Design Patterns for Synchronization Adapters of CORBA Objects

H.-Arno Jacobsen — Bernd J. Krämer

Humboldt University, Berlin
Institute of Information Systems
D–10178 Berlin, Germany
jacobsen@wiwi.hu-berlin.de

FernUniversität
Chair of Distributed Software Engineering
D-58084 Hagen, Germany
bernd.kraemer@fernuni-hagen.de

ABSTRACT. Standard middleware platforms offer Interface definition languages (IDLs) to achieve component interoperability in a heterogeneous computing context. IDLs serve to specify module and interface names, as well as operation signatures. The simplicity of IDLs ensures that they 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 further 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 IDLs 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. We demonstrate the effectiveness of our approach through an IDL-annotation compiler that synthesizes code portable across different CORBA implementations.

KEYWORDS: IDL, extended interface definition language, synchronization constraints, synchronization code pattern.
1. Introduction

Several distributed computing platforms have come to age over the past few years. The more common ones include CORBA [OMG91], DCOM [Mic96], Java RMI 1, and DCE [ROS 92]. They all aim at insulating distributed applications from the underlying proprietary infrastructure to achieve interoperability across disparate hardware platforms, network protocols, and operating systems. Each platform comes with an interface definition language (IDL) that 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 ASN.1 typically specify component interfaces in terms of module names, interface names, structured types, and operation signatures. A signature defines the operation name, return type, argument modes and types, and possibly an exception type.

This simplicity 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 [BOO 98], IDLs are expressively weak. The consequences 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 [SCH 98], quality of service [ZIN 97], and behavioral [ZAD 97] annotations into IDL;

– annotations invisible to the IDL compiler that impose synchronization constraints on the operations visible at object interfaces [KRÄ 98].

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 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) [HAR 97]. 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 [JAC 99].

To escape this trap, we developed a semantic model for handling synchronization constraints in object interfaces [KRÅ 98]. This model is briefly reviewed in Section 4.1. Based on this model a formal notation was defined that allows developers of CORBA object implementations to include synchronization annotations as comments in IDL interface definitions. In [JAC 98a] we presented a translation scheme that maps such annotations into code implementing corresponding sanity checks and a wrapper approach that integrates such code automatically with the developer’s object implementation and the skeleton code generated by the IDL compiler.

In this paper our approach will be further developed into a pattern including a discussion of alternative code integration schemes. We further explore this idea by developing 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 serve to empirically investigate their pros and cons. The general idea relates to the motivation behind aspect-oriented programming [LOP 98], 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 a 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

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 interface specification language with a C++–like syntax. 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.

2.2. Concurrency in CORBA

The CORBA standard does not foresee explicit support for concurrent method invocation in its design. Client–server communication is based either on synchronous RPC–style invocation, or on deferred asynchronous invocation. Deferred invocation allows a client to send off multiple requests in sequence, and subsequently poll for results. A one-way invocation scheme is additionally provided. It allows a client to send off a request and continue processing. No knowledge about the successful completion of the request is communicated back to the client. Transfer semantics of such invocations is best effort. An extension to the synchronous RPC–style invocation scheme has been proposed by Jacobsen and Weissman [JAC 98b]. The CORBA Messaging and CORBA Notification service also provide means for asynchronous communication. However, this mandates the use of these services.

2.3. 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 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 Figure 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. It is small enough to present as a whole and at the same time sufficient to demonstrate all the important aspects of our approach.

```idl
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 1** – IDL definition of a bounded buffer interface.

### 2.4. 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. This severely restricts the any attempt to extend standard IDL compilers (cf. [JAC 99]).

To describe the integration of object implementations and formalize design solutions implementing the proposed IDL extensions, we use a design–pattern notation [GAM 95]. In Figure 2 we use the UML 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

Most IDLs lack any semantic capabilities to express, for example, behavioral annotations, quality of service attributes, interaction protocols, or synchronization constraints.
Figur 2 – 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.

This deficiency has lead to many language extensions, including the following categories:

Expressing behavior and semantic: The annotation of IDL with behavioral extensions, such as pre- and post-conditions, invariants, abstract operation semantics, data integrity conditions, and Horn clauses (cf. [SAN 94, X/O96, SAN 96, LEA 93, PUD 98]).

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) (cf. [SCH 98, BEC 99, PAR 98, ZIN 97]).

Object interaction protocols: The description of sequences of legal operation invocations between interfaces or for one interface (cf. [WAT 98, GRU 97, PAR 98, BUK 96, NIE 95]). They provide hints to clients on “how” to use an interface and give rise to static and dynamic checking of a caller-callee interaction.

Synchronization constraints: Synchronization of accesses to shared system resources (cf. [FRØ 96, HER 97, JAC 98a]).

Miscellaneous: IDL extensions to express object co-location constraints [HOL 98], coordination constraints [HOL 98], data parallelism [KEA 97], component definition
language extensions [MAG 98], security annotations [HAG 96], and support for specific application domains. The latter include, for instance, the OMG CDL (Component Definition Language) to capture needs of business objects or TINA ODL [PAR 98] to specifically support telecommunication application needs.

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., [BRI 87]). A follow-on debate on inheritance anomalies [MAT 91] 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 in a process \( p \) 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: \( m_e \) and \( m_t \). Event \( m_e \) 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 \( \alpha \).
to map events in $E$ into a given set $A$ of event types. Function $\alpha$ serves to relate events observed in processes with the operations in object interfaces. In Figure 3 we denote the relation $\alpha(e) \equiv a$ by $e^a$. 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 (e.g., $e_3$ and $e_4$ or $e_7$ and $e_4$).

Figure 3. Graphical representation of a finite process

A finite process $p \equiv (E_p, \rightarrow_p, \alpha_p)$ 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 $\{p, q\}$ of operations Figure 3 depicts a finite process for the bounded buffer interface of Figure 1. $p_s$ and $q_b$ denote the start event types and $p_t, q_b$ 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 process in Figure 3 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 Figure 3. 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 [BRO 94] 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 [BRO 94] we use function "$\#$" to map an event type $\alpha$ and a process $p$ into the number of executions of $\alpha$:

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

2. A prefix is a subprocess $q = (E_q, \rightarrow_q, \alpha_q)$ of a process $p = (E_p, \rightarrow_p, \alpha_p)$ with $E_q \subset E_p$. $\alpha_q|E_q = \alpha_q$ and $\rightarrow_q : E_q \times E_q \rightarrow q$ which additionally satisfies the property that $E_q$ contains all predecessors of all events in $E_p$. 

8 L’objet. Volume 6 - n° 1/2000
This function, which can take the value infinite, provides the basis to define standard synchronization constraints such as \textbf{mutual exclusion}, i.e., an operation cannot be executed as it would interfere with another operation being processed, \textbf{self-exclusion}, i.e., an operation that can be executed by at most one thread at a time [LOP 97], and \textbf{precedence}, which requires that operations must occur in a certain sequence. For example, 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, i.e., started but were not terminated yet, in a particular process \( p \) as follows:

\[
\text{active}(m,p) = \forall q \mid q \text{ prefix } p \land q \text{ finite } \bullet \#(m_{st}q) > \#(m_{f}q).
\]

Inactivity is equivalent to \#(\( m_{st}q \)) = \#(\( m_{f}q \)) because the number of termination events \( m_{f} \) of \( m \) cannot be larger than the number of start event \( m_{st} \) of \( m \) 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:

\[
\text{mutex}(m,n,p) = (\text{active}(m,p) \Rightarrow \lnot \text{active}(n,p)) \land (\text{active}(n,p) \Rightarrow \lnot \text{active}(m,p)).
\]

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 by variables counting the begin and end of each operation execution. Associating two counter variables with each interface operation allows us to implement the constraint predicates defined over an individual operation in terms of arithmetic operations on these variables.

\subsection*{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):

\begin{itemize}
  \item \textbf{mutex(get, get')} and \textbf{mutex(put, put')} with \( \text{get} \neq \text{get}' \) and \( \text{put} \neq \text{put}' \), i.e., different invocations of the \textbf{get} (\textbf{put}) operation must not be executed concurrently;
  \item \((\#\text{put}_s - \#\text{get}_s) \leq \text{bufsize} \), a capacity limitation requiring that no \textbf{bufsize} more \textbf{put} than \textbf{get} invocations are allowed in any computation to avoid buffer overflow;
  \item \#\text{get}_s \leq \#\text{put}_s \), a precedence constraint, which requires that there must never be more executions of the \textbf{get} than there are executions of the \textbf{put} operation to prevent underflow.
\end{itemize}
The last two constraints can be combined in one, called \( \text{dist}(\text{put}, \text{get,bufsize}) \) to denote the synchronic distance between two operation executions. The semantics of \( \text{mutex}(\text{put}, \text{put}') \) relies on further information about the origin of an invocation, which is not made explicit here but allows us to distinguish two invocations of the same interface operation from different clients in the implementation of a constraint.

```plaintext
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,put) for get /= get'

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

Figure 4 –. 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 Frølund’s approach and interpret synchronization specifications as negated guards [FRØ 96]. For example, the intuitive meaning of the \( \text{mutex}(\text{msn}) \) constraint is that an operation invocation \( n \) is not executed if another invocation \( m \) 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.

Figure 4 presents the IDL of the bounded buffer example annotated with synchronization annotations introduced before.

5. Design Patterns for Implementing Synchronization Code

In this section we present the synchronization adapter design pattern for implementing synchronization constraints. We also analyze a set of alternative implementation choices for the pattern. The synchronization code generated from IDL annotations integrates with the code frames generated by standard IDL compilers and the object implementation of the application. The strength of our approach is its reliance on standardized features only. We strictly avoided exploiting proprietary extensions. To describe the design pattern we use a notation similar to the one introduced in [GAM 95] for presenting design patterns.
We now briefly motivate the different facets of the design patterns template according to Gamma et al. [GAM 95], with further details following in the next sections. The new design pattern is called synchronization adapter (name), as it allows to implement synchronization constraints for a given class, i.e., it adapts the interface of the given class to one supporting synchronization (intend). The pattern is part of the structural class pattern family (classification). Section 5.1 motivates the pattern, demonstrates its application in distributed object systems, and summarizes its benefits (motivation, applicability, and consequences). The class structure for alternatives implementations of the pattern are shown in Figure 7 and Figure 8, respectively (structure). Implementation issues are summarized in Section 5.2.1 and Section 5.2.2, respectively; code samples illustrating the patterns’ concrete implementations are provided in various figures in the following sections, illustrating different aspects of the design (implementation and sample code). The synchronization adapter pattern primarily aims at being a technique for the automated synthesis of code, but it may also be used to manually incorporate synchronization aspects into the distributed object system. The synchronization adapter relates to the proxy, bridge, and adapter patterns (related pattern).

5.1. The Synchronization Adapter Design Pattern

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 analysis 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,
Table 1 — The Synchronization constraint captures the specification level IDL annotation, whereas, the synchronization condition incarnates the implementation level enforcement of the constraint.

<table>
<thead>
<tr>
<th>Synchronization constraint</th>
<th>Synchronization condition</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex(m,n)</td>
<td>(#start_m - #end_n) ≠ 0</td>
<td>m is active</td>
</tr>
<tr>
<td>dist(m,n,k)</td>
<td>#start_m = (#end_n + k)</td>
<td>capacity k is exhausted</td>
</tr>
<tr>
<td>dist(n,m,k)</td>
<td>#start_m = #end_n</td>
<td>m cannot be started more often than n has terminated</td>
</tr>
<tr>
<td>alt(m,n)</td>
<td>dist(m,n,1)</td>
<td>produces an alternating sequence of m and n beginning with m</td>
</tr>
</tbody>
</table>

– 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.

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 [SCH 98].

To alleviate these problems we propose the following solution. First, in the interface specification phase, augment server interfaces with synchronization constraints. The synchronization constraint specifies the synchronization semantic of the object implementation (cf. Section 4.1 for the choices we offer).

To enable automated processing of synchronization constraints in the object 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 dispatching the 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. The synchronization condition, therefore, incarnates the implementation time condition that allows or disallows the execution of method m depending on the state the system is currently in.

Synchronization constraint "alt", which is derived from “dist” by setting alt(m,n) = dist(m,n,1), specifies the alternating execution of two operations (cf. Table 5.1). This predicate can easily be generalized to the repeated sequence of more than two operations (e.g., alt(m_1,...,m_n)). Such sequence constraints are, for example, useful to handle object interaction protocols (cf. Section 3).
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.

If a synchronization condition is violated the execution of method \( m \) is disabled. Either the execution of \( m \) is denied, or deferred to a later point in time when the synchronization condition may be satisfied again. In the former case an exception is raised by the object implementation (i.e., by the generated synchronization code) to signal the violation of the constraint to the client. The client has to take proper actions to process the exception. The client may, for instance, try to invoke \( m \) again at a later point. In the latter case the execution of \( m \) is deferred, i.e., re-scheduled to a later point by the object implementation (by the generated synchronization code in concert with the object broker.) This mode of operandi demands support by the broker, as the standard method invocation scheme is synchronous blocking (cf. Section 2.2), i.e., invocations cannot simply be deferred at the object implementation side, with standard broker support only. A common solution is the use of multi threading. An incoming request is associated with a separate thread and managed accordingly. As the CORBA standard is rather unspecific about threading and the handling of multiple concurrent request at the object implementation, it is difficult to develop a generic solution for this problem. Figure 5 shows the code patterns implementing the synchronization checks for single threaded and multi threaded servers, respectively. The single threaded solution based on denying the method execution constitutes a fully portable solution. The multi threaded solution depends on the thread support of the broker used. Many object brokers support multi threading, such that this implementation becomes possible.

This has the following 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 object implementation. The solutions we present aim at portability. Figure 6 provides an abstract view of the deve-
Figure 5: Multi-threaded and single-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).

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 (interface specification phase),
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

<table>
<thead>
<tr>
<th>Multi-threaded server</th>
<th>Single-threaded server</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Multi-threaded server</td>
<td></td>
</tr>
<tr>
<td>&lt;type&gt; &lt;server&gt;._sync: sm(...) {</td>
<td></td>
</tr>
<tr>
<td>// begin critical section - acquire lock</td>
<td></td>
</tr>
<tr>
<td>mutex( lock );</td>
<td></td>
</tr>
<tr>
<td>// compute and check SC conditions</td>
<td></td>
</tr>
<tr>
<td>while ( /* cond-1 or ... or cond-n are not satisfied */</td>
<td></td>
</tr>
<tr>
<td>cond_wait( &lt;ID&gt;, lock );</td>
<td></td>
</tr>
<tr>
<td>start_m++;</td>
<td></td>
</tr>
<tr>
<td>try {</td>
<td></td>
</tr>
<tr>
<td>&lt;type&gt; res = delegate-&gt;m(...);</td>
<td></td>
</tr>
<tr>
<td>catch ( /* most specific handler */ {</td>
<td></td>
</tr>
<tr>
<td>// SC book keeping</td>
<td></td>
</tr>
<tr>
<td>throw ( /* propagate exception */ )</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>catch ( /* least specific handler */ {</td>
<td></td>
</tr>
<tr>
<td>// SC book keeping</td>
<td></td>
</tr>
<tr>
<td>throw ( /* propagate exception */ )</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>end_m++;</td>
<td></td>
</tr>
<tr>
<td>// wake up waiting threads</td>
<td></td>
</tr>
<tr>
<td>cond_broadcast( &lt;ID&gt; );</td>
<td></td>
</tr>
<tr>
<td>// end critical section: let go of lock</td>
<td></td>
</tr>
<tr>
<td>mutex_unlock( lock );</td>
<td></td>
</tr>
<tr>
<td>return res;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

```cpp
// Single-threaded server
<type> <server>._sync: sm(...) { |
// compute and check SC conditions |
if ( /* cond-m not satisfied */ ) |
throw S(condition RELATED) |
...
if ( /* cond-n not satisfied */ ) |
throw S(condition RELATED) |
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; |
```
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.

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 Figure 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 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 Figure 2). 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. Figure 8). 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.
Synchronization Adapters

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 [GOK 98].

5.2.4. Miscellaneous Solutions

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.

Figure 8 – Modified Inheritance structure to accommodate the class implementing the synchronization constraints for the TIE-based design.
Other solutions could involve implementing a proper IDL compiler for managing the extended specification language (cf. [JAC 99] for problems with this approach due to the lack of openness of crucial ORB interfaces).

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. The CORBA standard has recently been augmented with a feature similar to filters, referred to as interceptor, that could – if available in an ORB – be used in this manner.

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 adapter 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 also introduce a new example to illustrate this approach. 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: Integrating adapter and skeleton classes

Besides discussing the generated code templates, we want to introduce a more practical example, further extended by two issues commonly found in practice. These are exception handling and interface inheritance, as well as, their interaction with the defined synchronization constraints. We study these issues on a bank account class hierarchy (cf. Figure 9). A base class supports an account balance attribute, with other operations not shown. A derived class — modeling a savings account — supports a withdraw and interest operations. The derived interface is visible to the client (e.g., an automated teller management system). An overdraft exception is raised if the savings account is overdrafted. The account management system runs every so often invoking the interest operation on the savings account. 3

3. This example has been constructed to make our point. However, we think it is sophisticated enough to bare real-world characteristics.
Access to the account has to be synchronized to protect shared data from corruption. The interest and the withdraw operation, for instance, have to be executed mutually exclusive.

```
//file: account.idl
interface Account {
  attribute float balance;
}...
```

```
//file: savaccount.idl
interface SavAccount : Account {
  exception OVER_DRAFT;
  //--sc:muter(withdraw,interest)
  void withdraw(float amount)
  raises (OVER_DRAFT)
  //--sc:muter(interest,withdraw)
  void interest Q;
}...
```

**Figure 9** –. Fragments of an `Account` and derived `SavAccount` IDL interface annotated with synchronization constraints and a user defined exception.

Figure 10 shows the template of the generated code implementing synchronization constraints according to the inheritance–based approach discussed in Section 5.2.1. Synchronization condition violations are signaled through exceptions raised by the synchronization adapter class.

The exception signaling a violation of a synchronization constraint is denoted in the example by `SC_VIOLATED`. This exception is automatically synthesized together with the adapter classes. To distinguish between different synchronization constraints, different exceptions are used, including call progress information and debugging information, not shown in the examples to simplify the presentation. Each constraint violation signaling exception is associated with the class its constraint is implemented in (i.e., it is a nested class). Note, that this exception is due to the processing of constraint violations (cf. Section 5.1), it has nothing to do with any exception defined by the object class implementer (e.g., `OVER_DRAFT` exception in the account example).

In the CORBA exception model user defined exceptions derive from `CORBA::UserException`. All system exceptions derive from `CORBA::SystemException`. Catch–clauses are processed sequentially with the first matching clause handling the exception. Specific exceptions, such as `OVER_DRAFT` (cf. Figure 10), 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.)

Figure 11 shows fragments of the `SavAccount` class generated by the IDL compiler and fragments of the `Buffer`–object implementation, which "wrap" the, in Figure 10 shown, synchronization adapter class.
Figure 10 – Generated synchronization adapter for the SavAccount interface. Not all operations are shown. Note, the macros ORBObject_impl(...) and ORBSkeleton(...) to parameterize code for different CORBA implementations. The implementation pattern combines multithreading with signaling constraint violations by raising exceptions.
6.2. Generalization

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. Similarly, a scheduling algorithm may dynamically decide whether a method invocation may be started given the current system load.

```c
//A.h - IDL-compiler generated class
class A
  : virtual public CORBA::Object{
    ...
  };

//SavA.h - generated class
class SavA
  : virtual public A,
    virtual public CORBA::Object{
    ...
    virtual void withdraw(float amount)
      throw (Sav::SOFEB::ABEND);
    virtual void interest (O;
  };

//impl.cc -
//Object Implementation by programmer
class A_impl :
  public:
    // constructor - destructor
    A_implO {};
    ...
};

// A_skel.h - IDL compiler generated class
class A_skel : virtual public A,
    virtual public CORBA::Object_skel{
    ...
};

// Sav_skel.h -
// generated class
class Sav_skel : virtual public Sav,
    virtual public CORBA::Object_skel{
    ...
};
```

Figure 11 – IDL-compiler generated classes and programmer provided object implementation class (The SavA object implementation class looks analogously. With A we abbreviate Account.)

// pre(deposit, amount > 0) 
void deposit (in float amount); 
// post(deposit, balance = old balance + amount)

Figure 12 – Interface annotated with pre and post conditions.

To demonstrate that our approach can also be used for realizing other IDL extensions, we now show how to enforce operation pre and post conditions. We introduce
a deposit(...) operation in the account example. It can be annotated with a precondition, stating that the entered amount must be positive, and a postcondition, expressing that the account balance changes by the amount deposited. This is depicted in Figure 12. Note, in this case, upon precondition violation, it does not make sense to re-schedule the operation (cf. synchronization constraints in multithreaded environments Section 5.1), rather the precondition violation should be signaled right away. The pre/post condition checking code may be implemented in a manner similar to the synchronization constraint enforcement. This is depicted in Figure 13.

```c
// in the adapter class, for deposit delegation

if ( /* pre condition is violated */ )
    // internal book keeping
    throw PRE_VIOLATED;

try{
    delegate->deposit(amount); }
catch (CORBA::SystemException &se){
    // internal book keeping
    throw (se); } // delegated call returned successfully

if ( /* post condition is violated */ )
    // internal book keeping
    throw POST_VIOLATED;
```

Figure 13 – Implementation pattern to check pre and post conditions.

### 6.3. Limitations

However, certain limitations remain; especially, the afore mentioned 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, standard CORBA 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 OMG "Portable Interceptor RFP (request for proposal)" [OMG 98] tried to address some of these issues and aimed 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 [JAC 99]). Note, that the CORBA product always bundles the ORB and the IDL–compiler.

6.4. 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.

<table>
<thead>
<tr>
<th></th>
<th>adapter</th>
<th>DI/DSI</th>
<th>spec. OA</th>
<th>compiler</th>
<th>filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>compliance</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>portability</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>client side extension</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>server side extension</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

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.

8. References


H.-Arno Jacobsen  Arno Jacobsen is currently involved in research on distributed database systems at INRIA-Rocquencourt in France. He received his doctorate from the Institute of Information Systems at Humboldt University in Berlin on "Distributed Infrastructure Support for Electronic Commerce Applications". His research interests include technical aspects, as well as information management aspects of distributed systems, electronic commerce applications, and middleware platforms. Mr. Jacobsen has worked on object-oriented programming languages design, compiler construction, and distributed computing at the International Computer Science Institute in Berkeley and at Lawrence Berkeley National Laboratory. He is actively involved in the Object Management Group (OMG), where he initiated and chairs the working group on "High Performance CORBA".

Bernd J. Kämmer  Bernd Krämer full professor and head of the Chair of Distributed Software Engineering of the FernUniversität in Hagen, Germany. He holds a doctorate in computer science from the Technical University of Berlin. He was an adjunct professor at the NPS in Monterey, California, a visiting professor at the Queensland University of Technology in Brisbane, at McGill University in Montreal, and at the University of California, Berkeley. His research interests include Web-based hypermedia systems, distributed systems management and engineering, and safety-related software. He has published extensively on these topics. He is Co-Editor in Chief of the new Software Engineering Journal SDPS.