JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 1, January – February 2006

Call-out Bracket Methods in Timor

J. Leslie Keedy, Klaus Espenlaub, Christian Heinlein and **Gisela Menger**, University of Ulm, Germany

Abstract

This paper extends the concept of qualifying types by describing how their implementations can include not only bracket methods which are applied when a method of a target object is invoked, but also further "call-out" bracket methods which can be applied to invocations by the target object of the methods of other objects. This additional technique can be used for example to provide enhanced synchronisation in qualifying types, as an aid to confining the activities of an object, and as a means of providing parallel activities associated with the sending and receiving of information, e.g. encryption and decryption, data compression.

1 INTRODUCTION

In an earlier paper we have described the concept of qualifying types, i.e. types whose objects (known as "qualifiers") can dynamically qualify the behaviour of objects of other types (known as "targets") by means of bracket methods [5]. As presented in that and earlier papers [2-4], qualifying types permit modules to be written in the programming language Timor[1] which can, for example, provide such general services as synchronisation, protection and logging. The bracket methods which carry out such activities are applied to the *incoming* invocations of a target object's methods. A technique for statically incorporating qualifying types into other types was presented in [6].

The present paper extends the concept of qualifying types by describing *call-out bracket methods* as a new category of bracket methods. These function in a similar way to those of normal bracket methods with the difference that they are applied to the *outgoing* calls which an object might make to the methods of other objects. Call-out brackets can for example be used to release the semaphores acquired on entry to an object when this calls the methods of further objects. They can also be used to implement information confinement policies for objects or to journalise communications with other objects.

The bracketing of outgoing calls from a source object might also be coupled with a complementary bracketing of incoming calls to a target object to provide services which require parallel activities when passing and receiving information, such as encryption and decryption, data compression and expansion, etc.

[1] www.timor-programming.org

Cite this column as follows: J. Leslie Keedy, K. Espenlaub, C. Heinlein and G. Menger: "Call-out Bracket Methods in Timor", in *Journal of Object Technology*, vol. 5, no. 1, January-February 2006, pp. 51-67, http://www.jot.fm/issues/issue_2006_01/article1

It is assumed that readers are familiar with the basic idea of qualifiers [5, 6], so that we do not here repeat information about the structure of Timor nor how qualifiers work in general. In the current paper we refer to the bracket methods described in the earlier papers as *call-in bracket methods*.

2 CALL-OUT METHODS: AN OVERVIEW

As a first illustration of the idea behind call-out methods we define a type whose instances are able to qualify target objects such that whenever an instance method of the target is invoked mutual exclusion is guaranteed and whenever the target invokes any other object the mutual exclusion is released, but this is then claimed again as the outgoing call returns to the method of the target. Here is a type definition:

```
type ReleasableMutex {
qualifies any: // the call-in bracket methods
  op bracket all(...); // provides the synchronisation
  // needed to enter a target method
callout any: // the call-out bracket methods
  op bracket all(...); // provides the synchronisation needed
  // when the target invokes methods of any other object
}
```

and now an implementation:

```
impl ReleasableMutexImpl of ReleasableMutex {
state:
 Semaphore mutex = Semaphore.init(1);
qualifies any:
 op bracket all(...) {
  mutex.p();              // claims mutual exclusion on entry
  try {return body(...)}
  finally {mutex.v();}    // releases mutual excl. on return
 }
callout any:
 op bracket all(...) {
mutex.v(); // releases mutual excl.
           // on callout
try {return call(...)} // the call statement is
  // the callout equivalent to the body statement
  // in call-in brackets
  finally {mutex.p();}    // reclaims mutual excl. after call
 }
}
```

As this is not a paper on synchronisation we refrain from a discussion of the usefulness of such a module, except to point out the obvious: that a thread which uses such a qualifier has no guarantee that the state of the target will be the same before and after it calls out to another object.

From the viewpoint of the design of the Timor language we see that call-out methods are listed in a `callout` clause, which is similar to the `qualifies` clause for

call-in bracket methods (as described in [5]). In the above example all calls out of the target to any other object are handled by the same call-out code.

The definitions of the bracket methods for incoming and outgoing calls have the same syntax. The difference in their meaning is illustrated in Figure 1:

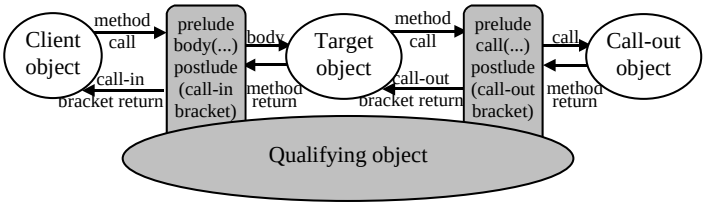


Figure 1: A Qualifier with Call-in and Call-out Bracket Methods

Call-in bracket methods "catch" a client's method invocation to the target (qualified) object, and begin executing their own code as defined in the `qualifies` section. If the call-in method executes a `body` statement the target method is then invoked. If the target itself then calls some other object (the "call-out" object), the qualifier's call-out bracket method(s) (defined in a `callout` section) "catch" this invocation on the way out. If the call-out bracket method then executes a `call` statement the call-out object is then called (or the next call-out bracket is activated). After the called object returns the call-out method continues execution at the statement following the `call` statement, and when this exits a return is made to the target object, or to the call-out bracket from which it was invoked.

A client object and a target object can of course both be qualified (usually by different qualifying objects, although we will see later that it sometimes makes sense for the same qualifier to qualify both). If a client object has a qualifier with call-out methods these methods are executed before the call-in methods of a qualifier associated with the target (cf. Figure 2):

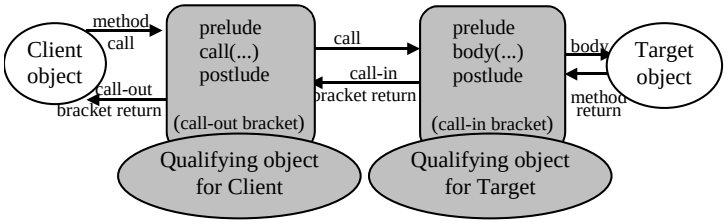


Figure 2: A Client with Call-out and a Target with Call-in Brackets

3 DEFINING CALL-OUTS FROM OBJECTS

Like call-in qualification, call-out qualification can be defined to apply either to `any` interface or to a specific view or type interface. However, whereas the counterpart of the bracket methods for call-in qualification are the methods of the object with which the qualifier is associated, the counterpart of the bracket methods for call-out qualification are the methods of other objects which it invokes. Since the type of an object is not normally the same as that of the objects which it invokes, there is no

implied type relationship between call-in and call-out types. Thus for example call-ins might be applied to a specific view or type while the call-outs might refer to any type, or a qualifying type might define only call-ins or call-outs.

An Example of Standard Call-ins and Call-outs: Monitoring

Following a `callout any` clause only *standard* call-out methods, i.e. methods defined to qualify `all` methods of the called object, or more specifically its `op` (writer) or `enq` (reader) methods, can be defined. Such qualification can be useful not only to support synchronisation (as illustrated in section 2) but also for example to monitor the activity of a target object, e.g. by maintaining counts of calls:

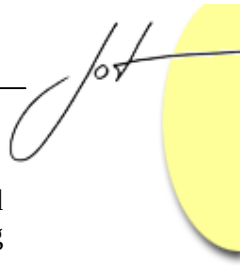
```
type CallCounting {
qualifies any:
 op bracket all(...);  // counts all calls to object
callout any:
 op bracket op(...);  // counts writer calls from object
 op bracket enq(...); // counts reader calls from object
instance:
 op int getAndResetCallsIn();
 op int getAndResetWriterOutCalls();
 op int getAndResetReaderOutCalls();
}
```

with an implementation:

```
impl CallCountingImpl of CallCounting {
state:
 int inCalls  = 0;
 int writerOutCalls  = 0;
 int readerOutCalls  = 0;
qualifies any:
 op bracket all(...) {inCalls++; return body(...);}
callout any:
 op bracket op(...) {writerOutCalls++; return call(...);}
 op bracket enq(...) {readerOutCalls++; return call(...);}
instance:
 op int getAndResetCallsIn() {
  try {return inCalls;} finally {inCalls = 0;}
 }
 op int getAndResetWriterOutCalls() {
  try {return writerOutCalls;} finally {writerOutCalls = 0;}
 }
 op int getAndResetReaderOutCalls() {
  try {return readerOutCalls;} finally {readerOutCalls = 0;}
 }
}
```

An Example of Standard Call-outs: Information Confinement

A qualifying type need not define call-in bracket methods. To illustrate how call-out methods alone can be usefully defined we consider how simple information confinement policies might be enforced, beginning with a type `Confined` which



ensures that its target object does not make any calls whatsoever to other objects (and therefore cannot release information, as Timor strictly enforces the information-hiding principle [8]):

```
type Confined {
callout any:
 enq bracket all(...) throws AccessViolation;
  // prevents all calls to objects of any type,
  // throwing an exception if a violation is detected
}
impl ConfinedImpl of Confined {
callout any:
 enq bracket all(...) {throw new AccessViolation.init();}
}
```

Although this type would be useful in many circumstances, it would often be desirable to enforce confinement policies which allow specific calls to be made. In this case the type `Confined` can be expanded, using a variant of normal inheritance which allows bracket methods to be inherited. In Timor the programmer has a choice between subtype polymorphic type inheritance (using the keyword `extends`) and interface inheritance without polymorphic implications (keyword `includes`). Here we choose the latter to define a qualifying type which only allows its target to make calls to the `print` method of a specific printer object. First we define a view which includes the permitted `print` method, then the qualifying type:

```
view Printer {
 op void print(Textfile* t);
}
type PrinterConfined {
includes: Confined;
maker:
 init(ObjId printerId); // the maker parameter specifies
 // the identifier of the printer to which calls are
 // permitted
callout Printer:
 enq void print(Textfile* t) throws AccessViolation;
  // this is a specific call-out bracket method, which
  // allows this call if the specified printer is invoked;
  // attempts to invoke a different printer object result in
  // an exception being thrown
 enq bracket all(...) throws AccessViolation;
  // prevents other Printer calls, throwing an exception
}
```

This type contains two `callout` sections, one of which is inherited from `Confined`. The new `callout` section, which is more specific and therefore has priority, defines what happens if an object containing the view `Printer` is called by the target. In this case the standard call-out bracket method (for `all`) is effective if the more specific method `print` has not been invoked. The overall effect is that only `print` calls to the correct printer are permitted. The inherited `callout` section (for `any`) defines what happens if objects of any (other) type are called, i.e. the object cannot make calls to objects other than the printer. Here is an implementation:

```
impl PrinterConfinedImpl of PrinterConfined {
state:
 ^Confined confined = Confined.init();
 // reuses the code of (any implementation of) Confined
 ObjId permittedPrinter;
maker:
 init(ObjId printerId) {permittedPrinter = printerId;}
callout Printer:
 enq void print(Textfile* t) {
  if (calledObject == permittedPrinter) call(...);
  else throw new AccessViolation.init();
 }
 enq bracket all(...) {throw new AccessViolation.init();}
}
```

The built in expression `calledObject` returns a value of type `ObjId`, which is a special type whose values are unique object identifiers. It can be used in both call-in and call-out bracket methods to determine which object is being called. Timor supports several such pseudo-identifiers which allow a bracket method to determine the environment in which it is working. However, the protection mechanisms provided by Timor are not the subject of this paper, and are therefore not described here in detail.

The effect of this qualifier is that the only call-out which a target object can make is a `print` call to a defined printer. Hence it is confined to using this printer. With call-out qualifiers in Timor any confinement policy known to us, including for example the Bell-LaPadula model [1], can be straightforwardly implemented.

Combining Call-in and Call-out Brackets: Communication Services

As we have seen in the examples of synchronisation (section 2) and monitoring (section 3) it is sometimes useful for a qualifying type to include call-in and call-out brackets which are designed to be applied to the same target object (cf. Figure 1). However, it can be equally useful in some cases to define a qualifying type which has call-out designed to operate in one target and call-in brackets designed to operate in another. Examples of this kind of communication service include compression or encryption of data in a call-out bracket of a sender object complemented by decompression or decryption in the call-in of the same call in a receiver object, cf. Figure 3.

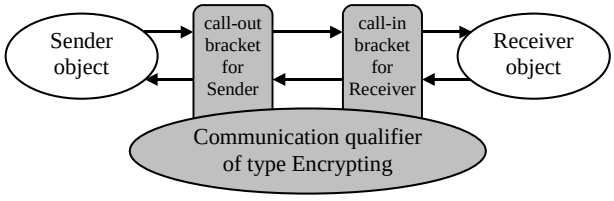
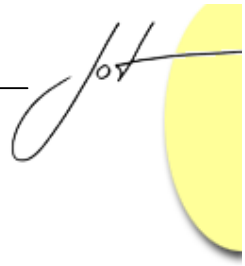


Figure 3: Providing a Communication Service

To illustrate this we define that text can be transmitted as a parameter to a method `transmit`, which is defined in a view that can be incorporated into many types.

```
view Transmission {
 op Text transmit(Text message);
```



```
 // the receiver responds using the returned Text value
}
type Encrypting {
maker:
 key(String messageKey, replyKey);
callout Transmission:
 enq Text transmit(Text message);
qualifies Transmission:
 enq Text transmit(Text message);
}
```

Here is a skeletal implementation:

```
impl EncryptingImpl of Encrypting {
state:
 String messageKey, replyKey;
maker:
 key(String key1, key2) {messageKey = key1;
                          replyKey = key2;}
callout Transmission: // associated with the sender
 enq void transmit(Text message) {
  // the callout bracket encrypts the parameter with the
  // message key before passing it on to the receiver
  Text reply = call(encrypt(message, messageKey));
  // when the call-out returns, the encrypted reply is de-
  // crypted using the reply key and returned to the sender
  return decrypt(reply, replyKey);
 }
qualifies Transmission: // associated with the receiver
 enq void transmit(Text message) {
  // before the receiver receives the message it is
  // decrypted using the message key
  Text reply = body(decrypt(message, messageKey));
  // the reply is encrypted using the reply key
  // and passed back
  return encrypt(reply, replyKey);
 }
instance:
 enq Text encrypt(Text aText, String aKey) {
  ... // encryption algorithm
 enq Text decrypt(Text aText, String aKey) {
  ... // decryption algorithm
 }
}
```

In the example it is assumed that the same encrypting object is used to bracket both the sender and the receiver. If the call is a local method call this could be useful, for example to hide the content of the message from later call-out and call-in brackets which might potentially contain trojan horses. In the case of a remote procedure call in a conventional environment, the desired effect could be achieved by having separate instances of the qualifier at the different locations, each initialised with the same keys and using the same algorithm. A symmetrical encryption algorithm is assumed, but alternative encryption qualifying types could be developed which use asymmetrical encryption algorithms.

4 USING CALL-IN AND CALL-OUT BRACKETS DYNAMICALLY

Qualifying types can either be associated with individual target objects dynamically, by placing them in a `List<:Qualifier*:>` [5] or bracketing can be defined statically in type definitions [6]. In this section we consider how the dynamic case is affected by call-out brackets.

Using Both Call-in and Call-out Brackets on a Target

In the normal case *all* the bracket methods (for call-in and for call-out) of a qualifier which appears in a `List<:Qualifier*:>` are applied to the target as appropriate. Thus an object can be instantiated as follows if it is to be exclusively synchronised when its methods are called and the synchronisation released when it makes nested calls (cf. section 2):

```
ReleasableMutex* releasingMutex =
        new ReleasableMutex.init();
Thing* syncThing = new {releasingMutex} Thing.init();
```

Using Call-out Brackets to Implement Pipes

In some cases qualifiers may only have call-out brackets, for example to transform the output of an object for use as the input of another object in a manner resembling Unix pipes. Suppose for example that the method `printOutput` of objects of type `Atype` sends a text file to a specific printer by invoking the latter's `print` method (described earlier in the view `Printer`). If the user wants to reorganise the output, e.g. by first sorting it, then truncating the first 50 lines, he might use two call-out qualifiers (of types `Sorter` and `Truncater`), which in their call-out brackets for `print` manipulate its `Textfile` parameter as appropriate. To set up the "pipe" he could use the following code:

```
Truncater* truncate = new Truncater.init(50);
Sorter* sort = new Sorter.init();
```

He could then create an object of type `Atype`, which is qualified by these qualifiers:

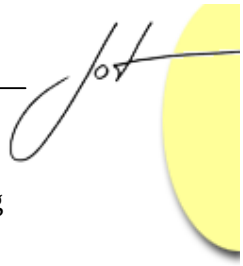
```
Atype* myObject = new {truncate,sort} Atype.init();
```

As will be explained in section 6, the order in which call-out brackets are executed is the reverse of their order in the qualifier list. To print the sorted and truncated text file he then simply invokes the `printOutput` method of `myObject`:

```
myObject.printOutput(thePrinter, theTextfile);
```

This example uses a list literal to associate the qualifiers with the object to be qualified. This is appropriate if `myObject` is used once only. However, if `myObject` is intended for frequent use with different qualifiers in a "pipe" then it could be set up as follows:

```
List<:Qualifier*:>* myList = {truncate,sort};
Atype* myObject = new myList Atype.init();
```



Subsequently `myList` could then be changed by inserting new qualifiers and removing existing ones, as described in [5].

The advantage of using call-out methods in conjunction with `myObject`, rather than similar call-in brackets of the `Printer` object invoked, is that each user can trim his own output as he wishes, without affecting other users of the `Printer` object.

Rules for Selecting Call-in and/or Call-out Brackets on an Individual Basis

As we saw in section 3, the call-in and call-out brackets of the same qualifier may need to be associated with different targets (e.g. a sender and a receiver) and it must therefore be clear what bracket methods should be applied to a target, the call-in methods, the call-out methods, or both. The technique adopted to achieve this in Timor is based on the idea that a reference for a qualifying type (i.e. a subtype of the type `Qualifier`) has two boolean values associated with it indicating whether the call-in and call-out brackets of the referenced qualifier are "active". This is the "normal" state of these indicators when a reference is declared and initialised to `null`, but bracket methods can be deactivated by using the arrow operator `->` in a reference expression. If the arrow appears before the reference expression then the call-out brackets are deactivated (and the call-in brackets remain active), while its use following the expression deactivates the call-in brackets (and indicates that the call-out brackets are active). The arrow operator can only be used once in a reference expression.

It is an error, detected at compile-time, if the arrow operator is used with any reference which is not for a qualifier, or for a qualifier reference which does not have the kind of brackets which are to remain activated. Once deactivated for a reference, bracket methods cannot be reactivated in association with the currently bound qualifier.

Assignment statements and parameter initialisations which have a reference with deactivated bracket category in the source reference result in the same restriction being set on the target reference, i.e. a restriction is not a permanent property of a reference as such, but only in so far as it is bound to a particular qualifier. Thus if an assignment statement has on the left side a reference with deactivated call-out brackets and the reference being assigned has deactivated call-in brackets, the result is that the reference on the left side becomes identical with that on the right side, without a run-time error occurring.

It is not considered to be an error to deactivate a bracket method category which has already been deactivated.

With this background, the Timor rules for dynamically selecting call brackets can now be formulated as follows:.

a) If a qualifier has both call-in and call-out brackets, but only its call-in bracket methods are to be applied to a particular target, the programmer deactivates the call-out brackets by prefixing the qualifier's reference with the operator `->`.
b) If a qualifier has both call-in and call-out brackets, but only its call-out bracket methods are to be applied to a particular target, the programmer deactivates the call-in brackets by placing the operator `->` after the qualifier's reference.
c) If the operator `->` is not used, *all* the bracket methods (call-in and/or call-out) associated with a qualifier in a `List<:Qualifier*:>` are applicable to a target.

To heighten the clarity of programs it is recommended that bracket methods are deactivated as they are inserted into qualifier lists.

Using the Rules for Communication Services

A user wishing to send an encrypted message (cf. section 3) could set up the sender and receiver objects as follows:

```
Encrypting* myEncryptor =
                  new Encrypting.key("Key1", "Key2");
Receiver* receiver =
                  new {->myEncryptor} Receiver.init();
Sender* sender =
                  new {myEncryptor->} Sender.init(receiver);
```

where the type `Sender` is defined as follows:

```
type Sender {
maker:
 init(Receiver* receiver);
instance: op void send(Text* aMessage);
}
```

with an implementation such as:

```
impl SenderImpl of Sender {
state:
 Receiver* receiver;
maker:
 init(Receiver* receiver) {this.receiver = receiver);}
instance:
 op void send(Text* aMessage) {receiver.transmit(aMessage);}
}
```

and the type `Receiver` includes the view `Transmission`.

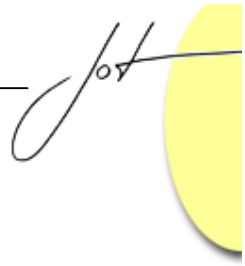
Using the Rules to Mask Out Call-in or Call-out Brackets

The above rules can be used not only to control communication services. They can also be used simply to mask out either the call-in or call-out brackets of a qualifier. For example a user wishing to monitor only the calls from an object using `CallCounting` (cf. section 3) might proceed as follows:

```
CallCounting* monitored = new CallCounting.init();
Thing* myObject = new {monitored->} Thing.init();
```

The arrow operator can also be used in a parameter to the insert method of `List`, but only in the context of insertion into a qualifier list, e.g.

```
List<:Qualifier:> myList = new List<:Qualifier:>.init();
myList.insert(monitored->);
Thing* myObject = new myList Thing.init();
```



5 STATIC TYPE DEFINITIONS WITH CALL-OUT BRACKETS

The syntax for declaring qualifiers statically for a new type was described in [6], a knowledge of which is assumed in this section. In the normal case, if a qualifier which has call-out brackets in its type definition appears in a static type declaration the call-out brackets are applied in an analogous way to their dynamic use. In this section the various possibilities, including restricting the use of call-in or call-out brackets are illustrated.

Static Call-in and Call-out Brackets for the Same Target

To define a `Thing` which is statically bracketed by the call-in and call-out brackets of `ReleasableMutex` is trivial:

```
type ReleasableMutexThing {
extends:
 {ReleasableMutex} Thing;
}
```

Static Call-out Brackets

Users can feel more confident that their information is secure if a print spooler is available which has been statically confined to accessing only a specified printer. For this purpose we assume the existence of a type `Spooler`, which extends the `Printer` view (i.e. has the method `print` as defined in section 3). In a new type `ConfinedSpooler` this could be statically confined by an instance of the type `PrinterConfined` (also defined in section 3) as follows:

```
type ConfinedSpooler {
extends:
 {PrinterConfined;} Spooler;
maker:
 init(ObjId thePrinter);
 // the parameter defines the permitted printer
 // for a particular spooler object
}
```

This might be implemented along the following lines:

```
impl ConfinedSpoolerImpl of ConfinedSpooler {
state:
 {^PrinterConfined confined;
  // reuses any implementation of PrinterConfined
 }
 ^Spooler spooler;
  // reuses any implementation of Spooler
maker:
 init(ObjId thePrinter) {
  confined.init(thePrinter);
  spooler.init();
 }
}
```

Masking out Brackets

Masking out either the call-in or call-out brackets statically (in the following example to apply only call-out brackets) can be achieved as follows (using a static version of the example in section 4):

```
type CallCountingThing {
extends:
 {CallCounting->;} Thing;
}
```

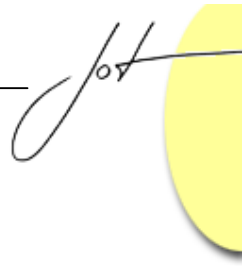
Here is an implementation:

```
impl CallCountingThingImpl of CallCountingThing {
state:
 {CallCounting-> callCounting = CallCounting.init();
 }
 Thing t = Thing.init();
}
```

In the case of static declarations the arrow operator is associated (as a prefix for call-in brackets, as a suffix for call-out brackets) with the *type name* of the qualifier in both type definitions and their implementations. This can only occur in a `qualifyingList` (see the EBNF syntax definition in section 3 of [6]). A compile time check is carried out to ensure that the type in question has bracket methods of the type indicated.

Notice that the use of an arrow in association with a type name is not to be understood as a property of the type being declared, but indicates merely that any qualifier being assigned (in an implementation) to the static qualifier will automatically have the corresponding bracket category deactivated (if it is not already deactivated, in the case of a reference).

As references for external qualifiers can be passed into the maker of a statically qualified type [6], the arrow operator can also be associated with a type name (with the same meaning) in the parameter list of such a maker, but only where the type definition contains a `qualifyingList` with an entry for the same type and with the same use of the operator. This rather unusual rule has two advantages. First, it allows the invoker of a maker to see the restriction without having to examine the code of an implementation. Second, it allows the condition to be checked at compile time (at the point in the client code where the maker is invoked).



Communication Services

We now define an encrypting communication service in which sender and receiver are statically bracketed. In this example we confine the sender to using the `Transmission` view (assuming that a type `TransmissionConfined` has been defined by analogy with `PrinterConfined` in section 3) and the receiver to printing out the message to a defined printer). Whereas the confinement bracketing can be defined "by value", the encryption is defined by reference cf. [6], allowing the sender and receiver to share the same qualifier. We begin with the sender:

```
type SecureSender {
extends:
 {TransmissionConfined; Encrypting*->;
 }
 Sender;
maker:
 init(Receiver* receiver; Encrypting*-> encrypting);
}
impl SecureSenderImpl of SecureSender {
state:
 {TransmissionConfined confined;
  ^Encrypting*-> encrypting;
 }
 ^Sender sender;
maker:
 init(Receiver* receiver; Encrypting* encrypting) {
  this.sender = Sender.init(receiver);
  this.encrypting = encrypting;
  this.confined = TransmissionConfined.init(receiver);
 }
}
```

The receiver definitions are as follows:

```
type SecureReceiver {
extends:
 {PrinterConfined; ->Encrypting*;} Receiver;
maker:
 init(->Encrypting* decrypting; ObjId aPrinter);
}
impl SecureReceiverImpl of SecureReceiver {
state:
 {PrinterConfined confined;
 ^->Encrypting* decrypting;
 }
 ^Receiver receiver;
maker:
 init(->Encrypting* decrypting) {
  this.decrypting = decrypting;
  this.confined = PrinterConfined.init(aPrinter);
 }
}
```

Other cases, such as the simulation of Unix pipes, present no problems in static definitions, although to define pipes statically would defeat their purpose from the viewpoint of flexibility.

6 SCHEDULING MULTIPLE QUALIFIERS

As was discussed in detail in [5], call-in bracket methods in a qualifier list are applied to the target object from left to right, i.e. the first qualifier in a qualifier list (or in the static case in a `qualifyingList` [6]) is applied, if it has a matching bracket method, then the second qualifier in the list, etc. In contrast, call-out qualifiers are applied in the reverse order of the list, i.e. from right to left. This has the effect that when a qualifier contains both kinds of bracket methods, these are nested as illustrated in Figure 4:

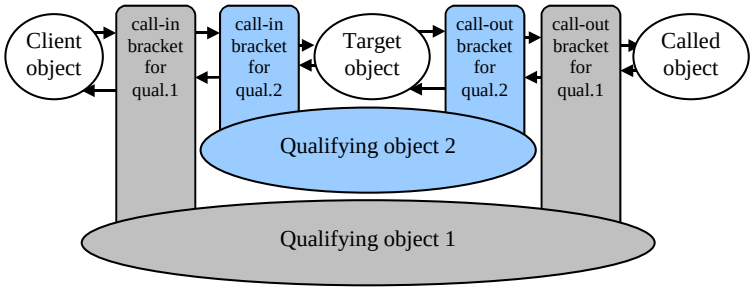


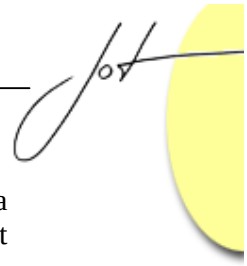
Figure 4: Applying Multiple Qualifiers

In this example the first entry in the qualifier list is the qualifying object 1, followed by the qualifying object 2.

7 RELATED WORK

The relation between qualifying types with call-in bracket methods and other work was discussed in detail in [5, 6]. Here it remains for us to discuss work related to call-out brackets. And here we have the problem that no such work is known to us.

What can be said however, is that there is clearly a close relationship between a call-out bracket associated with a client and a call-in bracket associated with its target. Thus it might appear that some of the examples in this paper can be handled by techniques which we have previously compared with call-in brackets. For example, combining call-in and call-out brackets for a single target (cf. `ReleasableMutex` in section 2) can be simulated using aspect oriented programming (AOP) techniques (cf. [7]). This is possible because AOP in effect allows program text to be cut and pasted into a class. Hence it can be effected at arbitrary points in a text, either where an object starts executing (cf. call-in) or where it calls another object (cf. call-out). But call-out brackets associated with one object (e.g. a sender) and call-in brackets associated with another object (e.g. a receiver) cannot be provided as a single AOP "module" in the way that this is possible in Timor. In addition, the other comments made in our comparison with AOP in [5] still apply, e.g. regarding the restrictions arising from (a) not distinguishing between `op` and `enq` methods, (b) operating at the source or



bytecode level and thus affecting all objects in a class, (c) not being able to associate a qualifier with a group of objects, (d) not treating qualifier methods as independent methods but as new methods of the class, etc.

A significant advantage of call-out methods is that they can be different for different callers of the same destination object. This is important for example in simulating Unix pipes. On the other hand it is sometimes desirable, as when confining a printer spooler, to ensure that the same call-out bracket methods (which guarantee the confinement) are statically defined and cannot be changed by different users.

8 CONCLUSION

The call-out bracket technique extends the notion of qualifying types as described in earlier papers (e.g. [4-6]) to allow the calls out of a target module to be bracketed in a similar manner to the calls into the target. This considerably extends the usability of qualifying types, for example by allowing more complex synchronisation and monitoring policies to be defined and implemented. It also opens new possibilities in the area of computer security, particularly in providing a technique which allows such protection features as information confinement policies and modular encryption techniques to be implemented at the programming language level[2]. It also introduces the possibility of providing a flexible Unix-like pipe mechanism at the programming language level.

ACKNOWLEDGEMENTS

Special thanks are due to Dr. Mark Evered and Dr. Axel Schmolitzky for their invaluable contributions to discussions of Timor and to the ideas which have been taken over from earlier projects. Without their ideas and comments Timor would not have been possible.

[2] This parallels our work in the operating system area, where the SPEEDOS operating system (cf. www.speedos-security.org) supports the notion of bracket methods for call-ins and call-outs at the system level.

REFERENCES

[1] D. E. Bell and L. J. LaPadula, "Secure Computer Systems: Mathematical Foundations," Mitre Corp., Bedford, Ma. ESD-TR-73-278, 1973.

[2] J. L. Keedy, M. Evered, A. Schmolitzky, and G. Menger, "Attribute Types and Bracket Implementations," 25th International Conference on Technology of Object-Oriented Languages and Systems, Melbourne, 1997, pp. 325-338.

[3] J. L. Keedy, K. Espenlaub, G. Menger, A. Schmolitzky, and M. Evered, "Software Reuse in an Object Oriented Framework: Distinguishing Types from Implementations and Objects from Attributes," 6th International Conference on Software Reuse, Vienna, 2000, pp. 420-435.

[4] J. L. Keedy, G. Menger, C. Heinlein, and F. Henskens, "Qualifying Types Illustrated by Synchronisation Examples," in *Objects, Components, Architectures, Services and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany*, vol. LNCS 2591, M. Aksit, M. Mezini, and R. Unland, Eds.: Springer, 2003, pp. 330-344, http://link.springer.de/link/service/series/0558/papers/2591/25910330.pdf.

[5] J. L. Keedy, K. Espenlaub, G. Menger, and C. Heinlein, "Qualifying Types with Bracket Methods in Timor," *Journal of Object Technology*, vol. 3, no. 1, pp. 101-121, http://www.jot.fm/issues/issue_2004_01/article1, 2004.

[6] J. L. Keedy, K. Espenlaub, G. Menger, C. Heinlein, and M. Evered, "Statically Qualified Types in Timor," *Journal of Object Technology*, vol. 4, no. 7, pp. 115-137 http://www.jot.fm/issues/issue_2005_9/article5, 2005.

[7] G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," ECOOP 2001 - Object-Oriented Programming, 2001, Springer Verlag, LNCS, vol. 2072, pp. 327-353.

[8] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," Communications of the ACM, vol. 15, no. 12, pp. 1053-1058, 1972.

About the authors

J. Leslie Keedy recently retired from the position of Professor and Head, Department of Computer Structures, University of Ulm, Germany, where he lead the Timor language design and the Speedos operating system design groups. His email address is keedy@jlkeedy.net. His biography can be visited at http://www.jlkeedy.net/biography_short.php

Klaus Espenlaub completed his Ph.D. in Computer Science at the University of Ulm in 2005. Currently he works as a research assistant in the Department of Computer Structures at the University of Ulm. His research interests include secure operating systems, protection mechanisms and computer architecture. His email address is espenlaub@informatik.uni-ulm.de.

Christian Heinlein received a Ph.D. in Computer Science from the University of Ulm in 2000. Currently, he works as a scientific assistant in the Department of Computer Structures at the University of Ulm. His research interests include programming language design in general, especially genericity, extensibility and non-standard type systems. His email address is heinlein@informatik.uni-ulm.de.

Gisela Menger received a Ph.D. in Computer Science from the University of Ulm in 2000. Currently she works as a scientific assistant in the Department of Computer Structures at the University of Ulm. Her research interests include programming language design and software engineering. Her email address is menger@informatik.uni-ulm.de.