# Making components contract aware

Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, Damien Watkins

# Making components contract aware

*Antoine Beugnard* ENST Bretagne

*Jean-Marc Jézéquel* Irisa/CNRS

*Noël Plouzeau* Irisa/University of Rennes

*Damien Watkins* Monash University

Before we can trust a component in mission-critical applications, we must be able to determine, reliably and in advance, how it will behave.

Components have long promised to neatly encapsulate data and programs into a box that operates predictably without requiring that users know the specifics of how it does so. Many advocates have predicted that components will bring about the bright future of large-scale software reuse, spawning a market for components usable with such mainstream software buses as the Common Object Request Broker Architecture (CORBA) and Distributed Component Object Model (DCOM). In the Windows world, at least, this prediction is becoming a reality.

Yet recent reports[1,2] indicate mixed results when using and reusing components in mission-critical settings. Such results raise disturbing questions. How can you trust a component? What if the component behaves unexpectedly, either because it is faulty or simply because you misused it?

Mission-critical systems cannot be rebooted as easily as a desktop computer. We thus need a way of determining beforehand whether we can use a given component within a certain context. Ideally, this information would take the form of a specification that tells us *what* the component does without entering into the details of *how*. Further, the specification should provide parameters against which the component can be verified and validated, thus providing a kind of contract between the component and its users.

In real life, contracts exist at different levels, from Jean-Jacques Rousseau's social contract to negotiable contracts, from various forms of license agreements to cash-and-carry. Likewise, we have identified four classes of contracts in the software component world: basic or syntactic, behavioral, synchronization, and quantitative.

Responsibility for contract management must be shared by both client and service provider. In those cases where the contracts are self-explanatory, assigning responsibility is easy. But some contracts rely on external constraints that too often remain implicit. In particular, quantitative contracts such as quality of service refer to features "outside the box," which reside in the system layer on which client and provider run.

In this article we define a general model of software contracts and show how existing mechanisms could be used to turn traditional components into contract-aware ones.

## FOUR LEVELS

When we apply contracts to components, we find that such contracts can be divided into four levels of increasingly negotiable properties, as Figure 1 shows. The first level, basic, or syntactic, contracts, is required simply to make the system work. The second level, behavioral contracts, improves the level of confidence in a sequential context. The third level, synchronization contracts, improves confidence in distributed or concurrency contexts. The fourth level, quality-of-service contracts, quantifies quality of service and is usually negotiable.

### Basic contracts

Interface definition languages (IDLs), as well as typed object-based or object-oriented languages, let the component designer specify

- the operations a component can perform,
- the input and output parameters each component requires, and
- the possible exceptions that might be raised during operation.

Static type checking verifies at compile time that all clients use the component interface properly, whereas dynamic type checking delays this verification until runtime.

Typically, a first-level contract would be implemented in an object bus such as CORBA. The IDL specification of the objects' interface ensures that client and server can communicate. Here, for example, is a possible interface for a component implementing a simple bank account management system:

```
Interface BankAccount {
  void deposit(in Money amount);
  void withdraw(in Money amount);
  Money balance();
  Money overdraftLimit();
  void setOverdraftLimit(in Money
    newLimit);
}
```

This definition states that you can make several operations on a bank account: deposit, withdraw, and set overdraft limit. Further, you can get specific information from the account, including its balance and overdraft limit. Both the Component Object Model (COM) and JavaBeans let you write similar descriptions, albeit with slightly different syntaxes.

## Behavioral contracts

Unfortunately, because the BankAccount specification does not define precisely the effect of operation executions, the user can only guess at their outcomes. A deposit would probably increase the balance, but what if the deposit amount is negative? What would happen to a withdrawal operation that exceeds the overdraft limit?

Drawing from abstract-data-type theory, we can specify an operation's behavior by using Boolean assertions, called pre- and postconditions, for each service offered, as well as for class invariants. First implemented under the name *design by contract* in the Eiffel language,[3] this approach has spread to several other programming languages (the Java extension iContract,[4] for instance) and can be found in the Unified Modeling Language as the Object Constraint Language (OCL).[5]

The example shown in Figure 2 more precisely defines our BankAccount interface's behavior. The assertions' syntax is based loosely on OCL; the operator @pre refers to the value its argument had when entering the method.

The design-by-contract[6] approach prompts developers to specify precisely every consistency condition that could go wrong, and to assign explicitly the responsibility of its enforcement to either the routine caller (the client) or the routine implementation (the contractor). A contract carries mutual obligations and benefits: The client should only call a contractor routine in a state where the class invariant and the precondition of the routine are respected. In return, the contractor promises that when the routine returns, the work specified in the postcondition will be done, and the class invariant will be respected. Depending on the strength of the postcondition, we can imagine negotiable contracts at this level. A client may, for example, require a prohibitively high-level guarantee that requires an expensive server-side check, but other clients—that may only need weak and cheap postconditions—could be implemented at a weaker level 2.

When using a third-party component, contracts offer a specification against which you can validate that component. Still, when working from the speci-

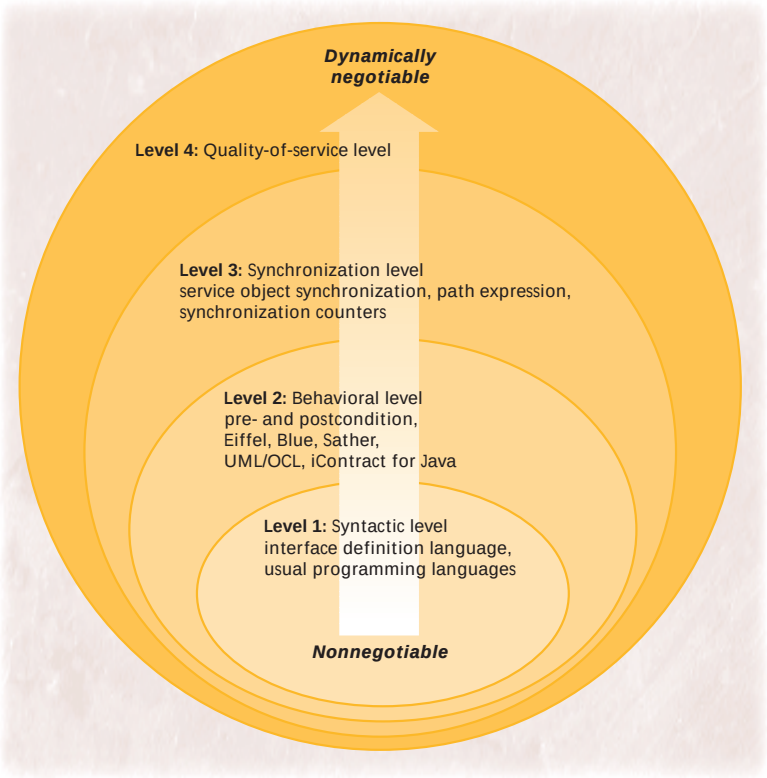


*Figure 1. The four contract levels, which start with basic syntactic nonnegotiable features and progress to include highly dynamic, negotiable, quantitative properties.*

```
Interface BankAccount {
  void deposit(in Money amount) {
    Require amount > 0;
    Ensure balance() = balance()@pre + amount;
  }
  void withdraw(in Money amount){
    Require amount > 0 and
      amount <= balance() + overdraftLimit();
    Ensure balance() = balance()@pre - amount;
  }
  Money balance();
  Money overdraftLimit();
  void setOverdraftLimit(in Money newLimit){
    Require balance() >= - newLimit
  }
  Invariant balance() >= - overdraftLimit()
}
```

*Figure 2. Defining the interface behavior for a simple bank account management system.*

**You can never be sure that a third-party component will perform correctly, only that it will perform as specified.**

fication alone you must bear in mind that you can never be sure the component will perform correctly, only that it will perform as specified.

### Synchronization contracts

The level 2 behavioral contract pretends that services are atomic or executed as transactions, which is not always practical or true. The level 3 synchronization contract specifies the global behavior of objects in terms of synchronizations between method calls. The aim of such a contract is to describe the dependencies between services provided by a component, such as sequence, parallelism, or shuffle. This kind of contract can be observed in entity structures or more formal applications.[7]

The contract reifies—makes concrete and specific—the ways in which the component serves its clients. A contract is useful to the functioning of a single client, but is significantly more important in an environment of one server and many clients. In such cases, the contract guarantees to each client that, whatever the other clients request, the requested service will be executed correctly. For example, we must know how our BankAccount would behave when a client requests a withdraw() while the BankAccount serves a setOverdraftLimit().

To express behavior in concurrent contexts, Ciaran McHale[7] proposed attaching to components special elements called synchronization policies. A subset of these synchronization policies can be described using the well-known path-expression formalism.[8]

Many strategies can be defined to manage intracomponent concurrency. For example, a Mutex strategy could force all services to be mutually exclusive, ensuring atomicity.

```
Synchronization Mutex(method a,b){
  // guard for a
  a : exec(a) == 0 and exec(b) == 0;
  // guard for b
  b : exec(a) == 0 and exec(b) == 0;}
```

Then our BankAccount component could use this Mutex policy between the methods withdraw() and setOverdraftLimit(), as follows:

```
Interface BankAccount
  use Mutex(withdraw,
     setOverdraftLimit) {
  // As before.
}
```

The synchronization contract is reified so that its binding to an object can differ from time to time, and so that various objects can share the same synchronization policy.

Java provides a stripped-down version of synchronization through the keyword "synchronized," which specifies that a given block or method should be run in mutual exclusion with other operations on the same object. While this method lacks the expressive power and versatility of McHale's synchronization policies, it is superior to implementing locks explicitly.

### Quality-of-service contracts

Now that we've expressed, qualified, and contractually defined all behavioral properties, we can quantify the expected behavior or offer the means to negotiate these values. You can specify the quality-of-service contract statically by enumerating the features the server objects will respect. Alternatively, you can implement a more dynamic solution that conducts negotiations between the object client and its server.

Common examples of quality-of-service parameters that may be exposed by a server are

- maximum response delay;
- average response;
- quality of the result, expressed as precision; and
- result throughput for multiobject answers such as data streams.

The main difficulty with satisfying such contracts is that they rely on third parties, which do not offer performance guarantees. Thus, these quantified contracts can be used only if their use is generalized. Several works deal with adding quality-of-service control to component-based software and propose extensions to CORBA with quality-of-service specifications.[9]

## CONTRACT MANAGEMENT

To a great extent, contract management issues are independent of the contract's level. We focus on the interactions that arise between a component and the client code that binds to the component's interfaces and requests services through them. Contract management deals with contract definition, subscription, application (including handling contract violations), termination, and deletion. Because contract and interface issues closely interlink, many aspects of contract management link to aspects of component management.

Several moments serve as milestones in a component's life: build time, load time, service exposition time, service request time, and unload time. Using the following implementation examples for each contract level—

- a plain CORBA IDL at level 1,
- Eiffel mechanisms for design by contracts at level 2,
- the Service Object Synchronization (SOS) mechanism at level 3, and

- TAO (the Adaptive Communication Environment object request broker)[10,11] and the Rusken computer-aided cooperative work environment[12] at level 4

—we sketch relationships between these particular moments using existing contract-implementation schemes.

### Contract definition and subscription

Although typical level 1 and 2 implementations bind these aspects so closely that they become indistinguishable, contract subscription and contract definition differ significantly at levels 3 and 4. At higher levels, defined contracts can be selected and tailored through subscription to accommodate dynamic operating conditions or client needs.

**Definition.** Contract *definition* is the feature set that fully describes the service offered by a component; *definition time* is the time a component makes the feature set available to customers as potential client code.

In level 1 components, such as exposing standard CORBA IDL definitions, contract definition time is the same as component-build time. The same relationship applies to level 2 contracts, which use Eiffel-like contracts: Preconditions, postconditions, and invariants are bound syntactically and semantically to method signatures.

Level 3 contracts, such as those provided by the SOS proposal, separate data properties (preconditions, postconditions, invariants) from control properties (invocation paths, concurrency). To retain as much flexibility as possible, this separation must be preserved in implementations.

Data and control constraints must, however, be merged before requests can be issued. This merge could take place at service exposition time, when a component exposes its interfaces. Choosing which control specification should be used with which data service is left to the configuration of the considered component.

Level 4 contracts also separate data constraints from control constraints. Stephane Lorcy and colleagues,[12] for example, reify contracts as instances from subclasses of the Contract class. This process completely separates the definition of contracts from that of the components that provide contract-aware services. The component thus controls contract definition time during the component's execution.

Level 4 contract reification allows dynamic adaptation of the contract to environmental changes, while level 3 contract reification enables a static choice under the service provider's responsibility. This static choice can be updated from time to time. Thus, a shared resource such as a printer can change its priority management policy from "first arrived, first served" to "largest job first." Such updating differs from level 4 contract negotiation in that it involves all clients simultaneously, while level 4 involves only a single client.

**Subscription.** We define contract *subscription* to mean the following:

- selection of a contract among a set of possible contracts,
- selection of parameter values when contracts have parameters, and
- agreement of both client and component parts regarding use of a contract in later service request execution.

Depending on the contract level and the typical implementations, contract subscription times differ broadly.

- Level 1 contracts are subscribed when the client code is bound to a component's interface definition.
- Level 2 Eiffel-like contracts are also bound at interface definition time.
- Level 3 SOS-like contracts may be bound at component load time or later, at interface definition time, depending on the component's behavior.
- Level 4 TAO- and Rusken-like contracts can be subscribed at any time prior to a service invocation, and can be changed or negotiated when the need arises.

Acquiring the ability to alter contracts in response to changing needs provided a strong reason for defining contracts as objects. Thus defined, contracts can be tailored at application design time by subclassing or aggregation: Several specialized subclasses of the Contract class provide a starting point for producing specific contracts for specific needs. Contracts can also be tailored at runtime during a negotiation phase.

Before a customer sends a request to a provider component, a contract object must be established between them. The customer selects a contract from the set exported by the customer, sets up parameters such as delays or other properties specific to the contract subclass chosen by the customer, then submits the contract for approval to the service provider. This provider then examines the desired terms of the contract and tunes them to make the request feasible. The contract is then returned to the customer for approval.

Should the customer reject the reconfigured contract, other contracts from other classes, such as those with weaker quality-of-service features, may be tried. The negotiation phase plays an important role in the Rusken scheme because quality of service involves both a customer (which has needs) and a provider (which has means): A scheme in which a customer

**Acquiring the ability to alter contracts in response to changing needs provided a strong reason for defining contracts as objects.**

**We must make components contract aware if we are serious about developing mission-critical applications based on third-party components.**

requires quality-of-service features blindly is not realistic in a world of components loaded with heterogeneity and instability. Further, a provider component also relies on other components as subcontractors and must negotiate subcontracts. Finally, components often include dynamic interface presentation and, thus, quality of service must also obey this property while taking advantage of it.

### Contract application

This aspect of contract management involves checking and handling violations.

**Checking contracts.** Provided there is enough theorem-proving machinery, some level 1 and 2 contracts could be checked statically, but most require some kind of runtime monitoring. This requirement forces us to decide if the check takes place on the client or server side. If dealing with static, third-party components, it is probably more efficient to monitor the contract on the client side. In a concurrent system, however, the component's state could change between the moment we do the check and the moment we call one of its methods. This suggests establishing a reified contract monitor somewhere in the component middleware. The monitor would then cooperate with the request broker, and should be sufficiently intelligent to optimize out some of the message passing when only one client is connected to a given component.[11]

**Handling violations.** When the system detects that a contract has been broken—say, for example, a client tries to withdraw too much money from a BankAccount file—an action can be undertaken. We have identified four possible actions:

1. *Ignore*. Proceed with the operation, ignoring any adverse effect. This is the effect of disabling the assertion checking in Eiffel, C++, and others.
2. *Reject*. In modern programming languages like Java, Eiffel, and C++, the reject action involves raising an exception and propagating it to the client. Older environments such as Unix C libraries returned a completion status to the client as the result of the "function call."
3. *Wait*. This action blocks the client call until the contract becomes valid through, for example, a wait for a deposit() or setOverdraftLimit() operation that could change the truth value of the withdraw() precondition. Clearly this option has no application to sequential programs, but it is the default behavior for so-called separate objects in Eiffel's concurrency extension proposal.[6]
4. *Negotiate*. Failure of the contract may be solved by retrying (in the case of a network failure) or renegotiating its terms.

### Contract termination and deletion

These activities share a highly symmetric relationship to contract definition and subscription. Level 1 and 2 contracts are *terminated* on service interface termination, which in turn occurs upon component unloading. Level 3 and 4 contracts, such as SOS and Rusken schemes, allow unilateral contract termination by a contract partner. Such terminations often occur to cope with changes in the environment, such as quality-of-service variations exported by the component's providers. Level 3 or level 4 contracts can be deleted by removing their definition from the component's interface.

## TOWARD A UNIFIED FRAMEWORK

We must make components contract aware if we are serious about developing mission-critical applications based on third-party components. Making contracts explicit improves confidence. The easiest way to achieve this goal is to design a new, component-oriented language that features in its semantics the various concepts we've outlined.

If we want the idea of contractible components to succeed, however, we must strive to integrate contract awareness concepts into popular component frameworks. Thus we propose building on existing technologies such as COM, CORBA, and JavaBeans by making contractible components available on demand for these platforms. This option means you pay for performance overhead only if you use the component.

Having described the many possible times at which contract definition, subscription, and application occur, we now elaborate on our BankAccount example to estimate how several of these techniques could be unified, and give some preliminary form for a contract-aware component technology.

For brevity's sake, we exclude schemes that bind contract subscription to service requests. We focus rather on binding service request times with contract subscription ones. Further, we base our examples on dynamic-interface-binding programming styles.

### Discovering contract-aware interfaces

Contractible components should export the basic interface-discovery interface ContractAwareComponent. This interface includes a QueryInterface() method for browsing contractible service interfaces.

```
interface ContractAwareComponent {
// Basic interface query system,
// with provision for contracts
CtServiceInterface QueryInterface(
String interfaceUniversalName);
}
```

The CtServiceInterface defines the interface base for contract-aware service interfaces. Hence all contract-

aware interfaces should derive from CtServiceInterface.

This interface also defines contract management operations and divides them into the categories previously described.

**Contract definition.** Once a client has a reference to an interface object with a CtServiceInterface, it can get the interface to export these contracts by using its QueryContract() method.

Contract objects contain all the information for level 2, 3, and 4 contracts managed by the service interface; level 1 contract information is available from the interface description itself.

```
interface Contract {
// Behavior and Synchronization
// Contracts
BehavioralContract
behaviorDescription();
SynchronizationContract
 synchroDescription();
// Returns true iff the contract has
// been changed by the component
boolean changed();
}
```

BehavioralContract and SynchronizationContract interfaces provide all level 2 and 3 contract information by any suitable means— for example, an XML-formatted description of the contracts expressed using UML and OCL formalisms. Although both types of contracts are usually nonnegotiable, the contract interface can be extended to handle negotiable quality-of-service issues, and level 4 contracts are often negotiable.

**Contract subscription.** Client code can fill in contract parameters provided by QueryContract(), then call Propose() to have the configured contract accepted by the component. If the contract configuration has been updated by the component to propose reasonable values, the method changed() returns a `true` result. For example, a component may adjust a service delay to some realistic value.

When the client code agrees to the last contract configuration, it can accept it by calling Accept() from the CtServiceInterface. The result value is the contract a customer must use in future request submissions. We supplied this provision because the provider may change the contract internally upon acceptance—to add, for example, implementation details used in future request invocation. Moreover, we need a rejection primitive to cope with the frequent case where a component uses subcontractors. In such a case, client code aborts the negotiation phase by calling Reject().

Thus, the CtServiceInterface takes the following form:

```
interface CtServiceInterface {
// Get a contract for a given
// service interface
Contract QueryContract();
void Propose(inout Contract c);
void Accept(inout Contract c);
void Reject(in Contract c);
```

**Contract application.** Once a contract has been subscribed, client code can issue contract-aware service requests. The component maintains a contract context together with the interface it provided as the return value of the QueryInterface() call. Hence no ancillary Contract parameter is necessary to request a service. Indeed, adding contract capabilities to some existing component does not imply alterations of service interface methods. However, we must provide some means to monitor the application of contracts. The predicate Failed() serves this purpose by returning a `true` result if and only if the last use of the subscribed contract failed. This style of explicit synchronous check can be enhanced to deal with implicit checks by, for example, using exceptions as provided by the environment.

```
// Returns true iff last service
// execution
// did not comply with
// contract terms.
boolean Failed();
```

**Contract termination.** Client code may terminate a contract subscription by calling the Terminate method from CtServiceInterface.

```
   void Terminate(Contract c);
}
```

### Using the BankAccount example

We now sketch out a possible contract definition for the BankAccount example, shown in Figure 3. Next, in Figure 4 we sketch some simple, Java-style code for getting a component and contract definition, subscribing to the contract, applying it by requesting a deposit, and finally terminating the contract.

```
interface BankAccount : CtServiceInterface {
void deposit(in Money amount);
void withdraw(in Money amount);
Money balance();
Money overdraftLimit();
void setOverdraftLimit(in Money newLimit);
}
interface BankAccountContract : Contract {
// Extend Contract with negotiable QoS contracts
double maxRequestDelay();
void setMaxRequestDelay(in double d);
}
```

*Figure 3. One possible contract definition for a simple bank account management system.*

```
BankAccountComponent component =
  Orb.GetComponent("Bank account manager");
BankAccount bankInterface =
  (BankAccount)
  component.QueryInterface("BankAccount");
BankAccountContract myContract =
  (BankAccountContract)
    bankInterface.QueryContract();
// Client code sets up maximum acceptable response
// time for all requests
myContract.setMaxRequestDelay(1000);
bankInterface.Propose(myContract);
if (! myContract.changed()) {
  // Component ready to accept contract with no
  // changes:
  // notify acceptance to component
  bankInterface.Accept(myContract);
  // Use contract (implicitly) by requesting a
  // deposit.
  bankInterface.deposit(500);
  if(bankInterface.Failed()) {
    System.println("Request execution" +
      "did not comply with contract terms");
    } else {
      System.println("Deposit succeeded");
    }
} else {
  // Don't accept the new terms of the contract
  bankInterface.Reject(myContract);
}
```

***Figure 4. An example of simple Java-style code for accessing, applying, and terminating a contract in a bank account management system.***

This example shows that contracts offer a powerful tool for enhancing the versatility and functionality of components.

> Reusing software components in mission-critical applications cannot succeed if the components do not export clearly stated service guarantees.

Indeed, a would-be component user cannot trust a component without strong statements from it. Several aspects of contractible-components technology have already been implemented in various projects.[9-12] We must now unify these implementations and generalize them to all standard component platforms so that mission-critical application designers can gain the full benefits of contractible components. This gain, however, will be effective only if developers consider their work as one link in a customer/provider chain. Links are hooked together at various levels with contracts. You build trust in the chain with contract-checking mechanisms. Although some techniques for checking contracts already exist, much research is still needed to deal with the palette of contracts we've outlined in this article. A key point would be the definition of a standard contract description language or notation mapped to the already existing contract techniques, as for the level 1 contract with IDL.

Classical component-based software such as network computer systems and interactive, distributed applications that run on the Internet are natural candidates for contract-based components. Any distributed software with soft real-time constraints, such as telecom software, would benefit from this approach. ❖

#### References


1. J.-M. Jézéquel and B. Meyer, "Design by Contract: The Lessons of Ariane," *Computer*, Jan. 1997, pp. 129-130.
2. E.J. Weyuker, "Testing Component-Based Software: A Cautionary Tale," *IEEE Software*, Sept. 1998, pp. 54-59.
3. B. Meyer, "Applying 'Design by Contract,'" *Computer*, Oct. 1992, pp. 40-52.
4. R. Kramer, "iContract—The Java Design by Contract Tool," *TOOLS 26: Technology of Object-Oriented Languages and Systems*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 295-307.
5. J. Warmer and A. Kleppe, *The Object Constraint Language*, Addison Wesley Longman, Reading, Mass., 1998.
6. B. Meyer, *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, Upper Saddle River, N.J., 1997.
7. C. McHale, *Synchronization in Concurrent, Object-Oriented Languages: Expressive Power, Genericity and Inheritance*, doctoral dissertation, Trinity College, Dept. Computer Science, Dublin, 1994.
8. R.H. Campbell and A.N. Habermann, "The Specification of Process Synchronizations by Path Expressions," *Lecture Notes in Computer Science*, E. Gelenbe and C. Kaiser, eds., *Vol. 16 Int'l Symp. Operating Systems*, Springer-Verlag, Berlin, 1974, pp. 89-102.
9. Z. Choukair and A. Beugnard, "Real-time Object-Oriented Distributed Processing with COREMO," *Object Oriented Technology: Ecoop 97 Workshop Reader*, J. Bosh and S. Mitchell, eds., Springer-Verlag, Berlin, 1998.
10. D.C. Schmidt, D.L. Levine, and S. Mungee, "The Design of the TAO Real-Time Object Request Broker," *IEEE Computer Comm. J.*, Vol. 21, No. 4, 1998, pp. 294-324.
11. D. Watkins, "Using Interface Definition Languages to Support Path Expressions and Programming by Contract," *TOOLS 26: Technology of Object-Oriented Languages and Systems*, IEEE CS Press, Los Alamitos, Calif., Aug. 1998, pp. 308-319.
12. S. Lorcy, N. Plouzeau, and J.-M. Jézéquel, "Reifying Quality of Service Contracts for Distributed Software," *TOOLS 26: Technology of Object-Oriented Languages and Systems*, IEEE CS Press, Los Alamitos, Calif., Aug. 1998, pp. 125-139.

*Antoine Beugnard is an assistant professor of computer science at Ecole Nationale Supérieure des Télécommunications de Bretagne, Brest, France. His main research activities deal with distributed object-oriented software architecture. Beugnard received a PhD in computer science from the University of Rennes.*

*Jean-Marc Jézéquel is a research manager at Irisa/CNRS. His research interests deal with object-oriented software engineering for distributed systems. He received a PhD in computer science from the University of Rennes, France.*

*Noël Plouzeau is an assistant professor of computer science at the University of Rennes, France. His main research activities deal with object-oriented and component-oriented software construction techniques, notably design of distributed and time-bound applications such as computer-supported cooperative work environments. Plouzeau received a PhD in computer science from the University of Rennes.*

*Damien Watkins is a PhD student at Monash University, Australia. He is interested in improving current component technologies in the context of distributed systems.*

*Contact the authors at antoine.beugnard@enst-bretagne.fr, {jezequel, plouzeau} @irisa.fr, damien.watkins@csse.monash.edu.au.*