\_n - \#start\_m < 1$ (once n completed an invocation, it's m's turn to execute next);

dist(m,n,k): $\#start\_m - \#end\_n = k$ (capacity k is exhausted);

dist(n,m,k): $\#end\_n - \#start\_m = 0$ (m cannot be started more often than n has terminated).

Similarly other types of constraints can be checked. If the actual condition holds, the execution of m is disabled. If m is involved in more than one synchronization constraint, m is disabled if either of these constraints is true. Otherwise variable $\#start\_m$ is incremented by one, the code implementing m's functional behavior is executed and finally variable $\#end\_m$ is incremented. This is illustrated in Fig. 5 for some operation m with result type op_t defined in interface sv with two synchronization constraints (note that the naming conventions for the state variables are slightly different).

Benefits.

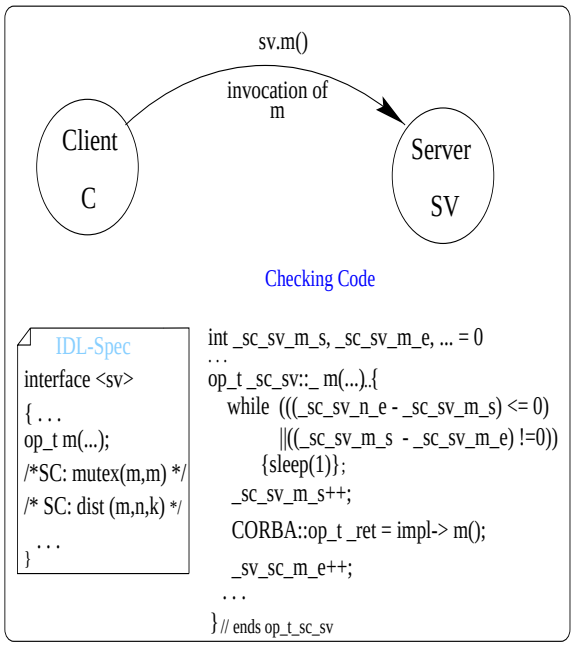

Figure 5: Synchronization constraints checking pattern.

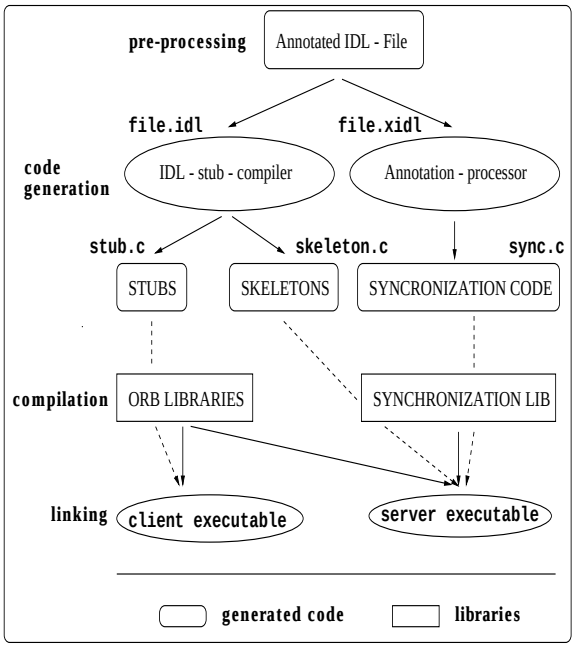

Figure 6: Development steps

- Concurrent accesses to shared objects are properly synchronized.
- A wide range of causal dependencies can be specified and verified, solely by reference to variables maintaining operation state.
- The formal semantics underlying the synchronization constraints enables rigorous analysis at the specification level based on the formal model underlying the constraint expressions.
- The code for synchronization checking can be synthesized automatically from the specification.

4.2 Development Steps and Pre–processing

In the following subsections we study several approaches towards an automatic implementation of synchronization constraints and their seamless integration with the code frames generated by standard IDL compilers and the developer's operation implementation. The solutions we present aim at portability. Fig. 6 provides an abstract view on the development steps including the different code fragments and compilation stages incurred.

The individual steps are:

1. Write IDL specification of server object,
2. annotate the signature specification with synchronization constraints,
3. *pre–process* the annotated IDL specification,
4. *generate* stubs and skeletons from the IDL specification with standard IDL compiler,
5. *generate* support code from the annotated IDL specification (this code implements the patterns described below for the management of the synchronization constraints according to the specification),
6. *compile* all resulting files, and
7. *link* the code with broker libraries, application code, and synchronization management libraries.

For the pre–processing stage several alternative approaches are possible which manifest themselves in the manner the synchronization constraints are expressed.

The constraints may be expressed as comments in the IDL specification file itself. This has the advantage that the file still parses through the standard IDL compiler. Maintaining the same advantage but incurring greater management efforts, the synchronization annotations may be kept in a separate file. Expressing the annotations together with IDL in a new language IDL+ leads to a more coherent specification language for the cost of a new IDL+ compiler.

We decided to express the annotations as IDL comments. Processing of the constraints is thus performed by a separate compiler that interprets the provided comments.

4.3 Inheritance–Based Solution

This solution uses class inheritance for integrating an object implementation in a CORBA platform. Instead of deriving the class instantiating the object implementation directly from the generated skeleton class, an *adapter class* is generated in the pre–processing stage by the annotation compiler. This class derives from the skeleton class generated by the IDL compiler and from another *synchronization class* which implements the synchronization constraints. This inheritance relationship is depicted in Fig. 7. The actual object implementation is 'plugged into' the platform by the newly generated adapter class which delegates invocations on its behalf.

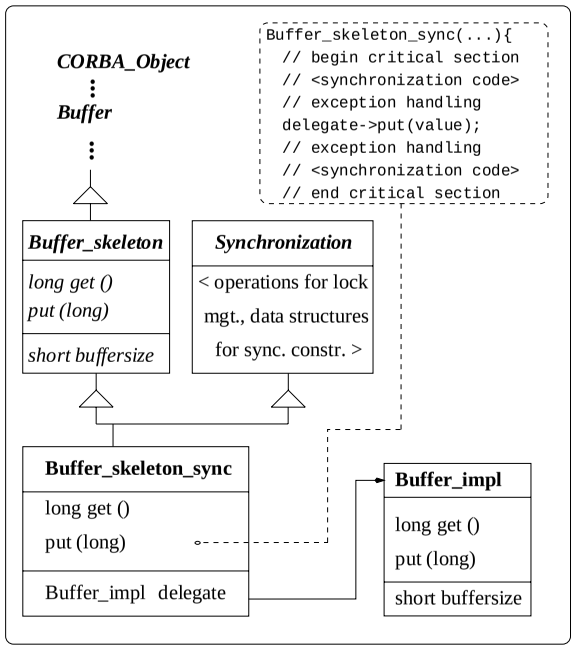

Figure 7: Modified inheritance structure to accommodate the class implementing the synchronization constraints for the inheritance-based design.

A client method invocation to the `get()` operation, for instance, is dispatched to its receiver, a server–proxy implementing the synchronization that delegates the invocation to the actual object implementation. If the constraint is not satisfied, the proxy defers the invocation. Excerpts of this code are shown in Fig. 8.

4.4 Delegation–Based Solution

This solution builds on the TIE–approach for integrating an object implementation in a CORBA platform (see Fig. 3). It 'ties' the platform and the object implementation together. This is achieved by generating a class that delegates client method invocations to operations of the object implementation (cf. Fig. 9). This approach is particularly useful for integrating components written in programming languages not supporting inheritance.

```
// file buffer_i.h
// Object implementation.
#include ''buffer_s.h''

class BoundedBuffer_impl {                                  5
public:
    BoundedBuffer_impl();
    ~BoundedBuffer_impl();
    // USER BoundedBuffer VIRTUALS
    virtual CORBA::Short bufsize();                         10
    virtual CORBA::Short get();
    virtual void put(CORBA::Short value);  }

// file: buffer_i_sync.cpp
// Synchronization code generated and                       15
// delegation to object implementation

#include "buffer_i.h"
#include <iostream.h>
                                                            20
BoundedBuffer::BoundedBuffer_impl_sync(){}
// 1. Check in with concurrency control mechanism.
// 2. Set up BoundedBuffer_impl instance for delegated calls.

BoundedBuffer::~BoundedBuffer_impl_sync(){}                 25
// 1. Sign off with concurrency control mechanism.
// 2. Delete BoundedBuffer_impl instance
// Note: step 2. depends on activation mode of server.
// Generated 'BoundedBuffer VIRTUALS' annotated with
// synchronisation code and delegated calls to              30
// BoundedBuffer_impl instance.

CORBA::Short BoundedBuffer_sync_impl::bufsize(){
    // *** begin critical section <syncronization code>
    result = delegate->bufsize();                           35
    // <syncronization code>
    // *** end critical section
    return result;          }
...
```

Figure 8: Implementation fragments for the buffer object

For synchronization constraint management we generate an *adapter class* analogously to the above solution. This time, however, it only inherits from the class providing the synchronization code. The adapter class delegates method invocations to the appropriate operations of the object implementation after performing synchronization management checks. In the manner explained above the adapter class is 'tied' together with the generated skeleton code. Thus, a client method invocation dispatched through the server skeletons arrives at its receiver, the appropriate adapter class, through delegation. In the adapter a synchronization check is performed and the call is again delegated to the object implementation of the invoked operation.

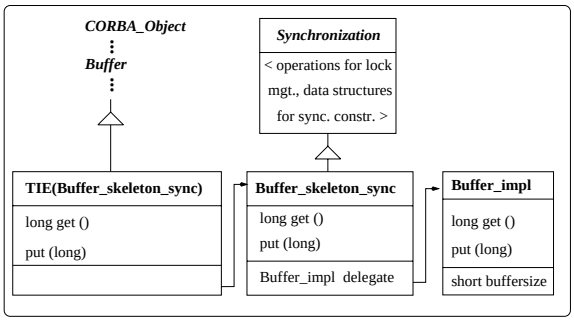

Figure 9: Modified Inheritance structure to accommodate the class implementing the synchronization constraints for the TIE–based design.

4.5 Dynamic Invocation and Dynamic Skeleton Interface based Solution

The DII and DSI are interfaces that grant direct run–time access to the object request broker's communication layer — i.e., requests and invocations may be generated dynamically — without prior compile time knowledge of method signatures and formal parameter types. The dynamic nature of these interfaces is not of much help for solving our problem, however. rather the direct access to the communication layer offers the key benefit as opposed to the above solutions that were based on CORBA's static invocation (SII) and static skeleton interface (SSI). This direct access can be exploited to interweave synchronization constraint management calls with object implementation upcalls. Since all necessary information is available statically, the entire dynamic stub (i.e., the steps that have to be taken to set up a dynamic call) can be generated automatically.

The main disadvantage of this approach is the inefficient nature of the DII and DSI. This has been extensively discussed in the literature, for example, in [GS98, OH97].

4.6 Specialized Object Adapter

When the CORBA standard was first introduced, it was envisioned that many specifications of specialized object adapters would follow (e.g., adapters for different kinds of databases). So far, only one additional adapter has been standardized, and it only serves to solve portability problems inherent to the initial adapter.

However, especially for the kind of extension we are proposing, a synchronization constraint based object adapter is a possible alternative. As we have outlined above, crucial interfaces on the server side of the distributed computing platform are not open. It is therefore difficult to implement a proper object adapter without knowing intimate details of a given ORB implementation. Clearly, this would not be a portable solution.

4.7 Proper IDL–compiler

The design of a proper IDL compiler for managing the extended specification language is certainly the most straight forward solution to the problem. Based on previous experience (cf. [Jac97]), we have to report that this is unfortunately not a universal solution because crucial ORB interfaces are not open. One may therefore only resort to this approach when the ORB interaction is based on the DII/DSI or when the source code of the implementation is at hand.

4.8 Proprietary Solution: Orbix Filters

Some CORBA products provide proprietary extensions to the CORBA standard. Iona's Orbix, for instance, provides a feature referred to as filters. A filter is a hook that allows the user to execute a function just before and just after an invocation is executed. Clearly, this feature is well–suited for managing synchronization constraints. From the examples above, it should be obvious how to adapt this feature to our case.

Interceptors, a novel feature that is derived from filters but is more general in nature, has recently been added to the CORBA standard. Unfortunately it has not yet been implemented in any ORB, as yet. Moreover, the current interceptor specification is incomplete. A revision task force within OMG is revising it. For the motivated case of extending IDL with synchronization constraints interceptors would solve some of the problems addressed. For other extensions, where more elaborate processing is required, a solutions based on the patterns discussed above would have to be taken.

5 Implementation

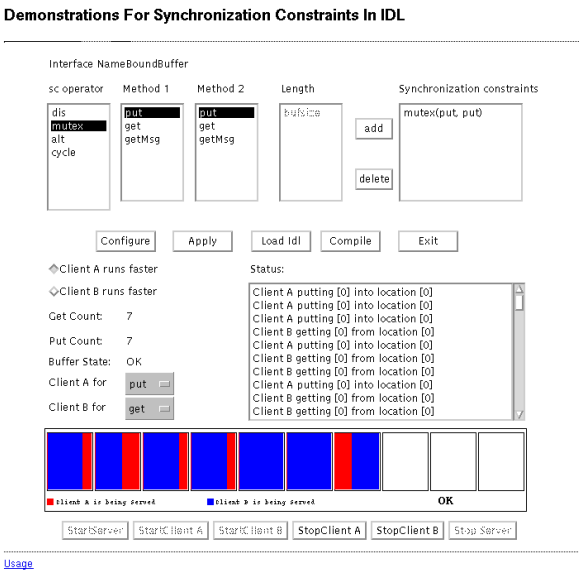

Figure 10: Test applet

To illustrate and test our approach, Ji Zhang, a guest researcher at FernUniversität, who is on leave

from the East China Research Institute of Computer Technology (ECI) in Shanghai, has implemented an experimental version of one of the design patterns described in this paper. The components of this test application for IDL with synchronization constraints include a service agent, a bounded buffer object, two client objects, and a graphical user interface depicted in Fig. 10. This applet serves to control experiments with this configuration. It allows testers to add or delete synchronization constraints formed over a predefined collection of synchronization operators, apply these to the configuration at hand, compile the enhanced IDL file, start the service agent with the compiled code and run either or both clients to observe their behavior on the buffer, whose slots are shown in the lower part of the applet. These slots are organized in a ring and the effect of Client A or B on a slot is depicted by different color codings. To make the experiments more interesting, the clients can be set to run at different speeds. In addition their action is recorded in a status window. The color coding shown in Fig. 10 illustrates the undesired interleaving of the two clients per slot, which is possible if no constraints are in effect.

The service agent's task consists in setting up an initial environment, maintaining changes to the IDL specification, run the IDL compiler of our Visibroker implementation, and start and stop the buffer server and its clients.

An extension of our test configuration that allows the definition of additional synchronization operators and corresponding checking code in C++ is under way.

A summary evaluation and comparison is depicted in Table 1. Some of the results listed there are still speculative as we have implemented prototype solutions only for a subset of the design alternatives discussed.

6 Conclusion

In this paper we investigated a number of alternative solutions to add semantic information to CORBA IDL interfaces. We designed tools that automate the synthesis of corresponding checking code. We demonstrated how to integrate this code with the skeletons generated by CORBA IDL compilers and the developer's object implementation.

Our design solutions exploited different mechanisms of the CORBA standard including class inheritance, delegation, dynamic interfaces, and specialized object adapters. These solutions were illustrated with synchronizing access to a bounded buffer.

We also argued that the solutions presented in the main body of this paper may also be useful to implement other IDL extensions capturing, for example, the functional behavior of interface operations or their dynamic behavior. Further test implementations of missing design alternatives are currently underway and first attempts with IDL extensions in terms of pre– and post–conditions are planned for the near future.

Acknowledgements

We would like to thank Ji Zhang for implementing an experimental tool based on the ideas presented in this paper. We would like to thank Rudolf Müller for giving us feedback on an earlier version of this manuscript.

References

[AJ96] M. Anlauf and S. Jähnichen. Qualitätssicherung in objektorientierten Client/Server-Architekturen. Technical Report, GMD FIRST, Berlin, 1996.

[ANS93] Architecture Projects Management Limited, Castle Park, Cambridge. *Application Programming in ANSAware*, release 4.1 edition, February 1993.

[BY87] J.B. Briot and A. Yonezawa. Inheritance and synchronization in concurrent OOP. In *Proceedings of ECOOP87*, pages 32–40, 1987.

[Eng97] R. Englander. *Developing Java Beans*. O'Reilly, 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. aw, 1995.

[GS98] Andy Gokhale and Douglas C. Schmidt. Measuring and optimizing corba latency and scalability over high-speed networks. *IEEE Transactions on Computing*, April, 1998.

[HG97] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[HOLS98] Tim Harrison, Carlos O'Ryan, David L. Levine, and Douglas C. Schmidt. The design and performance of a real-time corba event service. *IEEE Journal on Selected Areas in Communications*, 1998. (Submitted for publication.)

[Jac97] H.-A. Jacobsen. User extensible programming language interoperability with CORBA. In *CORBA Management Workshop*, Dublin, Ireland, OMG, 22 September, 1997.

[Krä98] Bernd J. Krämer. *Synchronization Constraints in Object Interfaces*, volume B.J. Krämer and M.P. Papazoglou and H.-W. Schmidt, chapter 5. Research Studies Press (Wiley & Sons), 1998.

[Frø92] S. Frølund. Inheritance of synchronisation constraints in concurrent object-oriented programming languages. In O. Lehrmann Madsen, editor, *ECOOP 92*, volume 615 of LNCS, pages 185–196. sv, 1992.

[Mey92] Bertrand Meyer. *Eiffel – The Language*. Prentice Hall, 1992.

	adaptor	DSI	spec. OA	compiler	filter	manual
std. conform	yes	yes	no	no	no	yes
portability	yes	yes	no	no	no	no
performance	good	poor	v. good	v. good	good	good
impl. effort	moderate	high	high	high	low	low
client server side extension	server only	client & server	server only	client & server	client & server	server only

Table 1: Comparison of different implementation patterns.

[Mic96] Microsoft and Digital Equipment Corporation. *Distributed Component Object Model Specification*, draft version 1.0 edition, October 1996.

[OH97] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA*. Wiley, 1997.

[OMG91] The common object request broker: architecture and specification. Technical Report 91.12.1, OMG, December 1991.

[OPR96] Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA*. Prentice-Hall, 1996.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[RKF92] Ward Rosenberry, David Kenney, and Gerry Fisher. *Understanding DCE*. O'Reilly & Associates, Inc., 1992.

[Sch98] Heinz-W. Schmidt. *Compatibility of Interoperable Objects*, volume B.J. Kr mer and M.P. Papazoglou and H.-W. Schmidt, chapter 6. Research Studies Press (Wiley & Sons), 1998.

[Sie96] J. Siegel. *CORBA fundamentals and programming*. John Wiley and Sons, 1996.

[SLM97] Douglas C. Schmidt, David Levine, and Sumedh Mungee. The design of the tao real-time object request broker. *Computer Communications*, 1997.

[Vin97] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[WBTK96] V. F. Wolfe, J. Black, B. Thuraisingham, and P. Krupp. Real-time method invocations in distributed environments. Technical Report, University of Rhode Island, Jan 1996.

[Zad97] V. Zadorozhny. Towards an integrated corba/raise semantic interoperable environment. Technical Report 117, UNU/IIST, P.O.Box 3058, Macau, July 1997.

[ZBS97] J. A. Zinkey, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):56–73, 1997.