Authors address: LIHS-FROGIS, Université Toulouse I, Place Anatole France, F-31042 Toulouse CEDEX, France

Contact Email: bastide@univ-tlse1.fr

Keywords: Distributed systems, CORBA, behavioural specification, high-level Petri nets

Abstract: We present a CASE tool based on an object-oriented Petri nets dialect called Cooperative Objects, dedicated to the design of CORBA systems. The notation is used for the formal behavioural specification of objects, and its associated tool puts an emphasis on supporting the design life cycle of CORBA systems. The tool offers enhanced interactivity to present the results derived from the capabilities of verification, validation and distributed interpretation provided by Cooperative Objects.
Formal Support for the Engineering of CORBA-based Distributed Object Systems

Rémi Bastide, Ousmane Sy, Philippe Palanque
LIHS-FROGIS, Université Toulouse I, Place Anatole France,
F-31042 Toulouse CEDEX, France
{bastide, sy, palanque} @univ-tlse1.fr

Abstract
We present a CASE tool based on an object-oriented Petri nets dialect called Cooperative Objects, dedicated to the design of CORBA systems. The notation is used for the formal behavioural specification of objects, and its associated tool puts an emphasis on supporting the design life cycle of CORBA systems. The tool offers enhanced interactivity to present the results derived from the capabilities of verification, validation and distributed interpretation provided by Cooperative Objects.

1 Introduction

The Object Management Group (OMG) has defined a standard for distributed objects communication called Common Object Request Broker Architecture (CORBA) [11]. Our team has been promoting the use of formal methods (and more precisely of Petri nets) in the field of distributed object systems [4], by the definition of an object-oriented, Petri-net based formal notation called Cooperative Objects (CO). The key idea is that Petri nets can be used to complement CORBA-IDL interfaces with a formal behavioural specification. The use of such a formal notation brings several advantages: Firstly and foremost, it provides a concise, complete and non-ambiguous specification of the behaviour of objects that implement an interface; this is highly beneficial, both for the software engineer that is in charge of implementing the object in some conventional programming language, and for developers of systems that will use this object. Secondly, and equally important, is the fact that a formal notation such as Petri nets is provided with several analysis techniques that allow to reason on the models, and to check important properties such as liveness and boundedness.

However, a complain often heard about formal approaches is that they are too far from the hard realities that software designers have to face in realistic projects, and that their cost is often excessive. Our goal is to show that the cost/benefit ratio of our approach can be attractive, provided that it is supported by a CASE tool carefully designed to take into account the actual tasks of engineers working in a CORBA-based software project. More precisely, we have developed a tool that addresses five main objectives:
1. The support of the design life cycle of CORBA systems;
2. The provision of an interpreted environment supporting immediate testing of the system under design;
3. The generation of a prototype of the system from the specification;
4. The analysis of the system to provide validation and verification support;
5. The pedagogic support for the teaching of Petri nets, and especially of CO.

The paper is organised as follows: in the first part, we present the CORBA standard of distributed object-oriented systems and highlight several of its shortcomings. In the second part we present the formalism, the Cooperative Objects. We then describe PetShop, the tool supporting our formalism, and give an overview of its interactive features and functionality.

2 Distributed object development with CORBA

CORBA describes the distributed objects in a programming language-independent way, through a specific Interface Definition Language (IDL). An IDL interface defines the public services provided by a class of objects, as well as the data types exchanged through service invocation. For consistence of implementation of the standard among the various ORB suppliers, the OMG has defined five mappings of the IDL to programming languages: C, C++, COBOL, SmallTalk, ADA, Java™, Fortran [11].
Exception insufficientFunds;
Exception accountIsClosed;

Interface BankAccount : Account {
    void open();
    void close();
    void transfer(in float amount);
    raises(accountIsClosed, insufficientFunds);
    float balance();
};

Figure 1. IDL of the BankAccount interface

Figure 1 describes a simple interface, BankAccount, in CORBA IDL. There are two exceptions and four public services:

- **insufficientFunds** is an exception raised when the account holder attempts to withdraw more money than available;
- **accountIsClosed** is an exception raised when the client tries to manipulate funds of a closed account;
- **open** is a service that opens a bank account;
- **close** is a service that closes an account;
- **transfer** is a service that deposits money into or withdraws money from the account. The amount is given as a float parameter. It may raise the above mentioned exceptions;
- **balance** is a service that returns the amount of money available in the account.

![Figure 2. Client/server interaction with CORBA](image)

In CORBA, the key component for inter-object communication is the Object Request Broker (ORB). Its task is to support distributed objects’ communications so that distribution is transparent in terms of invocation. Invocations take place according to a client/server protocol (Figure 2): for one object, C, to invoke another object, servA, and become its client, it is only necessary that C holds a reference to the server, servA. The ORB takes care of the difficulties of remote invocation: delivering a service invocation and returning the result, data marshalling and unmarshalling.

2.1 An overview of software development with CORBA

In this section, we describe the typical tasks done by programmers intending to develop a CORBA-based system. We describe six steps (Figure 3) corresponding to the general case of static CORBA invocations and give some applications using the Java mapping.

![Figure 3. Development with CORBA](image)

1. IDL specification: a programmer describes the services offered by an interface and their signatures in CORBA IDL.
2. IDL compilation: a programmer compiles the IDL source with an IDL compiler. This compilation produces two pieces of code in a target development language: the stub and the skeleton. According to the language mapping, some additional files may also be generated.
3. Skeleton implementation: a programmer writes a CORBA servant that implements the skeleton produced in step 2. For instance, in the Java language mapping, the skeleton is an abstract class that needs to be specialised to provide an implementation class.
4. Writing server program: a programmer writes a server program that instantiates and uses a CORBA servant, servA. This instance must be registered with the ORB.
5. Writing client program: a programmer writes a client program that uses the stub produced in step 2.
6. Code compilation: finally, a programmer compiles the client and server programs to produce executable code.
public class Client {
    public static void main(String[] args) {
        float res;
        org.omg.CORBA.ORB orb;
        // initialise the ORB
        orb = org.omg.CORBA.ORB.init(args, null);
        // get a reference to the servant
        BankAccount proxyA = BankAccountHelper.bind(orb, "myServer");
        try {
            res = proxyA.balance();
            // Print out the result
            System.out.println("Your balance is " + res);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}

Figure 4 Client code in Java.

Figure 4 illustrates a client, (that could be written in step 5), using the IDL-to-Java mapping and the Visibroker for Java 3.2 ORB [7]. The Client class has a main method (line 3), inside which it first initialises an ORB instance (line 5). It then declares a reference to the stub called proxyA (line 9), and binds it to a specific CORBA servant instance implementing the interface BankAccount (Figure 1), called "myServer" (line 9). This binding is done with the help of an auxiliary class produced in step 2 called BankAccountHelper (line 9). Finally, the client invokes the service balance in the right-hand side of the statement in line 11. The call is dispatched to the ORB using operations defined in the stub (Figure 2). The ORB then forwards the call and uses the operations defined in the skeleton to transmit the call to servA (Figure 2) which executes servA.balance(). The invocation ends either when the result is transmitted back to the client or when a CORBA exception is raised (Figure 2).

2.2 Shortcomings of CORBA

- **Lack of behavioural specification:** one of the problems with CORBA IDL is that it only defines signatures of services and does not address the behavioural specification of CORBA servants. As shown Figure 3, after IDL compilation, the next step is the implementation of the client. In order to use properly an object implementing the interface A, a client program needs some information about the behaviour of the servant. For instance, it needs to have some semantics about legal sequencing of invocation. The developers of CORBA servants usually receive the behavioural information about the class they have to implement in a "semi-formal" way, or in plain English text. Ultimately, the implementation code written in some programming language will often constitute the only actual formal behavioural specification of the class. We consider that a form of abstract, programming-language independent behavioural specification is highly beneficial to the development of CORBA systems. Therefore we have implemented tools to support CORBA application development with the help of Cooperative Objects, a member of the family of object-oriented Petri net formalism.

- **Cumbersome development:** as shown in Figure 3, the developer usually expresses the behaviour of the CORBA servants directly in a programming language. Doing so, the programmer breaks the link between specification and implementation, and makes it difficult to test the CORBA servant's implementation with regard to its specification. Therefore we have integrated the specification in the development phase smoothly, to ease the continuous task of specifying and tuning done by the developer following the advice of Holt and deChampeaux [6]. Another critical point in the design process is that it takes a long time before a designer can actually test the behaviour of the object designed and get insights on the way different parts of the system may interact. Therefore we have allowed, on one hand, for the immediate interpretation and testing of the CO Petri net as soon as its editing begins, and on the other hand the generation of standalone prototypes of CORBA servants at the designer's convenience.

3 The PetShop Tool

In this section, we briefly present the formalism used to complement the limitations of CORBA IDL in terms of behavioural specification, and we describe the tool that support this formalism. The formalism and its supporting tool are aimed at overcoming the shortcomings stated above. The theoretical details of the formalism and its integration with CORBA have been detailed elsewhere [4], and the remainder of the paper focuses on tool support.

3.1 The Cooperative Object Formalism

Cooperative Objects [3] (CO) are a dialect of object-structured, high-level Petri nets. Their lengthy formal
modelling of concurrent systems. A Petri net models a for a long time as a mathematical formalism for the state-changing operators called (represented as ellipses) and a set of state-changing information elements (called ). The state of the system is described as a distribution of tokens in the net’s places; this distribution is called the marking of the net. In basic Petri nets, tokens are dimensionless entities modelling only conditions. Several dialects of Petri nets (called high-level Petri nets) allow tokens to carry information and to manipulate this information at the occurrence of transitions. Our own dialect is close to well-known high-level Petri nets such as coloured Petri nets [8] or Predicate-Transition nets [5]. They differ from these mainly by the nature of the inscriptions attached to the net elements, and by their object-oriented structure.

In Petri nets, the causality structure of the systems (stating under which condition a change of state may occur, and what will be its resulting state) is described by arcs connecting places and transitions.

The input arcs of a transition (coming from places of the net) describe the preconditions of an occurrence of this transition. The transition may occur if the input places hold enough tokens. Conversely, the output arcs of a transition (going to places of the net) describe the postconditions of an occurrence of the transition, in terms of changes in the net’s marking. After an occurrence, tokens are removed from the input places of the transition, and new tokens are set in its output places.

Petri nets allow for modelling very naturally systems with distributed state, and concurrent activities. Two transitions that do not share any input place may occur concurrently (marking permitting) and the resulting state will be the same whether they occur concurrently, or sequentially in any order.

A huge amount of work has been devoted to the study and analysis of Petri nets. Several important safety and liveness properties, as well as invariants in the dynamic behaviour can be proved by mathematical means.

### 3.1.1 Petri nets basics

Petri nets [13] have been studied for a long time as a mathematical formalism for the modelling of concurrent systems. A Petri net models a system as a set of state variables called places (represented as ellipses) and a set of state-changing operators called transitions (represented as rectangles). The state of the system is described as a distribution of information elements (called tokens) in the net’s places; this distribution is called the marking of the net. In basic Petri nets, tokens are dimensionless entities modelling only conditions. Several dialects of Petri nets (called high-level Petri nets) allow tokens to carry information and to manipulate this information at the occurrence of transitions. Our own dialect is close to well-known high-level Petri nets such as coloured Petri nets [8] or Predicate-Transition nets [5]. They differ from these mainly by the nature of the inscriptions attached to the net elements, and by their object-oriented structure.

In Petri nets, the causality structure of the systems (stating under which condition a change of state may occur, and what will be its resulting state) is described by arcs connecting places and transitions.

The input arcs of a transition (coming from places of the net) describe the preconditions of an occurrence of this transition. The transition may occur if the input places hold enough tokens. Conversely, the output arcs of a transition (going to places of the net) describe the postconditions of an occurrence of the transition, in terms of changes in the net’s marking. After an occurrence, tokens are removed from the input places of the transition, and new tokens are set in its output places.

Petri nets allow for modelling very naturally systems with distributed state, and concurrent activities. Two transitions that do not share any input place may occur concurrently (marking permitting) and the resulting state will be the same whether they occur concurrently, or sequentially in any order.

A huge amount of work has been devoted to the study and analysis of Petri nets. Several important safety and liveness properties, as well as invariants in the dynamic behaviour can be proved by mathematical means.

### 3.1.2 Structure of a CO class

A CO class specifies a class of objects by providing their interface (the set of services offered, along with their signature) and their dynamic behaviour. The behaviour of a CO class is called its Object Control Structure (ObCS), and is defined with a dialect of high-level Petri nets. More specifically:

- Tokens are tuples of typed values. The arity of a token is the number of values it holds, and tokens of zero-arity are thus the “basic” tokens used in conventional Petri nets. We will call Token-type a tuple of types, describing the individual types of the values held by a token. Token-types will be noted `<Type1, ..., TypeN>` or just `<>` to denote the Token-type of zero-arity tokens.
- Places are defined to hold tokens of a certain Token-type, thus all tokens stored in one place have the same Token-type and arity. A place holds a multiset of tokens, thus a given token may be present several times in the same place.
- Each arc is inscribed by a tuple of variables, with a given multiplicity. The arity of an arc is the number of variables associated to it. The arity of an arc is necessarily the same as the arity of the Token-type of the place it is connected to, and the type of each variable is deduced from this Token-type. The multiplicity of an arc is the number of identical tokens that will be processed by the firing of a transition associated to this arc. The general form of an arc inscription is `Multiplicity*<v1, ..., vn>`. A multiplicity of 1 can be omitted (thus `1*<v1, ..., vn>` can be abbreviated as `<v1, ..., vn>`) and an empty list of variables can also be omitted (thus `2*<>` can be abbreviated as `2`).
- Transitions have a precondition (a Boolean expression of their input variables) and an action, which may use any service allowed for the types of their input or output variables. The scope and type of each variable of an arc is local to the transition the arc connects to. A transition is enabled when:
  - A substitution of its input variables to values stored in the tokens of its input places can be found
  - The multiplicity of each substituted token in the input places is superior or equal to the multiplicity of the input arc,
  - The precondition of the transition evaluates to true for the substitution.

The firing of a transition will execute the transition’s action, remove tokens from input places, compute new tokens and store them in the output places of the transition.

The formalism also supports two arc extensions [9]: test arcs (allowing to test for the presence of tokens in input places of a transition, without removing them at the occurrence of the transition) and inhibitor arcs (allowing to test for the absence of certain values in input places of a transition).

### 3.1.3 ObCS example

We now show a short specification of an implementation class for the IDL example given in Figure 1.
class BankAccountSpec
specifies BankAccount {
    place closed <float> = { <0> };
    place open <float>;
    transition transferFunds {
        precondition {
            (amount + balance) >= 0;
        }
        action {
            newBalance = amount + balance;
        }
    }
    transition t5 {
        precondition {
            (amount + balance) < 0;
        }
    }
}

Figure 5. CO specification of the BankAccountSpec class

The CO specification consists both of a textual declarative part and an ObCS. Figure 5 shows the CO specification of a BankAccountSpec class, along with the textual notation used to describe the Token-types of the places, their initial marking and the precondition and action of the transitions. Each service offered by the interface is mapped to two special places in the ObCS net:

- The Service Input Port (SIP) receives tokens corresponding to the input parameters (e.g. ⚪️, which is written as open in this article).
- The Service Output Port (SOP), which contains the tokens corresponding to the result and output parameters (e.g. 🔴, which is written open hereafter).

Figure 5 illustrates two different kinds of arcs: normal arcs and test arcs (between the place closed and transition T2).

There are two types of transitions used: normal transitions (e.g. transferFunds) and those that trigger exceptions (e.g. accountClosed).

The service open (transition T4) is invoked when a client object requests the opening of a new account. T4 has two input places (closed and open) and two output places (opened and open). Initially, there is a token of token-type <float> in the place closed. When the service open is invoked, a "black and white" token is deposited in the place open. As a result T4 is enabled. The occurrence of T4 results in removing the token from each input place and depositing one token in each output place: place opened receives a token {<0>} and place open a token {<>}. 
3.2 CORBA-based development with PetShop

In this section, we describe how the PetShop CASE tool fits into the CORBA program development cycle. As a whole, PetShop brings the ability to edit behavioural specification while handling the CORBA design phases of IDL shown in section 0. In addition to the steps shown in 0, it spans up to the prototyping and execution phase of the model. To describe PetShop we shall describe the different steps of development with the tool (Figure 6):

- Cooperative object class editing;
- IDL generation;
- Cooperative object class analysis;
- Interactive interpretation of models;
- Distributed systems prototyping.

3.2.1 Cooperative object class editing. The Petri net editor provides the graphical interface for both the editing of the textual part and the ObCS part of each CO class. As the ObCS can specify several IDL interfaces, the tool also provides access to existing IDL interfaces and allows for the creation of new IDL interfaces. The user can access the CORBA Interface Repository (IR) to query for existing interfaces. When an interface is chosen, a basic behavioural skeleton is automatically created in the tool, and can be further refined by the user.

3.2.2 IDL generation. The original goal of our approach is to add behaviour to IDL. A frequent design activity is to start from pre-defined IDLs (such as the ones provided in the definition of the CorbaServices [12]) and to write a behavioural specification for it. The designer can also start from scratch and generate CORBA IDL matching a given behavioural specification. This possible way of designing is also supported and the IDL is then generated from the Cooperative Object net. We thus support flexibility into CORBA development process.

3.2.3 Cooperative object class analysis. As Cooperative objects are a subclass of Petri nets, PetShop integrates several analysis modules that allow proving some interesting properties of the nets such as invariants, conflicts (structural properties), liveness, boundedness, home state (behavioural properties).

We wish to use Petri net analysis techniques not only to prove properties on an isolated object, but also to analyse constructs specific to object-oriented systems. For example, when two IDL interfaces are related through inheritance, some form of behavioural inheritance needs to be respected for CO classes that specify these interfaces [17]. We also want to analyse the co-operation between several CO instances, to check properties of a system of interacting objects. This global net can be used to prove properties of the system as a whole.

3.2.4 Interactive interpretation of models. One of the well-known advantages of Petri nets is their executability. This feature is exploited in the PetShop tool that offers the well-known advantages of interpreted environments in terms of flexibility, interactivity and ease of use [10]. PetShop allows an interpreted object to seamlessly access...
"real" third-party CORBA servants, or to be accessed as a servant from other CORBA objects developed independently and maybe in a different programming language.

In our implementation, each edited net is executed using the ObCS interpreter described in [2]: the editor itself acts as a client/server program. As a result, as soon as a net is edited, an instance of the ObCS starts executing. The interpreted instance is accessed using mechanisms provided by CORBA for the sake of interpreted, dynamically typed languages such as LISP. These mechanisms are known as Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI) [11]. DII occurs on the client’s side: it allows an object to dynamically create a request object, insert parameters in this request, send the request to a server and retrieve the results. Conversely, DSI occurs at the server’s side: it allows an object to receive any request through a single operation, to determine which operation has been called, to process the request and insert results in it. PetShop makes extensive use of DSI and DII, to allow for interactive testing of the behaviour in a CORBA runtime environment and to make real distributed experiments between CORBA objects hosted by the same or different computers.

The environment allows for graphic debugging of the interpreted net, by examining or modifying its marking, or even modifying its control structure dynamically.

3.2.5 Distributed System prototyping. In the last phase of development, the programmer is interested in shipping a finished stand-alone CORBA object that can be run without the support of the CASE tool. For that reason PetShop can generate a prototype and launch instances of the prototypes. Those instances are “almost functional” implementations, in the sense that only behavioural requirements are dealt with. Other “quality of service” requirements, such as performance, persistence, replication or fault-tolerance, have to be taken care of in a completely functional implementation. The generated prototype is still an interpreted high-level Petri net, but does not have graphical features, like graphic debugging or live editing capabilities, which allows it to perform faster.

3.3 Interactive Features

We want to achieve a highly interactive environment, designed after a careful study of the task of designers of CORBA-based systems. In this part, we give an overview of the tool’s interactive features.

3.3.1 Non-modality. Most Petri net environments feature a “modal” user interface. The user switches from editing mode (where the net structure is altered) to interpretation mode (where the net can be executed), or to analysis mode (where some properties can be checked over the net). In our tool, there is no such distinction between these modes: the net is always editable and executable. For instance, new nodes can be added or deleted at any time when the net is running.

Analysis of the net is performed in background, concurrently with the editing and simulation. An agent-based user interface is responsible for delivering the results of analysis: analysis agents are constantly at work on the net under design, and analysis results and the graphical rendering invariants are provided as soon as they are available, while the net is still being edited.

3.3.2 Interpreter control

To each edited ObCS class corresponds a running interpreter instance. This interpretation is controlled with VCR-like control panel (Figure 7) allowing to switch between different execution modes:

- Automatic mode with , in which the selection of the next transition to fire and choice of variable substitution is done automatically. The stop button allows halting the execution. Therefore the automatic and stop buttons are mutually exclusive;
- Step by step execution, in which the user can either fire a randomly chosen transition among the enabled ones using the button or choose directly the transition to fire. In the last case, the user can let the tool decide automatically which substitution is selected or select interactively a precise substitution to fire the transition with, if the transition is enabled by several substitutions;
- Backward mode, in which is the “undo” button, allowing playing the net “in reverse”. This is a useful interactive feature, especially during debugging sessions. It allows the designer to investigate various behaviours of the net, and to perform “what-if” scenarios, going backwards to the point where a choice has been made among conflicting transitions and firing another transition in the conflict set.
- Reset mode, in which the VCR control allows to bring the net back to its initial marking.
Figure 8. Selection of transition fired in step by step mode

The example in Figure 8 exemplifies the use of the controller in the step by step mode. The controller palette appears on the right, floating over the edited net. The stop button is deactivated and the other buttons are actionable.

In the net, transitions are rendered as buttons who are shaded when the transitions are not enabled. In Figure 8, only the close transition is enabled. The user sees that the places Accounts and closeAccounts are marked by a black dot, which means that there is at least one token in those places. The user can explore the possible variable substitutions enabling the transition with a right mouse click on the button. This opens a contextual pop-up menu that lists the available substitutions for that transition. The choice done in that menu triggers the firing of the transition.

3.4 Implementation

Petshop is entirely written in Java™[15]. We made the decision because CORBA is often used to build complex distributed information systems like Internet and Intranet applications. In combination to being naturally well suited to the Inter/Intranet application development, Java also provides object-orientedness and allows for code portability and executability on different platforms.

Java has been chosen as a target implementation language because it provides a clean, robust and rich object-oriented programming interface (customisable graphical library, threads, component approach with Java Beans) and because it provides a standardised possibility to generate documentation from the code and the possibility to switch from a standalone application to an applet that can be embedded in HTML documentation.

3.5 Communication, documentation and learning

One secondary goal of the tool is to allow creating “active Petri Nets texts” such as tutorials or overhead presentations.

The tool permits to integrate an executable net in presentations or any packaged documentation supporting Java Beans or ActiveX components. This allows, for example, the reader of documentation text to investigate the behaviour and properties of the model exposed. Online documentation in HTML format can be generated by the tool, which works both in application mode and applet mode.

4 Conclusion and Future work

The present paper focuses on PetShop, the tool that supports the Cooperative Objects (CO) formalism and gives little detail on the formalism itself. PetShop is a CASE tool that supports the formal behavioural specification of CORBA servers and at the same time actively supports the development cycle. By doing so, we reduce a part of the burden of the developers of CORBA applications and offer them an interactive environment in which an object's behaviour is described in terms of a Cooperative Object net that can be immediately tested. This interactivity is pushed to the point where they can build their objects, get feedback on analytical properties and interpret the net of objects at the same time.

Formal approaches are often criticised on the grounds of their lack of scalability: they often work well on small, ad-hoc examples, but fall short when realistic applications are considered. We are very concerned with this criticism, and we have tried our approach (along with our tool) on a realistic example taken from the OMG Common Object Services Specification (COSS) [12] set: we have completed a complete formal specification of the CORBA Event Service, which is meant to provide asynchronous, multicast communication primitives [16]. This fairly large case study as brought a number of insights on the formalism and the tool. For instance, it has shown us that the formalism was in need of some “syntactic sugaring”, to ease the specification, notably when exceptions are considered. It has also highlighted several
underspecifications in the original OMG text, that allowed us to find some inconsistencies in actual third party implementation, notably when boundary conditions were considered. Altogether, we were rather satisfied with the scalability of the tool in this reality check.

Future work on the tool will put more emphasis on documentation generation and on the integration of component technologies to specify nets. As for the formalism itself, we will tackle the problems of behavioural inheritance and co-operation. In the domain of software engineering, an important objective is to generate test suites from the Cooperative Object class definition so that the implementation can be tested for conformity against the original CO-based specification.

5 Acknowledgements

This work is partly supported by grant N° 981b059 of CNET, the research centre of France Telecom.

6 References