Extending CORBA Interfaces with $\pi$-calculus
for Protocol Compatibility

C. Canal, L. Fuentes, J.M. Troya and Antonio Vallecillo
University of Málaga, Spain.

TOOLS Europe 2000

June 6, 2000
Interoperability

“The ability of two or more entities to communicate and cooperate despite differences in the implementation language, the execution environment, or the model abstraction” [Wegner, 1996].

- We distinguish three main levels of Object Interoperability:
  - The *Signature* level (signature of operations)
  - The *Protocol* level (partial order between messages)
  - The *Semantic* level (real “meaning” of operations)
Traditional IDLs

- Describe supported services, but not required ones.
- Describe the syntactic interfaces of objects, not their behavior.
- Are mainly used at compile time, but not during object execution.

Therefore, from an object IDL I know what an object does, but:

- I don’t know how to use its services.
- I don’t know the external services it needs.
Our main aim

- Extend IDLs with protocol information:
  - Supported and required services.
  - Partial order in which objects expect their methods to be called.
  - Partial order in which objects call other objects’ methods.

Our present contribution

- Extend the CORBA IDL.
- Use Milner’s $\pi$-calculus for protocol descriptions and compatibility checks.
1. Introduction (✓)
2. The CORBA IDL
3. The polyadic $\pi$-calculus
4. Extending CORBA Interfaces with $\pi$-calculus
5. Checking protocols
6. Open Issues
7. Conclusions
2. The CORBA IDL: A case study

A simple E-commerce application:

interface AccountFactory {
    Account create();
};

interface Account {
    exception NotEnoughMoney {float balance; float requestedAmount};
    float getBalance();
    string deposit(in float amount);
    string withdraw(in float amount) raises (NotEnoughMoney);
};
interface Bookshop {
    struct BookRef {
        string ISBN;
        float price;
    };
    BookRef inStock(in string title, in string author);
    void order(in BookRef b, out account a, out string purchaseId);
    date deliver(in string purchaseId, in string rcpt, in string addr);
};

interface BookBroker {
    void add(in Bookshop b);
    oneway void remove(in Bookshop b);
    boolean getABook(in string author, in string title,
                      in float maxprice, in string addr,
                      out date when);
};
3. The polyadic $\pi$-calculus

- A process algebra with synch communications through channels
- Not only values but channel names can also be transmitted
- Semantics expressed in terms of a *reduction* system, and labeled transitions (*commitments*)

**Operators:**

- Sending values: $\text{ch!}(v)$
- Receiving values: $\text{ch?}(x)$
- Creation of fresh names: $(\overset{\sim}{z})P$
- Process composition: $| +$
- Matching operator: $[x=z]P$
- Specials: $\tau$ zero
• Main rule of communication in the $\pi$-calculus:

$$(\cdots + \text{ch!}(v).P + \cdots) \mid (\cdots + \text{ch?}(x).Q + \cdots) \xrightarrow{\tau} P \mid Q[v/x]$$

• Global choices are non-deterministic

• Local choices are expressed combining ‘tau’ and ‘+’:

$$(\text{tau}.P + \text{tau}.Q)$$

• In the polyadic $\pi$-calculus, tuples can also be sent along channels

• Extensions to the standard polyadic $\pi$-calculus:
  – Basic data types (lists, sets, ...)
  – Enriched matching operator, and the [else] construct:

$$( [G_1]P_1 + [G_2]P_2 + \cdots + [G_n]P_n + \text{[else]}P_0 )$$
Extending CORBA Interfaces with textual $\pi$-calculus

- **Modeling Approach**
  - Object reference $\mapsto$ one $\pi$-calculus channel
  - Method call $\mapsto$ ref!(m,(inArgs),(reply[,excep1,...]))
  - Method reply $\mapsto$ reply!(returnValue,outArgs)
  - Raising exceptions $\mapsto$ excep!(excepParams)
  - Object state $\mapsto$ Recursive eqs and process parameters

- **Syntactic sugar**
  - ref!(m,(args),(rep)) $\mapsto$ ref!m(args,rep)
  - ref!(m,(args),(ref)) $\mapsto$ ref!m(args)
  - ref?m(m,(args),(rep)).[m='op']P $\mapsto$ ref?op(args,rep).P
4. Extending the example IDLs with protocol information

protocol AccountFactory {
    AccountFactory(ref) =
        ref?create(rep) .
        (^acc)
        ( Account(acc,0) | ( rep!(acc) . AccountFactory(ref) ) )
    + [else]
        AccountFactory(ref)
};
protocol Account {
    Account(ref,balance) =
        ref?getBalance(rep) .
        rep!(balance) .
        Account(ref,balance)
    + ref?deposit(amount,rep) .
        (^receipt) rep!(receipt) .
        Account(ref,balance+amount)
    + ref?withdraw(amount,rep,notEnough) .
        ( tau .
            (^receipt) rep!(receipt) .
            Account(ref,balance-amount)
        + tau .
            notEnough!(balance,amount) .
            Account(ref,balance) )
    + [else]
        Account(ref,balance)
};
Yes, protocol information can be added to CORBA IDLs.

But now we have it.... What can we do with it?

- **What** to check?
- **When** to check?
- **How** to check?
- **Who** carries out the checks?
• Static analysis of ‘closed’ applications at compile/design time

• What can be checked?
  – Liveness and safety properties (e.g. absence of deadlocks)
  – Component Substitutability
  – Component Compatibility

• How to check?
  – Executing the components’ protocol descriptions, using \( \pi \)-calculus standard tools

• Who carries out the checks?
  – The application designer
Example of static checks

protocol User {
  User(ref,bookbroker) =
    (^author,title,price,addr)
    bookbroker!getABook(author,title,price,addr) .
    bookbroker?(yesorno,when) .
    zero
};

Appl() = (^ac) // AccountFactory’s address
    (^b1,b2) // Addresses of the two bookshops
    (^bb) // Book-broker’s address
    (^u) // User’s address
    (AccountFactory(ac) | Bookshop(b1,ac) | Bookshop(b2,ac)
     | BookBroker(bb,<b1,b2>) | User(u,bb) )

Deadlock-free test: Appl() $\xrightarrow{\tau^*}$ zero
Based just on the IDLs of the application’s components and the binds among them, they allow powerful interoperability tests prior to the components’ execution

However...

- They are useful for closed applications, but not so much for open applications in which the architecture is unknown, or the components may dynamically evolve
- Static analysis of $\pi$-calculus processes is an NP-hard problem
Run-time checks

- Dynamic analysis of ‘open’ applications, during the application’s execution time

- **What** can be checked?
  - Safety properties of applications (e.g., absence of deadlocks)
  - Component compatibility

- **How** to check?
  - CORBA *interceptors* reproduce the object run-time trace and check incoming messages against protocol specifications

- **Who** carries out the checks?
  - The object interceptors
Run-time checks

They eliminate the heavy burden of static checks, are tractable from a practical point of view, and are valid in open environments

However...

- They need a lot of accountancy by the interceptors
- Detection of deadlocks or other undesirable conditions is delayed until just before they happen
6. Concluding Remarks

• We have succeeded in extending CORBA IDLs with protocol info:
  – Description of both supported and required operations
  – Specification of partial ordering among them

• Benefits obtained:
  – Additional information available for component reuse
  – Some of the application’s architectural information is available
  – Improved interoperability checks
    . Component compatibility and substitutability
    . Safety and liveness properties of applications
    . Static and dynamic checks
Concluding Remarks (cnt’d)

• Object reference manipulations and client-server invocations have a good semantic matching with the $\pi$-calculus
  – Easy and natural modeling of object interactions
  – Formal support for reasoning about the applications
  – Standard tools available for the checks

However...

  – The $\pi$-calculus has a too low level syntax (despite the sugar)
  – Some static interoperability checks are too costly
Open Issues

- Adaptors
- Many-to-one substitutability
- Connection-time checks
- Conformance to specifications

Ongoing and future work

- Extensions of other models’ IDLs (COM, EJB, CCM, ...)
- Extend repositories and traders to deal with this sort of information
- Second version of our prototype
- Adding more semantic information to IDLs (Is it really practical?)
2nd Workshop on Object Interoperability

In Association with ECOOP’2000
Sophia Antipolis, France.
June 12, 2000.

http://webepcc.unex.es/juan/woi00/
protocol Bookshop {

    Bookshop(ref,bank) =
        (~rep) bank!create(rep) .
        rep?(account) .
        SellingBooks(ref,account)

    SellingBooks(ref,account) =
        ref?inStock(title,author,rep) .
        (~bookref) rep!(bookref) .
        SellingBooks(ref,account)
        + ref?order(bookref,rep) .
        (~purchaseId) rep!(account,purchaseId) .
        ref?deliver(pid,receipt,deliv,rep) .
        (~date) rep!(date) .
        SellingBooks(ref,account)
        + [else]
            SellingBooks(ref,account)
    }
}
protocol BookBroker {

    BookBroker(ref,bookstores) =
    ref?add(bs,rep) .
    rep!( ) .
    BookBroker(ref,bookstores++<bs>)
+ ref?remove(bs,rep) .
    BookBroker(ref,bookstores--<bs>)
+ ref?getABook(auth,title,price,addr,rep) .
    ( Buy(ref,auth,title,price,addr,rep,bookstores)
    | BookBroker(ref,bookstores)
    )
+ [else]
    BookBroker(ref,bookstores)

...
Buy(ref, auth, title, price, addr, rep, stores) =
    [ stores = NIL ]
    rep!(FALSE, NIL) . zero
    + [ stores = <bs>++dB ]
       bs!inStock(title, auth) .
       bs?(book) .
       ( [(book!=NIL)&&(book.price<=price)]
          bs!order(book) .
          bs?(account, pid) .
          account!deposit(book.price) .
          account?(receipt) .
          bs!deliver(pid, receipt, addr) .
          bs?(date) .
          rep!(TRUE, date) .
          zero
       + [else]
          Buy(ref, auth, title, price, addr, rep, dB) )
    );