Reasoning About Object Clients and Distributed Object Interaction

Anna Mikhajlova

Turku Centre for Computer Science, Åbo Akademi University
Lemminkäisenkatu 14A, Turku 20520, Finland
telephone: +358-2-2154032, e-mail: Anna.Mikhajlova@abo.fi

Abstract. Object-oriented systems consist of collaborating or interacting objects. The ability to model object clients and reason about them in a formal framework appears to be critical for ensuring the systems' reliability and correctness. Knowing the precise behavior of object clients, it is possible to analyze the effect that object substitution will have on the clients, design the appropriate object interface, etc. We show how to specify object clients, and demonstrate how our approach can be used to reason about interaction between distributed objects.

1 Introduction

Object-oriented systems consist of collaborating or interacting objects. To be able to ensure correctness of an object-oriented system in its entirety, it is first necessary to capture the behavior of the system in a precise specification. Although various standard methods can be used for specifying separate objects, specifying all possible clients of a given object presents a challenge. The ability to model and reason about object clients is critical for reasoning about systems consisting of interacting objects. The developer of a class should be aware of the behavior of clients that will use instances of the class. Knowing the precise behavior of the clients, it is possible to analyze the effect that substituting subclass instances for superclass instances will have on the clients, design the appropriate object interface, etc.

In this paper we show how to specify object clients using the mathematical foundation for reasoning about interactive systems, based on the idea of contracts as described in [3, 4], and developed in [1] to support contracts between objects and their clients. In particular, we demonstrate how our approach can be used for reasoning about objects interacting with other objects possibly residing in different address spaces across the network. Such *distributed objects* can issue and receive *remote method calls* with the object issuing the call referred to as *client* and the object receiving the call referred to as *server*. To bridge the conceptual gap between local and remote method calls, objects known as *client-sides proxies* and *server-side proxies* [8] are usually used. The key idea behind proxies is that the client calls a method m of the client-side proxy to achieve the effect of calling the method m of the remote server; similarly, the server-side proxy is used to transform an incoming request into a local call form and to transform the result of the call to a form suitable for transmitting to the client-side proxy. As such, the client-side proxy can be considered as a local representative of the corresponding remote object, while the server-side proxy as the representative of all potential clients of the remote object [7]. We show how our model of object clients can be used to specify proxies and how one can reason about their properties.

2 Specifying Classes, Objects, and Object Clients

2.1 Specification Language

The specification language that we use for specifying and reasoning about object-oriented programs is essentially a combination of specification constructs and standard executable statements found in any imperative object-oriented language. The specification constructs that we use here are *assertion*, *assumption*, *nondeterministic assignment*, *input statement*, *nondeterministic choice*, and *iterative choice*. The intuitive meaning of these additional statements is described in Fig. 1 and their formal semantics can be found, e.g., in [4, 1].

The assertion is a statement found in many programming languages, such as, e.g., Eiffel and C++. Assertions can be used to state contractual obligations between methods of some object and the client program invoking these methods. For example, a method m with input parameter c of a reference type can use a statement assert $c \neq null$ to express that a client invoking m and passing it an object reference $objC$ must assert that $objC \neq null$.

The assumption, also known as the *guard*, is the dual of the assertion, and is used to express the conditions that clients of a method can assume about the behavior of the method.

The nondeterministic assignment allows us to express "what is being done without saying how it is being done", i.e. abstract away from implementation details, yet specify precisely what functionality should be achieved. For example, if we want to specify a method calculating a square root of x with the precision e, provided that the precision is positive, we can use a nondeterministic assignment

assert p	skips if p holds in a state and aborts otherwise
assume p	skips if p holds in a state, otherwise establishes any postcondition
$[x := x' \mid b]$	assigns to x a nondeterministically chosen x' satisfying the condition b
$\{x := x' \mid b\}$	assigns to x a value x' chosen by the user and satisfying the condition b
choose S_1 or ... or S_n	executes one of the alternatives S_1 through S_n
do $q_1::S_1 \Diamond \ldots \Diamond q_n::S_n$ od	iteratively chooses one of the alternatives S_i provided that q_i holds

Fig. 1. Specification Statements

preceded with the corresponding assertion:

$$\mathsf{assert}\ e > 0; [x := x' \mid -e < x'^2 - x < e]$$

The input statement models a free choice that the user has in selecting a new value x' satisfying b. The input statement can be understood as a request to the user to supply a new value for x.

The nondeterministic choice allows us to express the nondeterminism in selecting one behavior among several possible behaviors. For example, we can specify a menu offering the choice of opening a new file, opening an existing file, and closing an open file as follows:

$$\mathsf{choose}\ \mathit{fm.OpenNew}\ ()\ \mathsf{or}\ \mathit{fm.Open}\ (f)\ \mathsf{or}\ \mathit{fm.Close}\ (f)$$

where *fm* is a file manager object with the interface including the corresponding method signatures.

Finally, the iterative choice models an event loop allowing the user to repeatedly choose from a number of actions an alternative which is enabled and have it executed. For example, we can specify a font selection dialog box, where the user is offered the choice of selecting a particular font and its size, as follows:

$$\begin{array}{rcl} \mathit{DialogBoxSpec} & = & \mathsf{assume}\ \mathit{fentry} \in \mathit{Fonts} \wedge \mathit{sentry} \in \mathit{Sizes}; \\ & & \mathsf{do}\ \ \mathsf{true} :: \{\mathit{fentry} := s \mid s \in \mathit{Fonts}\} \\ & & \ \ \Diamond\ \ \mathsf{true} :: \{\mathit{sentry} := n \mid n \in \mathit{Sizes}\} \\ & & \mathsf{od} \end{array}$$

The user can select a font by typing the font name in the entry field $\mathit{fentry} : \mathsf{String}$; the selection is accepted if the entered font name belongs to the set of available fonts $\mathit{Fonts} : \mathsf{set\ of\ String}$. The size of the font can also be chosen by typing the corresponding number in the entry field $\mathit{sentry} : \mathsf{Nat}$. The user may change the selections of both the font and the size any number of times before he presses the OK button, which results in closing the dialog box and changing the corresponding text according to the last selection. When the user opens the dialog box, he assumes that the default entries for the font name and size are among those available in the system, as expressed by the corresponding assumption in *DialogBoxSpec*. If this assumption is met by the system, the user may enter new font name, or new font size, or leave the current selections intact. The user may select any alternative any number of times until he is satisfied with the choice and decides to stop the iteration. Note that to model dialog closing, we do not need to explicitly maintain a boolean variable *Ok_pressed*, have all the options enabled only when $\neg \mathit{Ok\_pressed}$ holds, and set it explicitly to true to terminate iteration: all this is implicit in the model.

Apart from the ordinary executable statements, like assignments, conditionals and loops, and the specification statements that we have described above, our specification language naturally supports object-oriented constructs such as objects, methods and classes, and mechanisms such as subtype polymorphism and dynamic binding. The formal semantics of these constructs and mechanisms is presented in [6, 2]. Using this specification language, we can freely intermix executable statements, such as method invocations, with nondeterministic statements, such as nondeterministic choices, as was demonstrated in our file management menu example above.

2.2 Modeling Classes, Objects, and Object Clients

A class can be given by the following declaration:

$$
\begin{aligned}
C \ = \ & \mathsf{class} \\
& \quad \mathsf{var}\ attr_1 : \Sigma_1, \ldots, attr_m : \Sigma_m \\
& \quad C\,(\mathsf{val}\ x_0 : \Gamma_0) \ = \ K, \\
& \quad Meth_1\,(\mathsf{val}\ x_1 : \Gamma_1) : \Delta_1 \ = \ M_1, \\
& \quad \ldots \\
& \quad Meth_n\,(\mathsf{val}\ x_n : \Gamma_n) : \Delta_n \ = \ M_n \\
& \mathsf{end}
\end{aligned}
$$

Here $(attr_1, \ldots, attr_m)$ are the attributes that carry the internal state of class instances. The constructor C and the methods $Meth_1, \ldots, Meth_n$ are given by the statements K and $M_1, \ldots, M_n$ respectively. The constructor is used to instantiate objects and, semantically, is equivalent to a stand-alone global procedure which is associated with the class for encapsulation reasons. The methods operate on the attributes and realize the object functionality. Invocation of a method on a class instance (object) has a standard procedure call semantics, with the only difference that the value of the object itself is passed as a value-result argument. We will denote invocation of $Meth_i$ on an object c instantiated by class C with value and result arguments $v : \Gamma_i$ and $r : \Delta_i$ by $r := c.Meth_i(v)$.

A client program using an object $c : C$ does so by invoking its methods. Every time a client has a choice of which method to choose for execution. In general, each option is preceded with an assertion which determines whether the option is enabled in a particular state. While at least one of the assertions holds, the client may repeatedly choose a particular option which is enabled and have it executed. The client decides on its own when it is willing to stop choosing options. Such an iterative choice of method invocations, followed by arbitrary statements not affecting the object directly, describes all the actions the client program might undertake:

$$\mathsf{begin\ var}\ l \bullet b; \mathsf{do}\ \lozenge_{i=1}^{n} q_i :: d_i := c.Meth_i(g_i); L_i\ \mathsf{od\ end}$$

Here l are some local variables initialized according to b, predicates $q_1 \ldots q_n$ are the asserted conditions on the state, and statements L_1 through L_n are arbitrary. The initialization b, the assertions $q_1 \ldots q_n$, and the statements $L_1, \ldots, L_n$ do not refer to c, which is justified by the assumption that the object state is encapsulated.

Objects can, of course, be their own clients, and any method of class C can invoke other methods of C, including itself. The most general behavior of a method $Meth_j(\mathsf{val}\ x_j : \Gamma_j) : \Delta_j$ can, therefore, be described by

$$\mathsf{begin\ var}\ l \bullet b; \mathsf{do}\ \Diamond_{i=1}^{n} q_i :: d_i := self.Meth_i(g_i); L_i\ \mathsf{od\ end}$$

where *self* is free in b, all q_i can directly access *self*, and all L_i can directly access and modify it. For example, a method self-calling itself is an instance of this general definition. The meaning of such a recursive method is given by the least fixpoint of the corresponding function with respect to the refinement ordering.

The whole program statement is a contract between the object c and *any client program* using c. The method enabledness condition q_i corresponds to the assumptions made by the corresponding method $Meth_i$, as stated in its subcontract (the method body definition). For example, in a component *EntryField* the method $SetLength(\mathsf{val}\ l : \mathsf{Nat})$ can begin with an assumption that the length l does not exceed some constant value *lmax*. An environment invoking *SetLength* on *EntryField* will then have to assert that the specific *length* does indeed satisfy this requirement:

$$\mathsf{do}\ length \le lmax :: EntryField.SetLength(length); \ldots\ \mathsf{od}$$

The assumption of this condition in the body of *SetLength* will pass through, as $\mathsf{assert}\ p; \mathsf{assume}\ p = \mathsf{assert}\ p$, for all predicates p.

The theoretical foundation developed in [1] permits one to reason about behavioral properties of the iterative choice statement, and therefore, of object clients that this statement permits modeling. In particular, one can prove that given a precondition, the client program can establish a certain postcondition. For example, in [1] it was proved that the user of the dialog box, whose specification was presented above, can, by making the right choices, select a font of his or her preference, provided that it is one of the fonts available in the system. Furthermore, in [1] we prove that one can *refine* an iterative choice statement through offering new options, or refining the original options, or both. Therefore, the client program will be refined, i.e. will behave at least as well or even better, more deterministically, than before, if instead of using objects of the original class it will use objects of its subclass, provided that the subclass is a refinement of the original class. The details of class refinement are outside the scope of this paper and are presented in [6, 2].

3 Example: Specifying Distributed Objects

Let us now demonstrate how our approach to modeling clients can be used to specify participants in a distributed object communication protocol. As we already mentioned above, the conceptual gap between local and remote method calls can be bridged through the use of proxy objects. The client-side proxy supports the same interface as the remote server, and the client object, to achieve the effect of calling the method m of the remote server, calls a method m of the client-side proxy. Similarly, the server-side proxy is used to transform an incoming request into a local call form and to transform the result of the call to a form suitable for transmitting to the client-side proxy over the network. As such, the client-side proxy can be considered as a local representative of the corresponding remote object, while the server-side proxy as the representative of all potential clients of the remote object.

Suppose we would like to specify an electronic banking system which allows a customer to perform financial transactions through interacting with his bank. Naturally, the customer and the bank can be physically located in separate places, and the system should permit their interaction across the network. Figure 2 shows the client/server diagram reflecting interaction of the involved parties in a distributed environment. In this diagram *Customer* is the client, *Bank* is the remote server, and *BankProxy*, *CustomerProxy* are the client-side and the server-side proxies respectively. This architectural solution is an instance of the Broker pattern [7] with *direct communication* between client-side and server-side proxies. With direct communication, the client-side proxy and the server-side proxy communicate directly after a communication channel between them was initially established. With indirect communication, a special *Broker* object mediates between proxies, but we do not consider this variant of the Broker pattern here.

Checking balance on a *Customer* account in a *Bank* illustrates a typical communication scenario between the *Customer* and the *Bank*. The *Customer* asks its bank's local representative *BankProxy* to check balance on the specified account through invoking the method *AccountBalance* of the latter. The *BankProxy* packs the data identifying the *Customer*'s account in a format suitable for transmitting over the network, and calls service on the *CustomerProxy*, passing it the

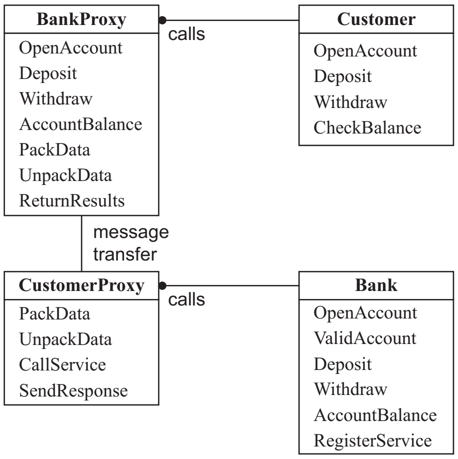

Fig. 2. Distributed electronic banking system

packed data. The *CustomerProxy* checks whether the associated *Bank* has registered with it the service requested by *BankProxy* on behalf of the *Customer*. If indeed it has registered such a service, then the *CustomerProxy* unpacks the data it gets from the *BankProxy* and calls the requested service on the *Bank* through invoking its *AccountBalance* method. The *Bank* checks, through calling its *ValidAccount* method, whether the specified account is valid and informs the local representative of the *Customer*, i.e. *CustomerProxy*, about the state of the account, through calling its method *SendResponse*. The *CustomerProxy* reacts by packing the data and returning it over the network channel to *BankProxy* which, in turn, unpack the information about the state of the account and returns it to the *Customer*.

Of particular interest in this architectural solution is the behavior of the *CustomerProxy*. As was mentioned earlier, the server-side proxy represents all potential clients of the server object. Indeed, the method *CallService* of the *CustomerProxy* embodies all potential clients of the *Bank* since it conveys clients' requests for all the (functional) services that the *Bank* provides. Let us specify the behavior of this method.

```
CustomerProxy = class
  var bank : Bank, client : BankProxy, services : set of Service
  CallService(val s : Service, c : Customer, a : AccountId, m : Amount) =
    begin var id : AccountId, b : Amount;
      if s ∈ services then
        do  s = 'open' ∧ m > 0        :: id := bank.OpenAccount(c,m);
                                         self.SendResponse(id)
        ◊   s = 'deposit' ∧ m > 0     :: bank.Deposit(a,m)
        ◊   s = 'withdraw' ∧ m > 0    :: bank.Withdraw(a,m)
        ◊   s = 'checkbalance'        :: b := bank.AccountBalance(a);
                                         self.SendResponse(b)
        od
      else self.SendResponse('Service ', s, ' is not supported!') fi
    end
  ...
end
```

This specification succinctly and precisely expresses the behavior of the proxy object. As seen from this specification, *CallService* is specified as a client of both *self* and the object *bank*, which is an instance of class *Bank*. Note that the methods *OpenAccount, Deposit*, and *Withdraw* of *Bank* are invoked under the additional condition that the sum m passed as an argument to these methods is greater than zero, which is rather meaningful, as it does not make much sense to, e.g., deposit zero units of a given currency to an account. Accordingly, the methods *OpenAccount, Deposit*, and *Withdraw* of *Bank* only *expect to be called* under the similar condition, so that the method *OpenAccount*, for example, is defined as follows:

$$
\begin{array}{l}
\mathit{OpenAccount}(\mathsf{val}\ c : \mathit{Customer},\ m : \mathit{Amount}) : \mathit{AccountId}\ = \\
\quad \mathsf{assume}\ m > 0; \\
\quad \mathsf{begin\ var}\ a : \mathit{Account}; \\
\qquad a := \mathsf{new}\ \mathit{Account}(c,\ m); \mathsf{return}\ a.\mathit{AccountNumber}(); \\
\quad \mathsf{end}
\end{array}
$$

Here *Account* is a class of accounts having an identically named constructor and methods including the method *AccountNumber*, which returns the number of a given account. The assumption of the condition $m > 0$ in the definition of *OpenAccount* is a part of the contract between this method and any client invoking it. If a client invokes *OpenAccount* in a state where this condition does not hold, the client breaches the contract and aborts. In fact, the conditions assumed by a method and asserted by a client calling this method do not need to be identical: the assumed condition can be weaker. If a client asserts that a stronger condition holds in a state, then a weaker condition will also hold. Accordingly, the method *OpenAccount* could just as well be defined to assume $m \geq 0$ instead of $m > 0$. An invocation of this method in a state where $m > 0$ holds would still work.

Using the specification of the method *CallService* expressed in terms of the iterative choice as shown above, we can reason about the behavior of the interacting objects and verify certain behavioral properties, such as delivering a particular request for a service from the client *Customer* to the server *Bank*. For instance, we can verify that calling the method *CallService* with the request to deposit a certain sum of money on a valid account will result in the balance of this bank increasing by this sum (provided that the method *Deposit* of *Bank* is specified accordingly). Also, we can reason about the behavior of *CallService* in the case when the *bank* attribute of *CustomerProxy* is instantiated with a subclass of *Bank* – *BetterBank* – which provides extra services and "improves the quality" of the original services provided by *Bank*. In particular, such a *BetterBank* can offer a loan or open a savings account to the *Customer*, providing the corresponding methods

$$
\begin{array}{l}
\mathit{Loan}(\mathsf{val}\ c : \mathit{Customer},\ m : \mathit{Amount}) : \mathit{AccountId}\ \text{and} \\
\mathit{PayInterest}(\mathsf{val}\ \mathit{id} : \mathit{AccountId}) : \mathit{Amount}
\end{array}
$$

for these purposes. To be able to use these new services, the *CustomerProxy* should be subclassed to a *BetterCustomerProxy* overriding the method *CallService* as follows:

$$
\begin{array}{l}
\mathit{BetterCustomerProxy}\ =\ \mathsf{subclass\ of}\ \mathit{CustomerProxy} \\
\quad \mathsf{var}\ \mathit{bank} : \mathit{BetterBank}, \ldots \\
\quad \mathit{CallService}(\mathsf{val}\ s : \mathit{Service},\ c : \mathit{Customer},\ a : \mathit{AccountId},\ m : \mathit{Amount})\ =
\end{array}
$$

```
begin var id : AccountId, b : Amount;
  if s ∈ services then
    do  s = 'open' ∧ m > 0        :: id := bank.OpenAccount(c,m);
                                     self.SendResponse(id)
    ◊   s = 'deposit' ∧ m > 0     :: bank.Deposit(a,m)
    ◊   s = 'withdraw' ∧ m > 0    :: bank.Withdraw(a,m)
    ◊   s = 'checkbalance'        :: b := bank.AccountBalance(a);
                                     self.SendResponse(b)
    ◊   s = 'takeloan' ∧ m > 0    :: id := bank.Loan(c,m);
                                     self.SendResponse(id)
    ◊   s = 'getinterest'         :: b := bank.PayInterest(a);
                                     self.SendResponse(b)
    od
  else self.SendResponse('Service ', s, ' is not supported!') fi
end
...
end
```

As we prove in [1], adding options in an iterative choice is a refinement and, therefore, the definition of the method *CallService* in the *BetterCustomerProxy* is a refinement of its definition in the *CustomerProxy*. If other methods of the *BetterCustomerProxy* also refine the corresponding methods in the *CustomerProxy*, then the *Customer* using the *BetterBank* will get better service than if using the *Bank*.

4 Conclusions

In this paper we showed how to specify object clients, and demonstrated how our approach can be used to reason about interaction between distributed objects. As future work we would like to test the applicability of our approach to modeling other architectural solutions such as the *implicit invocation* mechanism [5].

References

1. R. Back, A. Mikhajlova, and J. von Wright. Reasoning about interactive systems. To appear in *Proceedings of the World Congress on Formal Methods (FM'99)*. Previous version appeared as a TUCS Technical Report No. 200, http://www.tucs.abo.fi/publications/techreports/TR200.
2. R. Back, A. Mikhajlova, and J. von Wright. Class refinement as semantics of correct subclassing. Technical Report 147, Turku Centre for Computer Science, December 1997. http://www.tucs.abo.fi/publications/techreports/TR147.html.
3. R. J. R. Back and J. von Wright. Contracts, games and refinement. In *4th Workshop on Expressiveness in Concurrency, EXPRESS'97*, volume 7 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1997.

4. R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.
5. D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM 91, Volume 1: Conference Contributions*, LNCS 551, pages 31–44. Springer-Verlag, Oct. 1991.
6. A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proceedings of 4th International Formal Methods Europe Symposium, FME'97*, LNCS 1313, pages 82–101. Springer, 1997.
7. F. Plášil and M. Stal. An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM. *Software - Concepts and Tools*, 19(1):14–28, 1998.
8. M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computer Systems*, pages 198–205, Washington, 1986. IEEE Press.