Design Patterns for Synchronization Adaptors of CORBA Objects *

H.–Arno Jacobsen
Humboldt University, Berlin
Institute of Information Systems
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, DCE, DCOM, and JavaRMI offer standard interface definition languages, interaction protocols, data exchange formats, and communication services to achieve interoperability and support system integration for software development in heterogeneous and distributed computing environments. Integration is also achieved by common interface definition languages (IDLs) that serve to specify module names, interface names, and operation and type signatures in a uniform way. The simplicity of IDLs ensures that these languages are applicable to a wide range of application domains, can be mapped to a wide variety of implementation languages, and are simple to learn. However, for certain security and safety critical or reactive applications there is an urgent need to express other aspects of the software under development. Such aspects include synchronization constraints, pre- and post conditions, invariants, QoS annotations, and real-time annotations.

To leverage this semantic mismatch of current interface definition languages and domain specific extensions, we discuss solutions for adding specifications of semantic aspects 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 implementation alternatives, solely based on standardized features of the CORBA standard. We demonstrate the effectiveness of our approach through an IDL-annotation compiler that synthesizes support code portable across different CORBA implementations.

Keywords

IDL, extended interface definition language, synchronization constraints, synchronization code pattern.

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], and DCE [RKF92]. 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.

*Forschungsberichte des Fachbereichs Elektrotechnik 2/1999 ISSN 0945-0130 (Technical Report). (Submitted for publication.)

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. A signature defines the operation name, return type, argument modes and types, 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. But the price for this generality is that:

- different semantic properties of application objects such as functionality, dynamic behavior, timing requirements, synchronization constraints, or quality of service requirements cannot be formally documented in the object interfaces,
- the separation of a distributed system's requirements and constraints into a number of concerns that can be addressed, analyzed, and evolved in isolation throughout the lifetime of an application is not supported, and
- the 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 UML with its structural, dynamic, and functional views on object specifications [BR98], IDLs are expressively weak. The consequence are a growing number of proposals for IDL extensions, application-specific IDL conventions, and supplements including:

- the component definition language (CDL) developed by the OMG to express the interaction of business objects at the meta-level;
- the inclusion of real-time [SLM97b, WBTK96], quality of service [ZBS97], behavioral [Zad97], and quality assurance [AJ96] annotations into IDL;
- annotations invisible to the IDL compiler that impose synchronization constraints on the operations visible at object interfaces [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 such reference implementations may take time and applications exploiting such IDL extensions are not portable as long as they depend on individual vendor platforms. 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 [Jac97b].

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. The general idea relates to the motivation behind aspect-oriented programming [LTdMK98], namely to allow a distributed application to be constructed by describing each concern separately. The main difference is that we aim at a program specification rather than design level and use the specification to generate code implementing non-functional properties of programs.

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. Section 3 provides a survey of IDL extensions under investigation together with a discussion of their benefits and drawbacks. Section 4 then focuses on one distributed software aspect that is orthogonal to an object's functionality and serves as a pattern for extended interface specifications, namely synchronization constraints. We introduce a semantic model of non-sequential behavior on which we build IDL annotations. A simple example illustrates the approach. Section 5 develops a collection of design patterns to synthesize portable code that instruments such constraints in terms of before- and after-tests. Section 6 reports on our experiences in practical implementations of these design patterns and argues about their potential to carry over to other non-functional software aspects.

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 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' with a C++–like syntax that adds 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. Figure 1 shows the general structure of an IDL file.

```
module <identifier>
  { // module-scope
      <type declarations>;
      <constant declarations>;
      <exception declarations>;

      interface <identifier>   [:<inheritance>]
       {
         <type declarations>;
         <constant declarations>;                                    10
         <attribute declarations>;
         <exception declarations>;

              [<op_type>] <identifier> (<parameters>)
              [raises exception ][context];
                 ...
              [<op_type>] <identifier> (<parameters>)
              [raises exception ][context];
        }// end interface <identifier>
      ...                                                            20

      interface <identifier>   [:<inheritance>]    {...}
}// end module <identifier>
```

Figure 1: Schema of a CORBA IDL specification.

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. 2. 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
  short get();

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

Figure 2: 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. 3).

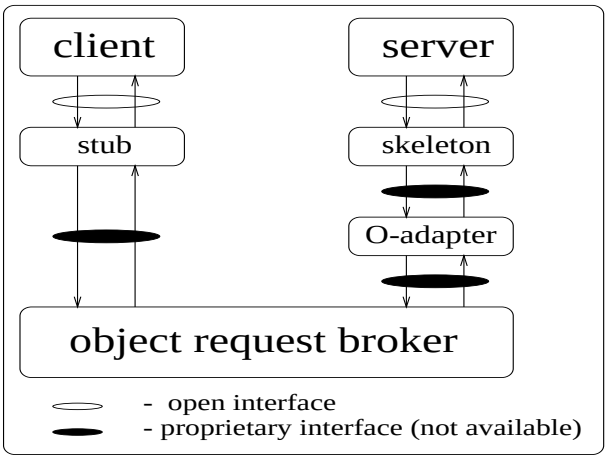


Figure 3: Open and proprietary interfaces to the ORB.

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. 4 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 Extending interface definition languages

In this section we present a comprehensive survey of interface definition language extensions and discuss proposals for processing the extensions in standard environments.

IDLs have been designed to provide interfaces to objects and components in an implementation language independent manner, with the objective of keeping the interface independent of the object implementation. Particularly, the OMG IDL has been designed to be expressible by a wide range of different language paradigms. This generality warrants many design tradeoffs. Operation overloading in derived interfaces, for instance, has been dismissed, since it cannot be supported by all target implementation languages in a developer intuitive manner. Note, the realization of IDL language features (e.g., inheritance, exceptions, overloading) must be represented in the "programming language mapping" applied by the application developer.

IDLs have therefore been kept purely syntactical expressing operation signatures, defining namespaces, and interfaces only. Most common IDLs lack any semantic capabilities to express behavioral annotations, quality of service attributes, interaction protocols, synchronization constraints, support for special application domains, object grouping facilities, and miscellaneous extensions. This deficiency has lead to many language extensions. We now provide an exhaustive classification.

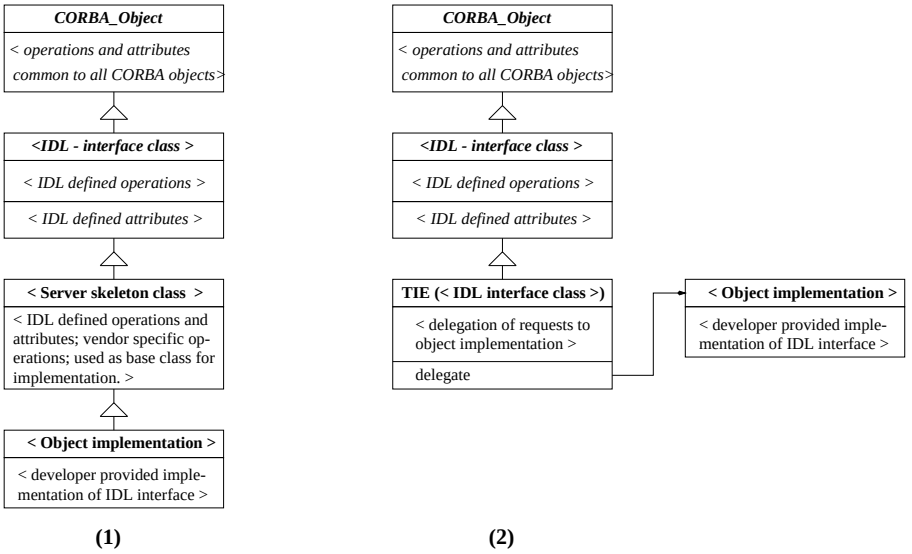


Figure 4: Integrating an object implementation with code generated by IDL compilers. (1) Inheritance–based: Object implementation inherits from server skeleton; (2) TIE–based: Object implementation and server skeleton are *tied* together via method delegation

Expressing behavior and semantic

Much work has focused on annotating IDL with behavioral extensions, such as pre- and post-conditions, invariants, abstract operation semantics, data integrity conditions, and Horn clauses.

The Assertion Definition Language [SH94, X/O96] a specification language developed at Sun Microsystems provides post-condition checking, semantic assertions on interface attributes and operations. This work has been extended with the Borneo [San96, bor] project adding formal specification constructs to OMG IDL, expressing input-output behavior of objects, and describing relationships between events occurring in the distributed environment. Other approaches along these lines include [DW98, LC93, Pud98].

Several programming languages approaches to annotating interfaces with assertions to improve program correctness and interface expressiveness have been precursors of this work, most notably Eiffel [Mey97], Sather [D. 96], and JAWA [FM98], among many others.. An object oriented programming language that introduces behavioral subtyping is defined in [Ame91].

Altogether different proposals to combine behavioral annotations and IDL are presented in [BHW, AJ96]. Born *et al.* propose to map IDL into a formal description technique, in their case SDL-92 [ITU93], in order to combine the specification of signatures with the more expressive features of formal description languages. Further processing is carried out via tools supporting SDL–92. Anlauf and Jähnichen [AJ96], on the contrary, propose to specify interfaces in Object–Z to capture behavioral aspects. They map Object-Z interface specifications to IDL and support code. In both proposals it is not shown how the integration with state-of-the-art distributed computing environments may be carried out.

Expressing quality of service requirements

The annotation of interfaces with real time constraints (e.g., priorities, deadlines, execution time) and quality of service attributes (e.g., required min/max bandwidth, allowed jitter, resource needs) have been widely proposed including motivating implementation alternatives.

Schmidt *et al.* [SLM97a] propose to extend OMG IDL to express QoS annotations in their TAO real time ORB through added syntax. The problem with such an approach is the non-inter-

ORB portability of the application. A similar solution is presented by Becker and Geihs [uKG99] who annotate IDL with QoS characteristics. Similarly TINA ODL [Par98] provides many built-in features to address QoS processing needs.

Wolfe *et al.* [WBTK96] add timing constraints via IDL context fields and interface attributes. They rely on programming conventions to implement the timing annotations but do not need to extend IDL itself. However, they still requires to extend the IDL compiler to insert time management functions in stubs and skeletons.

A very exhaustive description language encompassing the specification of QoS requirements and resource needs, service usage patterns, physical resources used, and internal object state and structural information is presented by Zinky *et al.* [ZBS97].

Object interaction protocols

Object interaction protocols are IDL annotations that describe sequences of legal operation invocations between interfaces or for one interface. They provide hints to clients on "how" to use an interface and give rise to static and dynamic checking of a caller-callee interaction.

The annotation of IDL with path-expressions to express order of invocation requirements is presented in [Wat98]. Alternatives are the management of server execution with Petri-nets [GG97], the features of TINA ODL [Par98, BHW] to express order of execution, IPDL (Interaction Protocol Definition Language) [Buk96], and regular types for active objects [Nie95].

Synchronization constraints

Synchronization constraints are treated extensively throughout this work. Related work approaches include Frolund [Fro96] who has proposed a framework to extend active object based languages with synchronization primitives (mutual exclusion, capacity constraints, etc.). Quiros *et al.* [vdlHQuOM97] have demonstrated how to prevent inheritance anomalies in multi-threaded CORBA environments that arise from mutual exclusion primitives added directly by the developer to the object implementation. In previous work, we have proposed extending OMG IDL with synchronization constraints [JK98], which we refine in this paper.

Expressing needs of special application domains

IDL extensions for supporting diverse application domains have been addressed by the OMG, for instance, CDL (Component Definition Language) [OMG98a] to capture needs of the business domain. Moreover, TINA ODL [Par98] has been designed with an eye on telecommunication application needs. Other examples include workflow modeling, database schema and access specification, document management, and object constraint languages. These languages mostly constitute entire definition languages themselves.

Miscellaneous proposals

A last category of interface extensions that are hard to fit in the above classification scheme include IDL extensions to express object co-location constraints [HBSG98], coordination constraints [HBSG98], data parallelism [KG97], security annotations [HMdPS96], and component definition language extensions [MK98, OMG98b].

4 Annotating Object Interfaces with Synchronization Constraints

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. In this section we illustrate the enhancement of object interface descriptions with annotations of synchronization constraints.

An extensive discussion of the need for synchronization constraints revived with the advent of concurrent object-oriented languages in the early 80s (cf., e.g., [BY87]). A follow-on debate on inheritance anomalies [MY91] reflected a serious difficulty of language designers and users in combining inheritance and concurrency without requiring substantial redefinitions of inherited synchronization code. Such anomalies are alleviated in our approach as we exploit inheritance only at the specification level but do not inherit synchronization code at the implementation level.

In the following subsection we sketch a simple semantic model of non-sequential behavior that is just rich enough to formalize synchronization constraints and make them a declarative part of interface definitions. To be compliant with the CORBA standard, such constraints will occur as comments to the IDL specification. This will be illustrated with annotations added to the interface of a bounded buffer.

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 technique.

4.1 A Semantic Model of Concurrent Processes

To understand the dynamic behavior of distributed applications, we use partially ordered sets of events, which we call processes. The events of interest are derived from the operations declared in IDL object interfaces. The executions of an operation m provided by a server object are represented by instances of two distinct event types: m_s and m_t. Event m_s denotes the start of a specific execution, while m_t denotes the termination of an execution of m.

Synchronization constraints impose a partial ordering $\rightarrow$ on a countable subset E of the universe of events that can be derived from an object interface. For a finite set M of event types associated with an object interface, we use the labeling function α to map events in E into a given set A of event types. For two events e_1, e_2 the relation $e_1 \rightarrow e_2$ represents a causal ordering between e_1 and e_2 and also defines their precedence in time in the sense that e_1 occurs before e_2. Two different events d and e for which neither $d \rightarrow e$ nor $e \rightarrow d$ holds may occur concurrently.

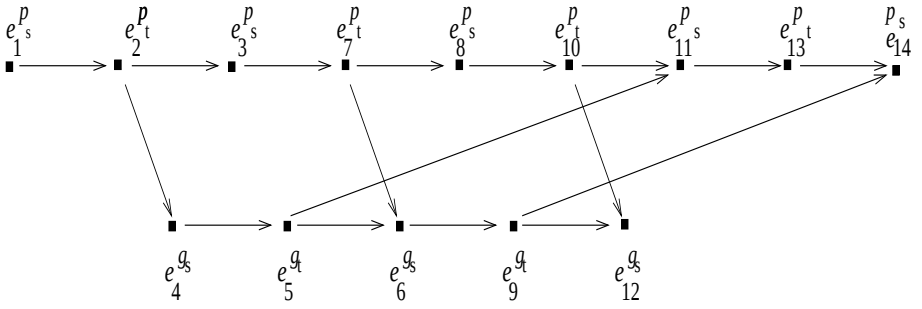


Figure 5: Graphical representation of a finite process

A finite process $p = (E_p, \rightarrow_p, \alpha_p)$ can be visualized through a directed acyclic graph whose nodes correspond to individual events and whose edges denote the causal relationship between events. For a given set $\{p, g\}$ of operations Fig. 5 shows a finite process in its graphical form for the bounded buffer interface of Fig. 2 with p_s, g_s denoting the start event types and p_t, g_t the termination event types derived from operations *put* and *get*, respectively. Such a process represents a specific protocol imposed on the operations at an object interface. The concrete

protocol depicted in Fig. 5 states that each *get* operation is preceded by at least one *put* operation, that all *put* (*get*) operations occur in sequence, and that the number of *put* operations must not exceed a certain limit as opposed to the number of get operations that occurred so far (this limit is 3 in our example).

Although such process graphs provide an intelligible model of the dynamic behavior of distributed systems, we cannot deduce general statements from a single observation as shown in Fig. 5. What we are after is a declarative approach to specify synchronization requirements as constraints imposed on the structure of processes. To achieve this goal, we rely on the definitions in [Bro94] and exploit the fact that each process is uniquely determined by the set of its finite prefixes. This allows us to characterize a process in terms of predicates stating properties that must hold for all the process' finite prefixes.

Following [Bro94] we use function "#" to map an event type a and a process p into the number of executions of a:

$$\#(a,p) = |\{e \in E_p | \alpha_p(e) = a\}|$$

This function, which can take the value infinite, provides the basis to define standard synchronization constraints such as

- **mutual exclusion**: an operation cannot be executed as it would interfere with another operation being processed,
- **self-exclusion:** an operation that can be executed by at most one thread at a time [Lop97], and
- **precedence**: operations must occur in a certain sequence, e.g., the microwave beam must be turned off before the door can be opened.

With the # function we can, for example, determine whether one or more execution instances of some operation m are currently active in a particular process p as follows:

$$active(m,p) := \forall q \mid q \text{ prefix } p \wedge q \text{ finite} \bullet \#(m_s,q) > \#(m_t,q).$$

Inactivity is equivalent to $\#(m_s,q) = \#(m_t,q)$ because m_t cannot be larger than m_s due to the standard operation execution semantics. The mutual exclusion of two operations m and n in an object interface can then be defined by:

$$mutex(m,n,p) := (active(m,p) \Rightarrow \neg active(n,p)) \wedge (active(n,p) \Rightarrow \neg active(m,p))$$

Function # can also be used to define predicates specifying safety properties, i.e., invariant constraints that must hold for all executions, and fairness requirements. Two examples of the former are capacity and in-bounds access constraints.

A nice feature of the #-function is that it can be easily implemented in terms of counter variables associated with interface operations and that the predicates defined over it boil down to arithmetic operations on these counter variables.

If we wanted to specify timing constraints as another aspect of object interfaces, we would have to add a function t mapping event types and processes into a domain $Time$ or into the set of natural numbers such that $t(e,p)$ would give us a discrete point in time at which e occurred in p. But this consideration goes beyond the scope of this paper and is therefore not further discussed.

4.2 Synchronization Constraints of a Bounded Buffer

A bounded buffer acting in a distributed environment gives rise to several kinds of synchronization constraints (for the sake of simplicity, the formal parameter p referring to the process under consideration is omitted):

- *mutex*(`get, get'`) with `get` $\neq$ `get'`,
  i.e., different invocations of the `get` operation must not be executed concurrently;
- *mutex*(`put, put'`);
- *capacity limitation*:
  $$\texttt{put}_s \leq (\texttt{get}_t + \texttt{bufsize})$$
  i.e., no `bufsize` more `put` than `get` invocations are allowed in any computation to avoid buffer overflow;
- *precedence constraint*:
  $$\texttt{get}_s \leq \texttt{put}_t$$
  i.e., 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.

```
interface BoundedBuffer {
  readonly attribute short bufsize;
  // the buffer takes at most 'bufsize' elements
  //--sc: dist(put,get,bufsize)

  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 6: IDL specification with synchronization annotations

In the implementation scheme to be developed in the following section, we want to associate synchronization code derived from constraint specifications with individual operation implementations. Therefore, we must decide what the semantics of constraints is with respect to operation execution. We adopt Frolund's approach and interpret synchronization specifications as negated guards [lun92]. For example, the intuitive meaning of the $mutex(m, n)$ constraint is that an operation invocation m is not executed if another invocation n is currently executed, and vice versa. 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.

Fig. 6 presents the IDL of the bounded buffer example annotated with synchronization annotations.

5 Design Patterns for Implementing Synchronization Code

In this section we present a design pattern with an embedded collection of alternative solutions implementing synchronization constraints. The synchronization code we automatically integrate

with the object implementation under development is synthesized from annotations. The strength of our approach is its reliance on standardized CORBA features only. We strictly avoided exploiting proprietary extensions.

5.1 Synchronization Constraints Pattern

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 presented in Section 4.1. To enable checks of synchronization constraints in the server implementation define two variables $\#start\_m$ and $\#terminate\_m$ (denoted by $\#end\_m$, for the sake of brevity) for each operation m involved in a synchronization constraint. Prior to activating the code implementing the functionality of operation m, verify one or more of the synchronization conditions depicted in Table 5.1 – depending on the constraint expression in which m occurs – and prevent the execution of m, while either of the corresponding conditions hold. Condition "alt", which is derived from "dist", specifies the alternating execution of two operations. This predicate can easily be generalized to the repeated sequence of more than two operations. Such sequence constraints are, for example, useful to handle safety constraints.

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

Sync. constraint	Sync. condition	Interpretation
mutex(m,n)	$\#start\_m - \#end\_n \neq 0$	n is active
dist(m,n,k)	$\#start\_m = \#end\_n + k$	capacity k is exhausted
dist(n,m,k)	$\#start\_m = \#end\_n$	m cannot be started more often than n has terminated
alt(m,n)	dist(m,n,1)	produces an alternating sequence of m and n beginning with m

Table 1: Synchronization constraints

implementing m's functional behavior is executed and finally variable $\#end\_m$ is incremented. This is illustrated in Figure 7 for some operation m with result type `<type>` defined in interface `<server>` with two synchronization constraints. Figure **??** shows the code patterns implementing the synchronization checks for single threaded and multi threaded servers respectively.
Benefits.

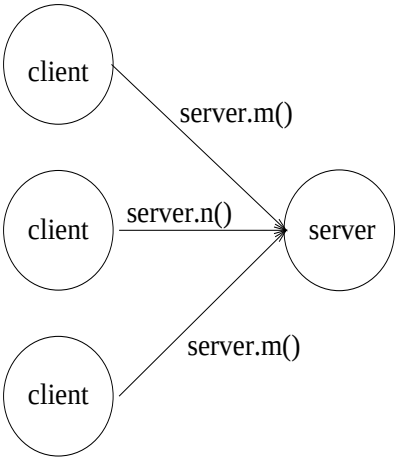
- Concurrent accesses to shared objects are properly synchronized.
- A wide range of causal dependencies can be specified and verified, solely by reference to variables maintaining operation state.
- The formal semantics underlying the synchronization constraints enables rigorous analysis at the specification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

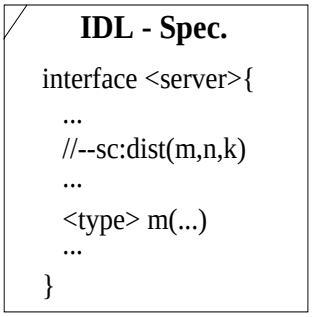
5.2 Development Steps and Pre-processing

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. 8 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.






```
// Multi threaded server
<type> <server>_sync::m(...)  {

  // begin critical section - acquire lock
  mutex( lock )

  // compute and check SC conditions
  while ( /* cond-1 or ... or
             cond-n are not satisfied */ )
    cond_wait( <CV>, lock );

  start_m++;
  try {
    <type> res = delegate->m(...);   }
  catch ( /* most specific handler */ ){
    // SC book keeping
    throw ( /* propagate exception */ )
  }
  ...
  catch ( /* least specific handler */ ){
    // SC book keeping
    throw ( /* propagate exception */ )
  }
  end_m++;

  // wake up waiting threads
  cond_broad( <CV> );

  // end critical section: let go of lock
  mutex_unlock( lock );
  return res;
}
```

```
// Single threaded server
<type> <server>_sync::m(...)  {

  // compute and check SC conditions
  if ( /* cond-1 not satisfied */ )
         throw SC_VIOLATED
  ...
  if ( /* cond-n not satisfied */ )
         throw SC_VIOLATED

  start_m++;
  try {
    <type> res = delegate->m(...); }
  catch ( /* most specific handler */ ){
    // SC book keeping
    throw ( /* propagate exception */ )
  }
  ...
  catch ( /* least specific handler */ ){
    // SC book keeping
    throw ( /* propagate exception */ )
  }
  end_m++;

  return res;
}
```

Figure 7: Multi threaded and singled threaded language patterns for synchronization condition checking code. (We assume a Solaris like threading model for the management of threads, i.e., for mutex and condition variables).

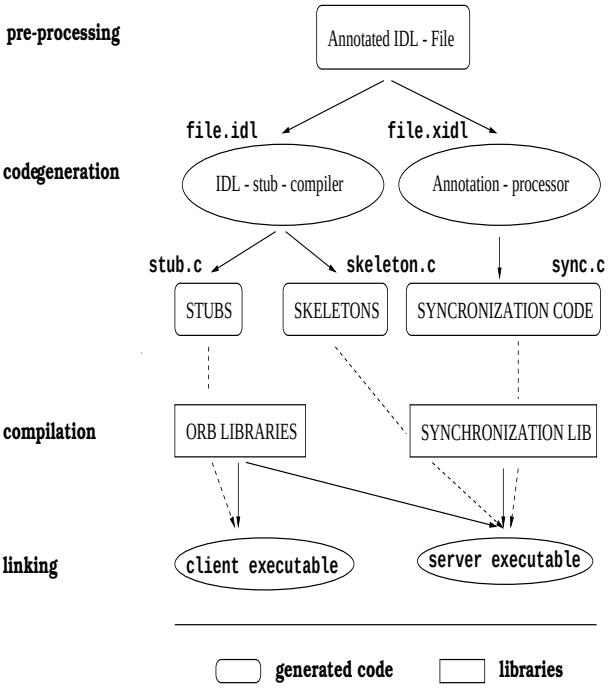


Figure 8: Development steps

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.

5.2.1 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. 9. The actual object implementation is 'plugged into' the platform by the newly generated adapter class which delegates invocations on its behalf.

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 Section 6.

5.2.2 Delegation-Based Solution

This solution builds on the TIE-approach for integrating an object implementation in a CORBA platform (see Fig. 4). It 'ties' the platform and the object implementation together. This is

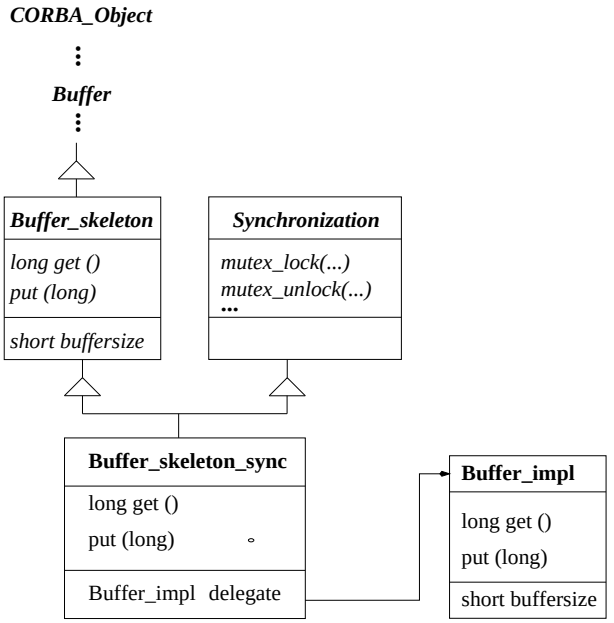


Figure 9: Modified inheritance structure to accommodate the class implementing the synchronization constraints for the inheritance-based design.

achieved by generating a class that delegates client method invocations to operations of the object implementation (cf. Fig. 10). This approach is particularly useful for integrating components written in programming languages not supporting inheritance.

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.

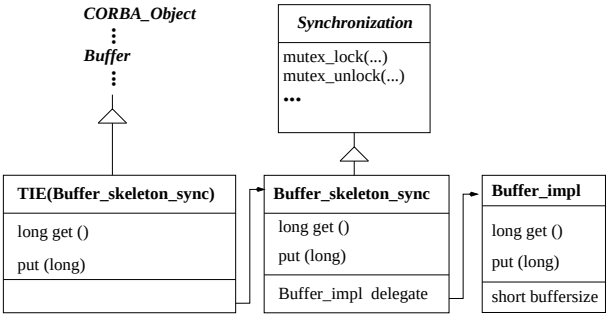


Figure 10: Modified Inheritance structure to accommodate the class implementing the synchronization constraints for the TIE-based design.

5.2.3 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 up-calls. 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].

5.2.4 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.

5.2.5 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. [Jac97b]), 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.

5.2.6 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.

6 Implementation

We have prototyped several of the above presented solutions for various ORB implementations (Mico, Orbix, and OrbAcus) to verify the portability claim of our solutions. We are also completing an annotation compiler for OMG IDL that synthesizes the adaptor classes from interface specification annotations for different CORBA products.

In this section we demonstrate how the code templates generated from the annotation compiler integrate with the skeletons generated by standard IDL compilers. The process is completely transparent to the application developer who simply annotates her IDL interface according to the application semantic. We then show how the developed design patterns generalize to the implementation of other IDL-extensions. We briefly point out limitations that arise due to the lack of the specification of crucial interfaces in the CORBA standard. Finally we give a comparative evaluation of the presented implementation alternatives.

6.1 Code templates

Besides discussing the generated code templates, we want to extend the example by two aspects commonly found in practice. These are exception handling and inheritance, as well as, their interaction with the defined synchronization constraints. We have slightly adapted the buffer interface from Figure 6 and introduced a user defined exception `ILLEGAL_ELEM` (cf. Figure 11). One interpretation of this could be that the buffer cannot hold all kind of elements from a particular type, such that the placement of an "illegal element" causes an exception to be raised by the buffer object. This, of course, is purely constructed to demonstrate the use of exceptions in synchronized objects. We have also derived the interface in Figure 11 from the base `Buffer` type to demonstrate issues arising due to inheritance.

```
// Buffer2.idl

interface Buffer2 : Buffer {

  exception  ILLEGAL_ELEM{};

  ...

  //--sc:dist(put, 2 * get2, size) -- overflow  'buffer full'
  //--sc:dist(get2, 2 * put, 0)    -- underflow 'buffer empty'

  long get2(out long element2) raises (ILLEGAL_ELEM);
  ...
}
```

Figure 11: Fragment of derived Buffer2 IDL interface annotated with a user defined exception.

Figure 12 shows the template of the generated code implementing synchronization constraints according to the inheritance–based approach discussed in Section 5.2.1. The example assumes a single threaded ORB. Synchronization condition violations are signaled through exceptions thrown by the synchronization adapter class.

For the sake of simplicity of the example we define one single generic exception signaling the erroneous condition (`SC_VIOLATED`). This exception is automatically generated. As a matter of fact, for each synchronization constraint a separate exception will be generated by the annotation

processor, including call progress information and debugging information, which is not shown in the examples. Each "synchronization exception" is associated with the class its constraint is implemented in (i.e., it is a nested class).

In the CORBA exception model user defined exceptions derive from `CORBA::UserException`. All system exceptions derive from `CORBA::SystemEception`. Catch-clauses are processed sequentially with the first matching clause handling the exception. Specific exceptions, such as `ILLEGAL_ELEM` (cf. 12), have to be handled first to ensure proper treatment.

Note, the necessary book-keeping operations in the exception handler. They ensure that after an erroneous condition was detected the synchronization state prior to executing the delegated call is re-instantiated before the exception is propagated back to the calling site, (i.e., back through the ORB to the client side.)

```
// buffer2_sync.cc -single threaded
#include "buffer2.h"       // stub compiler generated
#include "buffer2_impl.cc" // object implementation
#include "xidl.h"          // ORB specifics, macros ...

// ORBskeleton(Buffer2) -> e.g., 'virtual public Buffer2_skel' for OrbAcus
class Buffer2_sync_impl : ORBskeleton(Buffer2),
                          virtual public Buffer_sync_impl,
                          SYNCHRONIZATION                 {
private:
  Buffer_impl *delegate2;
  ...
public:
  //Constructor and destructor  ...

  class SC_VIOLATED : public CORBA::UserException { ... };

  CORBA::long
  put2( CORBA::Long elem ) throw ( Buffer2_sync_impl::SC_VIOLATED,
                                   ORBobject_impl(Buffer2)::ILLEGAL_ELEM ){
    //--sc: dist(put, 2 * get, size) -- overflow
    int tmp1 = ( (start_put - (2 * end_get) ) == this.size() );
    // sc:-guard, if violated throw an exception
    if ( tmp1 ) throw SC_VIOLATED;

    // other constraint checking goes here
    ...

    start_put++;
    // special handling for certain in, out, and inout arg.
    try{
      delegate2->put2(elem); }
    // exception handler:
    catch (ORBobject_impl(Buffer2)::ILLEGAL_ELEM &ie) {
       start_put--;   // SC book keeping
       throw (ie);   }
    catch (CORBA::SystemException &se){
       start_put--;  // SC book keeping
       throw (se);  }
    // delegated call returned successfully
    end_put++;
  };
  ...
};
```

Figure 12: Generate synchronization adapter for `Buffer2` interface. Note the macros `ORBbject_impl(...)` and `ORBskeleton(...)` to parametrize code for different CORBA implementations.

Figure 13 shows fragments of the `Buffer2` class generated by the IDL compiler and fragments of the `Buffer`-object implementation, which "wrap" the in Figure 12 shown synchronization adapter class.

```
//buffer.h - IDL-compiler generated class

class Buffer : virtual public CORBA_Object
{
  ...
  virtual void put(CORBA_Long element)

};

//buffer2.h - IDL-compiler generated class

class Buffer2 : virtual public Buffer,
                virtual public CORBA_Object
{
  ...
  virtual void put(CORBA_Long element)



};
```

```
//buffer_impl.cc - Buffer Object Implementation

class Buffer_impl {
public:
  // constructor - destructor
  Buffer_impl()  { ...};
  ...
  void put( CORBA::Long elem ) {...};
  CORBA::Long get() {...};
  ...
};

// buffer_skel.h - IDL compiler generated class
class Buffer_skel : virtual public Buffer,
                    virtual public CORBA_Object_skel
{...};

// buffer2_skel.h - IDL compiler generated class
class Buffer2_skel : virtual public Buffer2,
                     virtual public CORBA_Object_skel
{...};
```

Figure 13: IDL–compiler generated `Buffer` and `Buffer2` class and developer provided Buffer object implementation class (The Buffer2 object implementation class looks analogously.)

6.2 Generalization and limitations

The design patterns we have developed and motivated for implementing synchronization constraint extensions to OMG IDL may also be used to implement other IDL extensions. Examples include adding behavior and semantic, expressing quality of service requirements, defining interaction protocols, and several of the proposals listed in the miscellaneous category above (cf. Section 3). In general, it is possible to implement extensions that operate on methods and types defined within the extended interfaces through "before" and "after" processing steps wrapping the actual method invocation. These constitute extensions that do not require any additional state information not already expressed inside the interface, like ORB internal state information. For example, expressing real-time requirements (e.g., annotating an operation with a completion time deadline) may be realized by concurrently starting a timer with the operation invocation. Similarity, a scheduling algorithm may dynamically decide whether a method invocation may be started given the current system load.

However, certain limitations remain; especially, the latter real-time extensions will require threading support to be implemented in the described manner. To achieve this in a portable and non-platform specific way, CORBA standard support for threading is absolutely necessary. Moreover, extensions that would require access to ORB internal APIs cannot be implemented by third parties, since such APIs are not revealed in the CORBA standard. The emerging "Portable Interceptor RFP (request for proposal)" [OMG98c] addresses some of these issues and aims at providing means to "intercept" a method invocation at several points in the invocation chain. At interception points a user may insert application specific code to manipulate the method invocation (e.g, encrypt data stream, perform access control). This RFP does not solve the problem of accessing ORB internal APIs, it merely gives the developer more access points to

influence method processing within the ORB. The standard technology realizing this RFP is, at the time of this writing, still far from completed. It is therefore hardly possible to use the ORB APIs as a "compilation target" for an IDL processor (cf. similar problems for implementing language mappings [Jac97a]). Note, that the CORBA product always bundles the ORB and the IDL–compiler.

6.3 Evaluation

This section compares and evaluates the different design pattern solutions according to the following criteria: compliance, portability, implementation effort, client side extension, and server side extension.

Compliance describes whether or not a solution is based on CORBA standard features only, not requiring any additional ORB API support. *Portability* refers to the pattern implementation being portable from one CORBA product to the next. *Implementation effort* describes whether or not the approach requires one-time or per-ORB investment to be implemented. *Client* and *server side extension* capture the applicability of the solution to additions at client and server side, respectively.

	adaptor	DII/DSI	spec. OA	compiler	filter
compliance	yes	yes	no	no	no
portability	yes	yes	no	no	no
implementation effort	one–time: annotation compiler	one–time: annotation compiler	per–ORB: object adapter	per–ORB: IDL–compiler	per–ORB: support code
client side extension	no	yes	no	yes	yes
server side extension	yes	yes	yes	yes	yes

Table 2: Comparison of different implementation patterns.

A summary evaluation and comparison is depicted in Table 2. Some of the results listed there are still speculative as we have implemented prototype solutions only for a subset of the design alternatives discussed.

7 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.

Acknowledgments

Support from the German Research Society (DFG grant nos. SFB 373/A3 and GRK 316) and the German Minister of Research and Technology (grant CHN 178/95) is gratefully acknowledged. We would also like to thank Ji Zhang for implementing an early experimental tool based on the ideas presented in this paper to verify initial ideas. We would like to thank Rudolf Müller for giving us feedback on an earlier version of this manuscript. We are graceful for comments regarding interface definition languages extensions given to us by Stefan Tai and Gerald Weber.

References

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

[Ame91] P. America. Designing and object oriented programming language with behavioral subtyping. In Springer, editor, *REX School/Workshop LNCS 489*, 1991 1991.

[BHW] M. Born, M. Hoffmann, and M. Winkler. Evaluation and improvement of mapping rules from corba-idl and tina-odl to sdl-92. (GMD Focus Internal Report).

[bor] *The Borneo Language Reference DRAFT*. EMail:borneo-help@assp.eng.sun.com.

[BR98] Grady Booch and Ivar Jacobsonand James Rumbaugh. *The Unified Modeling Language User Guide*. The Addison-Wesley Object Technology Series. aw, 1998.

[Bro94] Manfred Broy. *Informatik: Eine grundlegende Einführung – Teil III*. Springer-Lehrbuch. Springer Verlag, 1994.

[Buk96] B. Bukowski. *IPDL - Interaction Protocol for Distributed Objects*. Freie University, Berlin, 1996.

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

[D. 96] D. P. Stoutamire and S. M. Omohundro. The Sather 1.1. Specification. Technical report, ICSI, 1996.

[DW98] D. D'Souza and A. Wills. OOA/D and CORBA/IDL: A common base. Technical report, ICON Computing Inc., 1998.

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

[FM98] C. Fischer and D. Meemken. Jawa: Java with assetions. In *Jit98*, pages 49 – 59, 1998.

[Fro96] S. Frolund. *Coordinating Distributed Objects*. MIT Press, 1996.

[GG97] H. Gruender and K. Geihs. On the object-oriented modelling of distributed workflow applications. In *3. Internationale Tagung Wirtschaftsinformatik 1997 (WI'97)*, Berlin, Germany, 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*, page April, 1998.

[HBSG98] O. Holder, L. Ben-Schaul, and H. Gazir. Dynamic layout of distributed applications in faro. Technical report, Technion – Israel institute of Technology, Aug 1998.

[HMdPS96] D. Hagimont, J. Mossire, X. Rousset de Pina, and F. Saunier. Hidden software capabilities. In *16th ICDCS Conference*, May 1996.

[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.

[ITU93] ITU. ITU recommendation Z.100 — CCITT specification and description language SDL. Technical report, CCITT, 1993.

[Jac97a] H.-A. Jacobsen. Programming language interoperability in distributed computing environments. In *DAIS'99*, Helsinki, Jnne 1997.

[Jac97b] H.-A. Jacobsen. Programming language transparency in distributed computing environments. In *OMG Workshop, Dublin, Ireland*, 1997.

[JK98] H.-A. Jacobsen and B. Krämer. A design pattern based approach to generating synchronization adaptors from annotated idl. In *IEEE Automated Software Engineering Conference (ASE'98)*. IEEE, 1998.

[KG97] K. Keahey and D. Gannon. Pardis: A parallel approach to CORBA. In *International Symposium on High Performance Distributed Computing*, pages 31–9. IEEE Comp. Society, Aug 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.

[LC93] G. T. Leavens and Y. Cheon. Extending corba-idl to specify behavior with larch. Technical report, IASTATE, 1993. ftp.cs.iastate.edu:pub/techreports/TR93-20/larch-corba.txt.

[Lop97] Christina Lopes. *D: A Language Framework for Distributed Programming.* PhD thesis, Northeastern University, 1997.

[LTdMK98] C. Lopes, B. Tekinerdogan, W. de Meuter, and G. Kiczales, editors. *Aspect-Oriented Programming Workshop at EXCOOP'98.* sv, 1998.

[lun92] S. Frølund. Inheritance of synchronisation constraints in concurrent object-oriented programming languages. In O. Lehrmann Madsen, editor, *ECOOP 92*, volume volume 615 of LNCS, pages 185–196. sv, 1992.

[Mey97] B. Meyer. *Object Oriented Software Construction.* ISE, 1997. (2nd editon).

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

[MK98] Jeff Magee and Jeff Kramer. *Composing Distributed Objects in CORBA*, volume B.J. Krämer and M.P. Papazoglou and H.-W. Schmidt, chapter 7. Research Studies Press (Wiley & Sons), 1998.

[MY91] Satoshi Matsuoka and Akinori Yonezawa. Analuysis of inheritance anomaly in object-oriented concurent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*. MIT Press, 1991.

[Nie95] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object Oriented Software Composition*, chapter 4, pages 99–121. Prentice Hall, 1995.

[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.

[OMG98a] OMG. Component definition language (cdl). Technical report, Object Management Group, 1998.

[OMG98b] OMG. Corba components rfp. Technical report, Object Management Group, 1998. http://www.omg.org/techprocess/meetings/schedule/CORBA_Component_Model_RFP.html.

[OMG98c] OMG. *Portable Interceptor RFP*. Object management Group (OMG), 1998.

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

[Par98] A. Parhar. Tina object definition language manual. Technical report, TINA Consortium, 1998. http://www.tinac.com/deliverable/deliverbal.html.

[Pud98] A. Puder. A declarative extension of idl-based type definitions within open distributed environments. In *International Conference on Object Oriented Information Systems*, 1998.

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

[San96] S. Sankar. Introducing formal methods to software engineering through OMG's corba environment and interface definition language. In *Springer-Verlag*, number 1101 in LNCS, pages 52–61, 1996.

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

[SH94] S. Sankar and R. Hayes. An interface definition language for specifying and testing software. *ACM Sigplan Notices*, 29(8), Aug 1994.

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

[SLM97a] D. Schmidt, D. Levin, and S. Mungee. The design of the TAO real time object request broker. *Computer Communication Journal*, 1997.

[SLM97b] Douglas C. Schmidt, David Levine, and Sumedh Mungee. The design of the tao real-time object request broker. *Computer Communications*, 1997.

[uKG99] C. Becker und K. Geihs. Generic qos specifications for corba. In *Kommunikation in Verteilten Systemen (KIVS'99)*, 1999.

[vdlHQuOM97] von de las Heras Quiros und Olmo Millan. inheritance anomaly in corba multithreaded environments. *Theory and Practice of Object Systems*, 3(1):45–54, 1997.

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

[Wat98] D. Watkins. Using interface definition languages to support path expressions and programming by contract. In *Tools 26*, 1998.

[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.

[X/O96] X/Open SUN Microsystems. *ADL Language Reference Manual (1.0 edition)*, 1996. www.sunlabs.com/research/adl.

[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.