Soul & Smalltalk

Just married

Kris Gybels & Maja D’Hondt
• Recap earlier presentation
• Logic + OO languages survey
• Improvements of SOUL+Smalltalk symbiosis
• Application: business rules
Linguistic Symbiosis of SOUL and Smalltalk

Johan Brichau
Kris Gybels
Roel Wuyts
Linguistic Symbiosis: definition

Programs written in one programming language can transparently call programs that are written in another language.

Advantages:
- Transparant replacement of parts of a program (with a program implemented in another language)
- ‘Multi-language’ programming
- Multi-paradigm programming

Towards Linguistic Symbiosis

- Identical paradigm programming languages
  - Fairly straightforward mapping
  - Mostly technical issues
  - E.g. RbCl, Agora, etc...

- Different paradigm programming languages
  - More complex mapping
  - Technical issues are equally hard
Querying Smalltalk Objects

- Predicate name <-> Message selector
- Predicate arguments <-> Message arguments
  - + extra last argument for return value

```ruby
if Array.new:(10,?instance), ?instance.at:put:(1,2,?)
```
Messages to SOUL modules

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>add:</td>
<td>(first argument bound)</td>
</tr>
<tr>
<td>add:with:to:</td>
<td>(all three arguments bound)</td>
</tr>
<tr>
<td>addwith:</td>
<td>(second argument bound)</td>
</tr>
<tr>
<td>addwith: to:</td>
<td>(2\textsuperscript{nd} &amp; 3\textsuperscript{d} argument bound)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

1 predicate defines $2^n$ messages ($n = \text{number of arguments}$)

result := Arithmetic add: 1 with: 2.

If Arithmetic.add:with:to:(1,2,?result)
Messages to SOUL modules

- Problem: returning of multiple variables
  - A query allows more than one free variable (which are the return variables)
  - A message only returns one ‘variable’

- Solution: return results in a container
  - E.g. a Smalltalk Collection
    
    ```smalltalk
define: add:with:to:
  {x,y,3} => Arithmetic addwithto: 3
```

```smalltalk
xyCollection := addwithto: 3
```
Messages to SOUL modules

- Problem: returning of multiple results
  - A query can have more than one solution
  - A message has exactly one result

- Solution A
  - Hide this from the Smalltalk programmer using an ‘implicit’ collection.

- Solution B
  - Return an explicit collection
<table>
<thead>
<tr>
<th>Language</th>
<th>Approach</th>
<th>Logic type</th>
</tr>
</thead>
<tbody>
<tr>
<td>R++</td>
<td>Single: ext. of C++</td>
<td>Simple data-driven</td>
</tr>
<tr>
<td>Kiev</td>
<td>Single: ext. of Java</td>
<td>Query-based</td>
</tr>
<tr>
<td>Prolog++</td>
<td>Single: ext. of Prolog</td>
<td>Query-based</td>
</tr>
<tr>
<td>CommonRules</td>
<td>Two: Java + Logic</td>
<td>Forward-chained</td>
</tr>
<tr>
<td>OPSJ</td>
<td>Single: ext. of Java</td>
<td>Forward-chained</td>
</tr>
<tr>
<td>Oz</td>
<td>Single: from scratch</td>
<td>?</td>
</tr>
</tbody>
</table>
**Rectangle**

monitored int height, width;

rule check_bounds {
    height < 0 || width < 0 ||
    height > 50 || width > 50
=>
    cerr << “illegal size”
}

int getHeight();
int getWidth();
void setHeight(int h);
void setWidth(int w);

<table>
<thead>
<tr>
<th>R++</th>
<th>Single: ext. of C++</th>
<th>Simple data-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean C++ expression</td>
<td>C++ statements</td>
<td>Autogenerated getters and setters that have to be used</td>
</tr>
</tbody>
</table>
Person

String name;
monitored int age;
monitored Set_of_p<Person> children;

rule all_adult_children {
    children.size() > 0 &&
    all Person * c @ children { c->age >= 21}
=>
    cout << name << “ has empty nest syndrome”;}

rule weird_child {
    Person * child @ children { child->age > this->age }
=>
    cerr << “Error: child older than parent”}

String[] numbers = {"one","two","three"}

public rule check(pvar String s) {
    s ?= "four" ;
    s @= numbers
}

public void foo(String s) {
    print((check(s) ? "" : "not ") + "possible");

    print("Possible values: ");
    foreach(PVar<String> ss = new PVar<String>;
    check(s)) {
        print(ss);
    }
}
class person
  mother(M) :- ...
  father(F) :- ...
  brother(B) :-
    sibling(B),
    B <- gender(male)
  sister(S) :- ...
end person

class jane
  inherits person
  name('Jane Hindley').
  gender(female).
  birth(1942).
  sibling(roger).
  child(lynette).
  child(walter).
end jane

class roger
  ...
end roger

class person
  inherits db
  attribute name, gender, birth.
  setName(N) :-
    name &-= N.
  setBirth(B) :-
    birth := B.
end person

person <- create(Jane, [name=’Jane Hindley’,
  gender=female, 
  birth=1942]).
...
Jane <- assertz(child(Roger)).
Jane <- assertz(child(lynette)).
**Customer**

```java
int purchaseAmount
String name
```

**CustomerSens : SensorManager**

testAllowedDiscounts(...) {
   ....
}

**CustomerManipulation**

```java
setDiscount(...) {
   ....
}
```

**Sensor and Effector definition**

**Attribute mappings**

If
```java
   customer(?c, ?pa) AND allowedDiscount(?c) AND then
   setDiscount(?c, 10)
```
Two: Java + Logic

CommonRules

Forward-chained

- Customer
  - Customer
    - int purchaseAmount
    - String name

- CustomerSens
  - testAllowedDiscounts
    - ...

- CustomerManipulation
  - setDiscount
    - ...

Customer

```
Class: examples.customer_purchase.Customer customer
purchaseAmount PurchaseAmount
name Name
End

Sensor: allowedDiscount
Class: CustomerSens
Method: testAllowedDiscount
BindingRequirement: (BOUND)
path: "examples.customer_purchase"

Effector: setDiscount
Class: CustomerManipulation
Method: setDiscount
path: "examples.customer_purchase"
```

customer(?c, ?p, ?n)

allowedDiscount(?c)

setDiscount(?c, ?d)
### CommonRules

**Customer**

- `int purchaseAmount`
- `String name`

**CustomerSens : SensorManager**

```java
void testAllowedDiscounts(...) {
    // ...
}
```

**CustomerManipulation**

```java
public void setDiscount(TermList termList) {
    ObjectTerm objTerm = (ObjectTerm)termList.ithTerm(0);
    Customer c = (Customer)objTerm.getObject();
    int discount = ((IntegerTerm)termList.ithTerm(1)).getInteger();

    c.setDiscount(discount);
}
```

### Two: Java + Logic

If

- `customer(?c, ?pa) AND allowedDiscount(?c) AND then setDiscount(?c, 10)`
Customer

int purchaseAmount
String name

CustomerSens : SensorManager

testAllowedDiscounts(...) {
    ....
}

CustomerManipulation

setDiscount(...) {
    ....
}

Main

public void main(String[] args)
{
    // create customer
    Customer co = new Customer();
    // create agenda
    Agenda agenda = new Agenda("discountRules.clp",
                              "discountAgenda",
                              12345,
                              "discountMethod.clp",
                              "attributeMap.map",
                              true);
    // create an instance of the engine
    IAE iae = new IAE(agenda, true, true, true);
    // trigger the engine using the Customer object instance
    iae.trigger(co);
    System.out.println(iae.getConclusionFactSet());
}
**TempReading**

String status;
int value;

int getValue();
String getStatus();
### Meal

BString drinkclass

### BString : BackchainingControl

Value will be determined when needed by a rule

---

Backward chaining simulation with special task object in working memory

```java
rule startrule2
    if {
        m isa Meal with (
            drinkclass of m != null;
        )
    } do {
        System.out.println("Drink class is " + m.view().drinkclass.getValue());
    }

rule r3
    if {
        currentTask isto DetermineValue
drinkclass of Meal meal;
        (alcoholIndicated of meal == true);
        (isformal of meal == true);
    } do {
        Meal mptr = meal.update();
        mptr.drinkclass.setValue("wine");
        currentTask.delete();
    }
```
Kernel (super)language

A paradigm is a style of programming in this language
Oz | Single: from scratch | ?
---|---|---

Based on concurrent constraint programming + mutable state

Values are:
- procedures
- mutable cells
- numbers etc.

\[ E ::= \text{false} | \text{true} | x = S \]

\[ \text{proc} \{ f \ a1 \ a2 \ldots \} \ E \ \text{end} \]

\[ \{ f \ a1 \ a2 \ldots \} \]

\[ \text{E1 \ E2} \]

\[ \text{local \ x \ in \ E \ end} \]

\[ \text{if} \ C1 \ [] \ldots \ [] \ Cn \ \text{else} \ E \ \text{fi} \]

\[ \text{or} \ C1 \ [] \ldots \ [] \ Cn \ \text{ro} \]

\[ \text{OR} \ C1 \ [] \ldots \ [] \ Cn \ \text{RO} \]

\[ C ::= x \ \text{in} \ E1 \ \text{then} \ E2 \]

(simple) constraints

procedure definition

procedure application

composition

variable declaration

conditional

disjunction

nondistribution conjunction

clause

x, f, a1 etc. are variables

S is an atomic value or variable
Oz | Single: from scratch | ?

Deterministic append

```oz
declare proc {Append Xs Ys Zs}
  case Xs of
    nil then Zs=Ys
    [] XIXr then Zr in Zs=XIZr
      {Append Xr Ys Zr}
    end
  end
end
```

Functional SG

```oz
declare fun {Append Xs Ys}
  case Xs of
    nil then Ys
    [] XIXr then XI{Append Xr Ys}
  end
end
```
proc \{Append Xs Ys Zs\}
  choice
    Xs=nil Zs=Ys
    \[
      \text{X Xr Zr in Xs=}X\mid Xr Zs=X\mid Zr \{Append Xr Ys Zr\}
    \]
  end
end

% 1. Define a new search query:
proc \{P S\} X Y in \{Append X Y [1 2 3 4 5]\} S=sol(X Y) end

% 2. Set up a new search engine:
E={New Search.object script(P)}

% 3. Get and display the first solution: (and others, when repeated)
local X in \{E next(X)\} \{Browse X\} end
class Counter
  attr val
  meth browse
    {Browse @val}
  end
  meth inc(Value)
    val <- @val + Value
  end
  meth init(Value)
    val <- Value
  end
end
Results

• Passing objects
  – Single language: (almost) automatically
  – Mapping to predicates

• Invoking messages
  – OO-language in predicates
  – Mapping

• Invoking queries/rule triggering
  – Rule base construction
  – Mapping
Improvements to SOUL

Messages to SOUL modules

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>add:with:to:</td>
<td>add: (first argument bound)</td>
</tr>
<tr>
<td></td>
<td>add:with:to: (all three arguments bound)</td>
</tr>
<tr>
<td></td>
<td>addwith: (second argument bound)</td>
</tr>
<tr>
<td></td>
<td>addwith:to: (2nd &amp; 3rd argument bound)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

1 predicate defines $2^n$ messages ($n =$ number of arguments)

result := Arithmetic add: 1 with: 2.

If Arithmetic.add:with:to:(1,2,?result)

Not: existing Smalltalk + existing SOUL
But: existing Smalltalk + suitable symbiotic logic language

Use Smalltalk syntax for both, mapping comes free
New Soul Syntax

member(?item, ?list)

member(?item, <?item | ?rest>).
member(?item, <?first | ?rest>) if member(?item, ?rest)
New Interpretation Process

?list containsSomethingRed if
?list contains: ?x & ?x isRed

No rule for isRed, so send as message to object in ?x

containsSomethingRed(?list) if
member(?x, ?list)
[ ?x isRed ]

Smalltalk term, send message isRed to object in ?x

Note: defining a rule for isRed would override message sending
?x isRed if
  ?x color = ?color &
  ?color wavelength = ?wl &
  ?wl isBetween: 650 and: 750
From Smalltalk to Soul

No method for ancestors so resolve as query

`?person ancestors = ?ancestor if
?person parent = ?ancestor`

`?person ancestors = ?ancestor if
?person parent = ?parent &
?parent ancestors = ?ancestor`

query solutions

collection of results for parameter after =
or FakeSingleItemCollection

ancestors

Soul programmer would want to write
?
p ancestor = ?a
or
?
p hasAncestor: ?a
Unbound Variables

Possible but not really necessary:

```
test
I m f I
self hasMother: x andFather: y
```

Invokes query

```
if hasMother: ?x andFather: ?y
```

Unnecessary because unnatural (paradigm leak)
Business Rules

- preferences, policies, decisions
- when certain conditions are satisfied
  - infer new knowledge
  - initiate actions
- logic reasoning system
  - represented using implications: antecedent => consequent
  - logic programming language
    - consequent is logical conclusion
    - backward chaining
  - production system
    - consequent is action recommendation
    - forward chaining
Survey of Business Rule Systems

- core application functionality in Java
- implementation in Java of logical reasoning system
- direct object manipulation in rules
- explicit and manual rule triggering
- • core application functionality in Java
• implementation in Java of logical reasoning system

1. if {
   ... ?l ...;
   ?i: ?l.incomeType();
} then {
   //do something...

2. IlrContext myContext = new IlrContext();
   IlrRuleset myRuleset = myContext.getRuleset();
   myRuleset.parseFileName("fish-promo-rules");
   myContext.fireAllRules();
Business Rules in an Online Store

ShoppingCart
- items
- addProduct:quantity

OnlineStore
- checkout: createOrder:

Customer
- name
- age
- shippingAddress
- billingAddress
- hasChargeCard
- location
- addToShoppingCart:quantity:
- totalPurchases

Product
- name
- price
- internationalShippingAllowed

Order
- customer
- items
- subtotal
- taxes
- shipping
- total
- orderStatus
- paymentStatus
- orderDate
- shippingDate
- deliveryDate
- shippingAddress
- billingAddress
- calculateSubTotal
- calculateTaxes
- calculateShipping
- calculateTotal
Business Rules in an Online Store

Product
- name
- price
- internationalShippingAllowed

Hardware
- brand

Computer
- space
- speed
- dvd
- hasDVD

AllInOneComputer
- screen

Monitor
- screen

Screen
- size
- resolution

Laptop
- weight
`Customer`

`addToShoppingCart: p quantity: i`

```plaintext```
(self canBuy: p) ifTrue: [

shoppingCart addProduct: p

quantity: i
```

`Customer`

`locationIs: s`

`^shippingAddress country = s`

`BRLayer`

`?customer canBuy: ?product if`

`?customer isLocatedInUSA.

?customer canBuy: ?product if not(?customer isLocatedInUSA) &

?product internationalShippingAllowed.

?customer isLocatedInUSA if

?customer locationIs: 'USA'

`Product`

`internationalShippingAllowed`

`^internationalShippingAllowed`
**OnlineStore**

createOrder: c

<table>
<thead>
<tr>
<th>o</th>
<th>l</th>
</tr>
</thead>
</table>
o := Order new customer: c;
  orderDate: Date today;
  items: c shoppingCart items;
  shippingAddress: c shippingAddress;
  billingAddress: c billingAddress;
  calculateTotal.
self addOrder: o.
c shoppingCart empty.
^o

**Order**

**calculateTotal**

self calculateSubTotal.
self calculateShipping.
self calculateTaxes.
self calculateShipping.
total := subtotal + shipping

**calculateSubTotal**

subtotal := 0.
sel items do: [ :el | ... ].
subtotal := subtotal -
  (subtotal * customer discount / 100)

**BRLayer**

?c discount = 10 if ?c premiereCustomer.
?c discount = 5 if ?c loyalCustomer.
?c discount = 0 if ?c ordinaryCustomer
BRLayer

?c discount = 10 if ?c premiereCustomer.
?c discount = 5 if ?c loyalCustomer.
?c discount = 0 if ?c ordinaryCustomer

?c ordinaryCustomer if
    not(?c loyalCustomer) &
    not(?c premiereCustomer).
?c premiereCustomer if
    ?c totalPurchases = ?p &
    greater(?p, 10000)
?c loyalCustomer if
    ?c hasChargeCard &
    not(?c premiereCustomer).
`BRLayer`

```log
?c multimediaComputer if
  ?c isKindOf: [BRules.Computer] &
  ?c hasDVD &
  ?c hasMultimediaScreen

?c hasMultimediaScreen if
  ?c allInOneComputer &
  ?c screen = ?s &
  ?s multimediaScreen.
?c hasMultimediaScreen if
  not(?c allInOneComputer) &
  ?c compatible: ?s &
  ?s multimediaScreen

?s multimediaScreen if
  ?s size = ?x &
  greaterOrEqual(?x,17)

?c compatible: ?s if ...
```
Conclusion

• UoD for SOUL becomes more OO
• Still style/paradigm differences to be resolved
• logic expressive but not most suitable for BR
• our focus: **transparent** and **automatic** connection, supported by SOUL
• concepts and technique portable