A Prototype-based Approach to Distributed Applications

Tom Van Cutsem

Stijn Mostincx

Promotor: Prof. Dr. Theo D’Hondt
Advisors: Wolfgang De Meuter and Jessie Dedecker
Overview

• Context
• Prototype-based languages
  ▪ Pic%
• Concurrent languages
  ▪ cPico
• Distributed languages
  ▪ Advantages of prototypes
  ▪ dPico
• Future Work & Conclusions
Context: Ambient Intelligence

- Evolution towards increasingly smaller mobile devices embedded in the environment
- User is surrounded by a ‘processor cloud’ or Personal Area Network
- Programs and objects are able to move
- We need…
  - new design methodologies
  - adequate hardware support
  - runtime support, standards
  - new programming languages
Problem Statement

• Contemporary languages are not designed to write programs inhabiting complex, dynamic, flexible, open hardware constellations
• Need for a distributed (and concurrent) programming language
• Design of a distributed programming language based on the prototype-based OO paradigm
• Exploring the use of object-based inheritance in a distributed context
Language Overview

**Pic%**
- Prototype-based extension of Pico (D’Hondt, 1996)
- Small, minimal, exploratory object-oriented language
- Features **parent sharing**: two or more objects can inherit from (delegate to) the same parental object

**cPico**
- Concurrent extension of Pic%
- Features **active objects** and asynchronous communication
- Uses **parent sharing** to control mutable shared state

**dPico**
- Distributed extension of cPico
- Uses **active objects** as the unit of distribution
- Uses **parent sharing** to control mutable distributed state
Prototype-based Languages

- **Classless** object-oriented languages
- **Ex-nihilo** object construction and cloning
- Inheritance is either
  - **Delegation-based**: objects delegate incomprehensible messages to a ‘parent’
  - **Concatenation-based**: objects directly copy slots from a given ‘parent’ object
Parent Sharing in Pic%

```plaintext
window(width, height) :: {
    minimized: false;
    draw() :: { ... };
}

asBorderedWindow(border) :: {
    draw() :: { .draw(); drawBorder(); }
    drawBorder() :: { ... };
    capture()
}

asScrollableWindow() :: {
    draw() :: { ... };
    capture()
}

capture()

w: window(320, 240);
borderW: w.asBorderedWindow(blue);
scrollW: w.asScrollableWindow();
```

Overriding and super-sends

Create a view on an object, Object-based inheritance

Object creation
Concurrent Programming Languages

- Languages able to cope with concurrent program execution
- Concurrency creation: threads, active objects, forking, ...
- Concurrency control: synchronization
  - Conditional Synchronization

Languages:
- ABCL
- PScheme
- Java
- Obliq
- cPico
Concurrent Programming Languages

• Concurrency Paradigm ‘design space’:

  - The functional extreme
  - No shared state
  - Continuation-passing style
  - Asynchronous communication
  - Transparent synchronization

  - The imperative extreme
  - Controlled using shared state
  - Need for locks/semaphores/
  - Communication through shared data

Actors  cPico  Threads

- Synchronous and asynchronous communication
- Transparent synchronization
cPico: a Concurrent Pic% 

- An Integrative Approach (Briot et al., 1998):
  - Messages sent to active objects ...
    - are handled **asynchronously**
    - are processed **autonomously** by receiver
    - are processed **serially** ("one at a time")
Promises: Inter-object Synchronization

- Placeholders for the return value of an asynchronous message send
- Transparently become the return value
- Access to an “unfulfilled promise” blocks the accessor (“lazy synchronization”)
- Conditional synchronization achieved using “call-with-current-promise”
- Based upon futures:
  - Multilisp (Halstead, 1985)
  - ABCL/1 (Yonezawa et al., 1986)
  - Eiffel// (Caromel, 1989)
cPico: Design Issues

• Striving for simple consistent semantics
  ▪ No active objects in delegation chains

<table>
<thead>
<tr>
<th></th>
<th>... to Active Objects</th>
<th>... to Passive Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Passing</td>
<td>Asynchronous</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Delegation</td>
<td>Not Applicable</td>
<td>Synchronous</td>
</tr>
</tbody>
</table>

• Delegation versus synchronization problems
  ▪ Return to static scope
  ▪ Restricting visibility of variables
  ▪ Distributed state more susceptible to deadlocks
Example: Fibonacci

\[
\text{fibactor}(n) :: \{
\text{do()} :: \text{if (n<2, n, fibactor(n-1).do() + fibactor(n-2).do());}
\text{activate()}
\}\n\]

Active Object creation

Asynchronous Message Passing

Implicit Synchronization
Facilitating Parent Sharing

- Scope Functions allow controlled access to a parent’s variables:

  - `this(exp)`
  - `super(exp)`
  - `cloning(exp)`

  **Dynamic receiver**

  **Static parent**

  **Clone of object**

  Evaluation in...

  **atomic execution!**
Distributed Programming Languages

• An application can be distributed across several machines linked by a network
• Introduces several issues:
  ▪ Remote Method Invocation
  ▪ Serialization of RMI parameters
  ▪ Representation of Remote Objects
  ▪ Partial Failure Handling
  ▪ Object Lookup

Emerald
Borg
dPico
Obliq
dSelf
Argus
Why prototypes for distribution?

• Moving objects is more problematic in class-based languages:
  ▪ Moving an object requires its class to ‘move along’
  ▪ The transitive closure of the class’ superclasses must move along too
  ▪ ‘moving along’ classes implies class replication
    • What about class consistency? Requires class versioning
    • What about static class variables? Requires replication management
Why prototypes for distribution?

- Concatenation-based objects are dependency-free (no class or parent pointers)
- Delegation-based objects can share parents across virtual machine boundaries
  - This relation is explicit and thus transparent to the programmer, who remains in control
  - Shared parents can encapsulate distributed state and allow for broadcast communication (Dedecker et al., 2003; De Meuter et al., 2003a)
- Prototype-based languages have no trouble defining new ‘types’ of objects at run-time
dPico: a Distributed Pic%

- Transparent Remote Active Objects
- Extension mechanism based on Agora (Introducing several *types of methods*)
- Active objects can ‘publish’ themselves in ‘channels’ accessible by remote VM’s
- Very simple RMI parameter passing rules:
  - Active objects are always passed *by reference*
  - Any other dPico value is passed *by copy*
  - Remote references *always point to active* objects
RMI Problem: distributed parent sharing?

Parent is not shared but duplicated!
Solution: restructuring active hierarchies
Solution: restructuring active hierarchies

True distributed parent sharing!
Active Scope Functions

- Active counterpart of passive scope functions
- Operate asynchronously and immediately return a promise

Evaluation in...

\[ \text{atthis}(\text{exp}) \]
\[ \text{asuper}(\text{exp}) \]

- Dynamic active receiver
- Static active parent

atomic execution!
Example: a Distributed Chat Client

```plaintext
aview.chatServer(channel, maxClients) :: {
    clients[maxClients] : void;
    occupancy: 0;
}

aview.registerClient(nam) :: {
    `create a new chat client`
};

sendMsg(msg) :: {
    `send msg to all clients`
};

athis().register(channel)
```

transforms a regular method in a mixin method

code executed in an active extension of the receiver

registers receiver in a channel
Example: a Distributed Chat Client

```plaintext
aview.chatServer(channel, maxClients) :: {
  ...
  
aview.registerClient(nam) :: {
    receiveMsg(from,msg) :: display(from," ": ",msg,eoln);
    asuper(
      if (occupancy = maxClients,
          error("Sorry, channel is full"),
          clients[occupancy := occupancy+1] := athis()) )
  
  sendMsg(msg) :: {
    from: athis(nam);
    for(i:1, i <= occupancy, i:=i+1,
      clients[i].receiveMsg(from, msg));
  }
}
```
dPico: Strengths & Limitations

- Separation of active and passive entities leads to simple semantics
- Allows for true distributed object inheritance
- Primitive strong mobility due to first-class continuations
- RMI is expensive due to object graph serialization
- Message passing semantics are not totally location-independent

hotelObject.book(reservationObject)
Situating cPico and dPico
Future Work

• Using active objects to represent split objects
• Partial Failure Handling
  ▪ Dealing with asynchronicity and promises
  ▪ Modelling devices going “out of range”
• Incorporating multivalues
  ▪ Cloning family abstractions
  ▪ Classification abstractions
  ▪ Broadcast mechanisms
• Distributed Garbage Collection
Conclusions

• Design and implementation of
  ▪ prototype-based concurrent language cPico
  ▪ prototype-based distributed language dPico

• Parent sharing in a distributed setting
  ▪ Scope functions allow controlled access to shared distributed state
  ▪ Sharing of state without sacrificing encapsulation
  ▪ Separation of active and passive hierarchies ensures clean semantics

• Basis for future language engineering research in the field of Aml