A Framework for Arbitrary Invocation Semantics

Position Paper

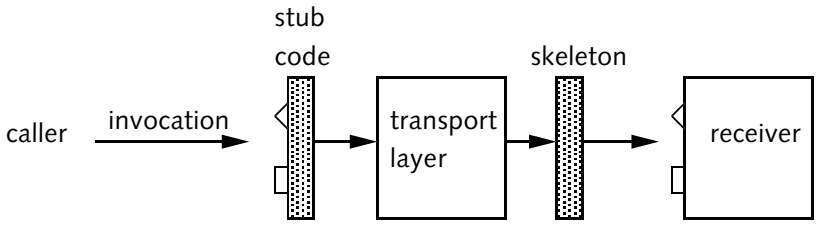
Markus Hof

Department of Computer Science (System Software)
Johannes Kepler University Linz, Austria
hof@ssw.uni-linz.ac.at

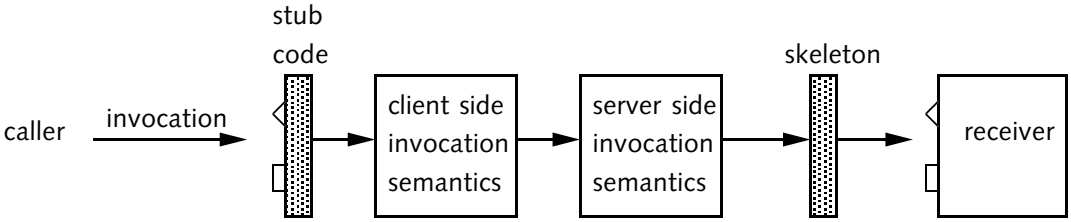
Abstract. Common object oriented systems have a fixed invocation semantic (synchronous procedure call). Some systems extend this concept with a fixed amount of additional semantics (RPC, best effort, asynchronous...). We claim that we have to go a step further and introduce invocation semantics as first class abstractions. We offer the possibility to write or compose completely new invocation semantics.
We implemented our ideas in a prototype with two concrete applications. First, we implemented a distributed object system, which introduces new semantics to forward invocations across a network. Second, we decorate local objects with additional semantics, e.g. transparent invocation logging, synchronization semantics...

1 Introduction

Common object oriented systems mostly have fixed invocation semantics. A method invocation is a synchronous procedure call. Distributed systems [RMI98, Corba] introduce a new semantic that uses custom code pieces in order to intercept and transport invocations in distributed environments.



However, this scheme is too restrictive. The programmer often needs different semantics depending on his current application's domain. E.g. asynchronous invocations, best-effort or at-most-once semantics... Programming environments that offer a fixed set of additional semantics [Voyager] are a first step in this direction. We claim that we need to go further, and therefore introduce a framework that allows creating arbitrary new invocation semantics.



Each method can have its own invocation semantic that consists of two parts: Client-side and server-side invocation semantics. Client side semantics are executed on the host of the stub object. Server side semantics on the host of the real object. This distinction has two advantages. First, the programmer is able to decide, individually for each invocation abstraction, where it is executed (on the client or on the server). Second, when several hosts use an object, a client side abstraction is executed only when the corresponding stub object received the invocation. A server side abstraction is invoked for every method invocation on this object, regardless of the host (stub) that started the invocation.
Our scheme is much more flexible in many aspects. It is also useful in non-distributed environments. As it is not necessary that the stub code resides on another machine, one can also decorate local objects. This offers a wide area of applications:

- Logging semantic transparently logs method invocations for later replay or visualization.
- Synchronization semantic introduces a flexible concept for arbitrary synchronization problems.
- Transaction semantic transparently ensures the ACID requirements during later method invocations.

We implemented our prototype in Oberon. Oberon offers type extension, polymorphism and dynamic binding. However the invocation semantics are fixed (local and synchronous). However, Oberon offers some introspection facilities which allow to view and change variables.

2 A library of invocation abstractions

The invocation semantics of common object oriented environments are fixed. Either the programmer is forced to use the only available semantics, or he can choose between a small fixed-sized set of semantics. Our invocation abstractions offer an open way to create arbitrary many new semantics.
The library consists of two parts. Both of them are presented in this section. The first part offers a meta-programming interface. It allows to browse class interfaces, as well as to view and to change their invocation properties. The second part allows the construction of new invocation abstractions either by writing new ones, or by combining existing abstractions.

2.1 Meta-Programming Information

Our meta-programming information offers a view on the capabilities of classes relevant for the invocation semantics. We introduce *Class* objects for individual instances. This object contains all relevant information on all methods (including inherited ones) needed in order to modify the invocation semantics. One may now scan this information or change it according to the current necessities. One can change the behaviour of method invocations in three different dimensions:

Set a parameter to shallow or deep copy mode
This setting influences the stub and skeleton code generation. It defines how the parameter is marshalled (this setting only affects parameter of a pointer type). In deep copy mode the parameter is copied (possibly recursively) onto the marshalled stream. The amount of marshalled data depends on the size of the actual passed data structure. In shallow copy mode, an object identifier (OID) is written to the marshalled stream. The invocation semantics do not generate OIDs. They have to be supplied by the current application.

Set the return value to shallow or deep copy mode
This setting defines the handling of the return value. The semantics are the same as described above for parameters, i.e. one can choose, whether the return value is deep or shallow copied. This mainly is of importance when using the invocation semantics in a distributed environment. Either the returned object is copied to the calling host, or the calling host receives a stub object and later method invocations are forwarded to the actual object.

Change the invocation semantics
Finally, this setting defines the invocation semantics individually for every method. This library actually implements no concrete invocation semantics. It just offers the basic methodology to define invocation semantics for individual objects.

2.2 Building and Using Invocation Abstractions

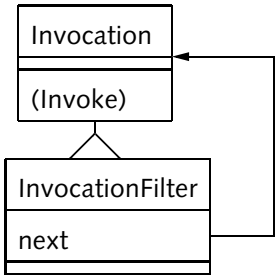
Building new invocation semantics or combining existing ones is quite simple. The library itself implements no concrete implementation, but only the abstract base classes. *Invocation* is the base class of all invocation abstractions. It defines only one method.

```
PROCEDURE (inv: Invocation) Invoke (obj: SYS.PTR; id: LONGINT; s: Stream): Stream;
```

This method has to be implemented by all concrete sub-classes. Whenever a method is invoked that uses an invocation abstraction, the corresponding method *Invoke* is called either directly by the stub code, or indirectly by another abstraction. When called, *Invoke* receives three parameters. *obj* is the receiver, *id* identifies the called method, and *s* is a stream containing the marshalled parameters. An actual implementation on this level is not possible, but depends on the actual application domain. Whatever the idea of a concrete subclass is, after the actual method has been invoked, the skeleton code returns the marshalled output (return value and output parameters) in another stream. This stream has to be returned to the caller.

```
PROCEDURE (inv: MyInvocation) Invoke (obj: SYS.PTR; id: LONGINT; s: Stream): Stream;
BEGIN
	DoSomething(obj, id, s);
	result := InvokeMethod(obj, id, s);
	DoSomething(obj, id, s, result);
	RETURN result
END Invoke;
```

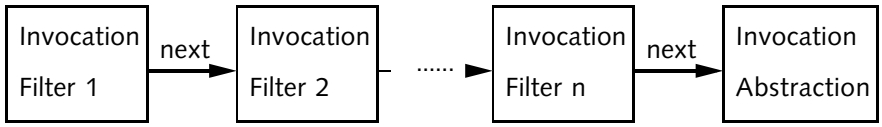
One subclass of *Invocation* is predefined. *InvocationFilter* allows decorating other invocation abstractions using the decorator pattern [Gamma95].



Invocation filters decorate actual invocation abstractions. Each filter has an attribute *next* which allows to chain arbitrary many filters to create new semantics.

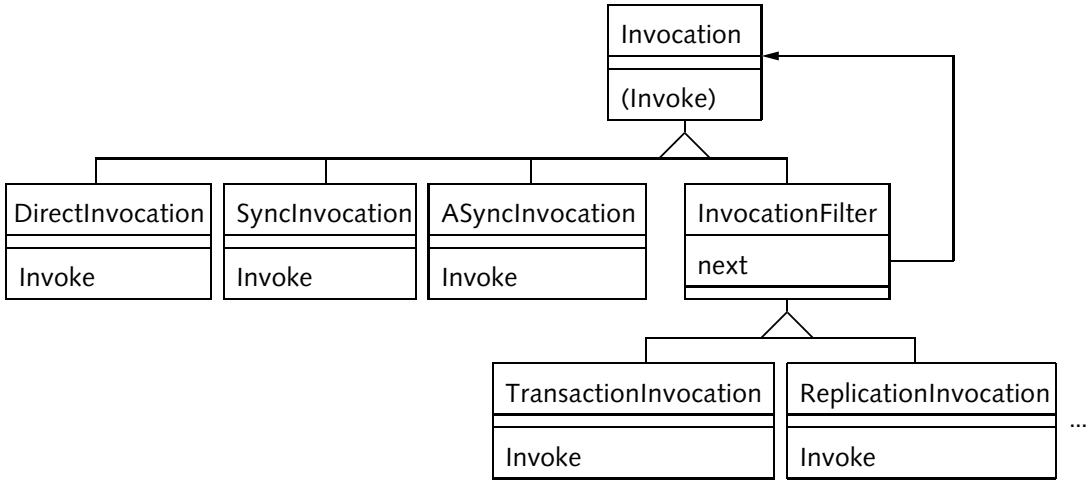
```
PROCEDURE (inv: MyInvocation) Invoke (obj: SYS.PTR; id: LONGINT; s: Stream): Stream;
BEGIN
	DoSomething(obj, id, s);
	result := inv.next.Invoke (obj, id, s);
	DoSomething(obj, id, s, result);
	RETURN result
END Invoke;
```

Therefore, invocation semantics consist of arbitrary many chained filters and one concrete invocation abstraction.

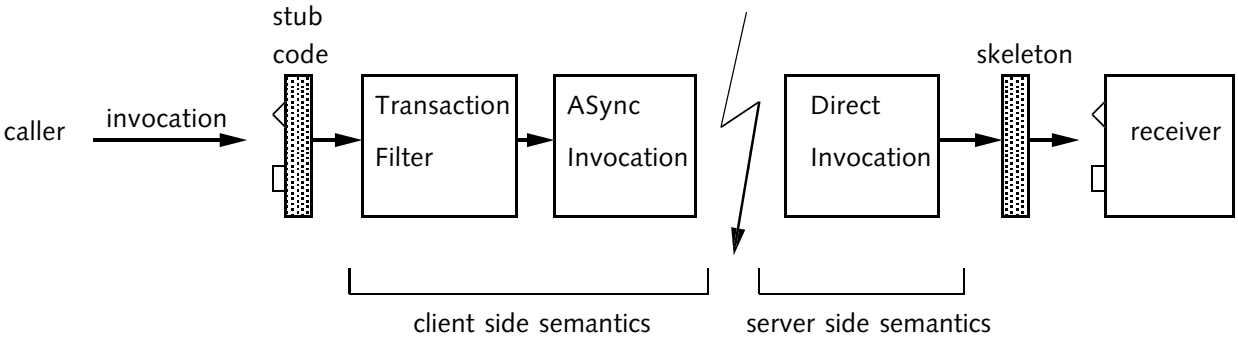


2.3 Concrete Invocation Abstractions

There is no limit to the number of possible invocation abstractions. In the figure below, we see a possible class hierarchy with some concrete sub classes.



We see three new invocation abstractions and two new decorators. The new abstractions are part of a package, which uses the invocation semantics to implement distributed objects. The two decorators are universally applicable. Each new abstraction or decorator may have new methods or attributes. E.g. *TransactionIvocation* has additional attributes necessary for transaction handling (logging, UNDO, REDO...). As an example, let's suppose one needs invocation semantics that is asynchronous and has transaction semantics. One has to combine the abstractions for the client and the server side.



To achieve this behaviour one has to generate the two semantics and assign them to the desired method(s) in the meta-class-information.

```
clientSemantics := TransactionInvocation(AsyncInvocation());
serverSemantics := DirectInvocation();
client := Invocations.GetClass(receiver);
server := Invocations.GetClass(receiver);
FOR mClient & mServer are all involved methods DO
        mClient.SetInvocation(clientSemantics);
        mServer.SetInvocation(serverSemantics)
END
```

To actually use these semantics, one has to assign them to an object. This is handled by the distributed object system. Later access would look similar to the following statements:

```
clientSemantics.StartTransaction();
obj.Method1(...);
....
obj.Methodn(...);
IF ok THEN clientSemantics.Committransaction()
ELSE clientSemantics.AbortTransaction()
END
```

By using the same instance of clientSemantics for different objects, the invocations on different objects are combined into one transaction.

3 Just-in-Time stub generation

With help of the library described in section 2, one composes the desired semantics and parameter passing modes. Tu put them to work one has to generate code that intercepts ordinary method invocations. This is done by cloning the actual object. Methods invoked on the actual object are still handled as before. Methods invoked on the clone object (stub object) are handled, as specified in the meta-class-information, by the generated stub code. It has the same interface as the actual method and replaces it transparently. It marshalls the parameter and activates the appropriate invocation semantic. Simultaneously, the skeleton code is generated. The skeleton is the counterpart to the stub. It unmarshalls the parameters and calls the actual method.
These code pieces are generated with help of the module *Objects*. It offers the necessary facilities to generate stub and skeleton code as well as the abstraction *DirectInvocation*, which invokes a method as done when no interception is used. These facilities are accessed through the following three procedures:

```
PROCEDURE Receiver (obj: SYS.PTR; sC, cC: Invocations.Class; VAR res: INTEGER);
PROCEDURE Intercept (obj: SYS.PTR; sC, cC: Invocations.Class; VAR res: INTEGER);
PROCEDURE Invoke (obj: SYS.PTR; id: LONGINT; s: Stream) : Stream;
```

Receiver is responsible for the server side. With help of the information contained in the passed meta-information, it generates the skeleton and sets up the necessary data structures. Afterwards, the passed object *obj* is able to receive method invocations, which were intercepted by the assigned semantics.

Intercept is the corresponding part on the client side. It generates the code needed by the stub object and sets up the client-side data structures. Afterwards, methods invoked on *obj* are handled as defined by the assigned semantics.

Finally, *Invoke* invokes a previously intercepted method. It chooses the assigned server-side semantic and executes it. In typical applications, one doesn't see this call, but will use a higher level abstraction, e.g. when using the

distributed objects package, it's the transport layer that calls *Invoke*. The application programmer sees nothing of these details.

One of the main goals was - from the beginning - that the delay, introduced by the increased flexibility of arbitrary invocation abstractions, is kept as small as possible. That means, that the delay is constant regarding to the number of managed objects and the number of different semantics. This is achieved by introducing an array for the invocation semantics. At run-time, each newly defined semantic is assigned a slot within this array, i.e. it receives a unique number. As the stub code is generated but afterwards, its possible to use this knowledge and access the correct semantics directly through an array access. At compilation time, this index is a constant. Therefore, the compiler can calculate the offset at compile-time. This reduces the actual overhead on the client side to a neglectable size.

For an example, let's look at the following class *Object*: It defines a single method *Dec*.

```
Object = POINTER TO ObjectDesc;
ObjectDesc = RECORD
   PROCEDURE (o: Object) Dec (n: INTEGER) : INTEGER;
END;
```

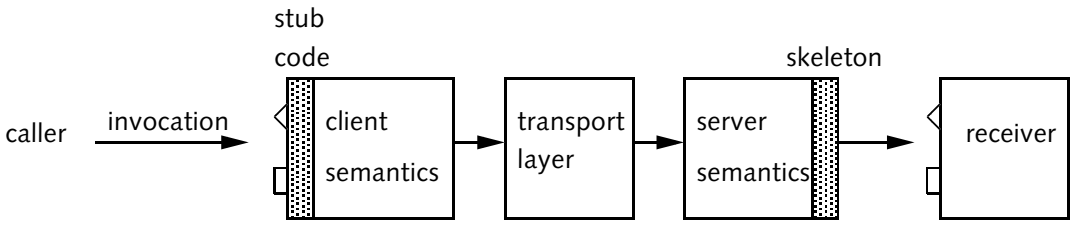
The generated code consists of two procedures: one for the stub object and one for the skeleton. The generated procedure *Dec* is used by the stub object to intercept invocations. It mostly handles the marshalling and unmarshalling of parameters. The actual invocation (in bold) is a simple method invocation. In this run, the chosen semantic is at index 10 of the semantics array. The offset is constant and is calculated by the compiler.

```
PROCEDURE (o: Object) Dec (n: INTEGER): INTEGER;
VAR lin: Linearizer; s: Stream; retVal: INTEGER;
BEGIN
   lin := Linearizers.NewWriter();
   lin.Integer(n);
   s := lin.Stream();
   s := Invocations.inv[10].Invoke(o, 2, s);
   lin := Linearizers.NewReader(s);
   lin.Integer(retVal);
   RETURN retVal
END Dec;

PROCEDURE DecSkeleton (receiver: Object; VAR stream: Stream);
VAR lin: Linearizer; retVal, n: INTEGER;
BEGIN
   lin := Linearizers.NewReader(stream);
   lin.Integer(n);
   retVal := receiver.Dec(n);
   lin := Linearizers.NewWriter();
   lin.Integer(retVal);
   stream := lin.Stream()
END DecSkeleton;
```

The corresponding Skeleton *DecSkeleton* has the inverse task. It unmarshalls the parameters, invokes the actual method, and marshalls the output values. The code, as shown above, is never generated. Our implementation generates directly compiler intermediate code, i.e. it builds an abstract syntax tree. This approach is portable, as we have a portable intermediate language with back-ends for individual platforms. This mechanism allows extremely fast code generation. The back-end generates the code pieces with the help of the meta-class information.

Up to now, we had no loss in speed. Even the amount of data transferred from client to server has not been increased. However, if we look at the complete picture, we see a point were additional work has to be done.



What is missing in our scheme so far is the transport layer. It's the means used to transport the information from the client to the server. This part is replaceable and depends only on the chosen semantics. In a distributed environment, the transport layer would actually send the data to another host. If only local decoration is used its the code piece which connects the stub object with the actual object. Whenever this layer transports an invocation to the server, it has to look for the correct server-side semantic. There are two possibilities on how this lookup can be done.

The first possibility uses a more or less advanced data structure to lookup the necessary semantics. The invocation supplies the receiver object, as well as a method identifier. With the help of the meta-class information one searches the appropriate semantic. The complexity of this data structure defines the amount of time spent searching. Our prototype implementation uses this approach. It uses simple linear lists, which imposes a linear dependency of the lookup time to the number of classes and the complexity of the called class.

A second possibility would be to supply the client with information about the exact location of the server semantic. The stub code would have to be modified to know both the client-side and server-side semantic locations. The server-side location would have to be forwarded to the transport layer. The layer doesn't have to search for the desired semantic anymore, which makes this alternative's time penalty constant. Unfortunately, this approach has two drawbacks. First, each invocation abstraction needs an additional parameter. Second, each method invocation's data increases by two bytes, which have to be transported from the client to the server.

4 References

[Corba] Object Management Architecture, http://www.omg.org/library/oma1.htm

[Gamma95] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns Elements of Reusable Object-Oriented Software, Addison Wesley, 1995

[RMI98] Remote Method Invocation specification, http://java.sun.com/producs/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html

[Voyager] Objectspace Voyager, http://www.objectspace.com/voyager