Rapid Prototyping and Incremental Evolution Using SLAM

RSP'03: Shortening the Path from (Formal) Specification to Prototype

Ángel Herranz   Juan José Moreno-Navarro

aherranz@fi.upm.es   jjmoreno@fi.upm.es

School of CS

Department of Languages and Systems, and SE

Universidad Politécnica de Madrid

Agenda

- Motivation.
- Advantages of FM.
- RAD and FM in RAD.
- SLAM-SL Flavour.
- Conclusions.
- Related Work

Strategic Motivation of Our Group

- Improve the acceptance of FM and declarative programming.
- Handicaps:
  1. Alien notations.
  2. Tool and technology maturity.
  3. Productivity and processes.
- The SLAM Project:
  - OO notation for design and programming.
  - Non-alien notation (close to *widespread* languages).
  - Research in and development of tools.
- How to address the third handicap.

Paper Motivation

- Agile and prototyping processes' risk?
  1. Agile and prototyping processes are less controlled.
  2. Lack of automated tools.
  3. Emerging requirements.
  4. The cost of the prototype.
  5. Prototype efficiency (reimplementation needed).
  6. Prototype perceived as the final product.
  7. Reliance on third-party components (false features).
  8. The testing stage (reliability, robustness and safety).
  9. Scalability.
- Question: how do FM address that risk?

Paper Motivation (cont'd)

- FM = *formal specifications* + *verified design*
  - Precisely specify every piece of software.
  - Prove that a software component meets the spec.
- How do FM address agile and prototyping processes' risk?
  1. FM introduce a rigorous discipline.
  2. Already operative tools.
  3. Inconsistencies can be early detected.
- In general specs are not executable but sometimes. . .
  - Interpreted (algebraic approach).
  - Animated (model based approach).
- In those cases we have a prototype!

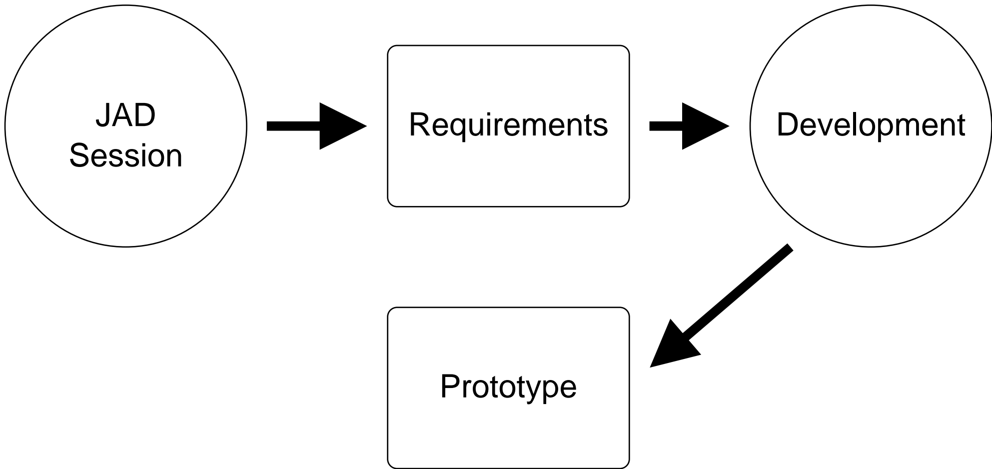
Paper Motivation (cont'd)

- Disposable vs. Evolutive.
- Classically, specs are used as disposable prototypes:

  FM are not productive!
- Would code synthesis be enough?
- Hypothesis: code synthesised should be human readable:
  - Programmers can understand and improve the prototype code.
  - The prototype can evolve.

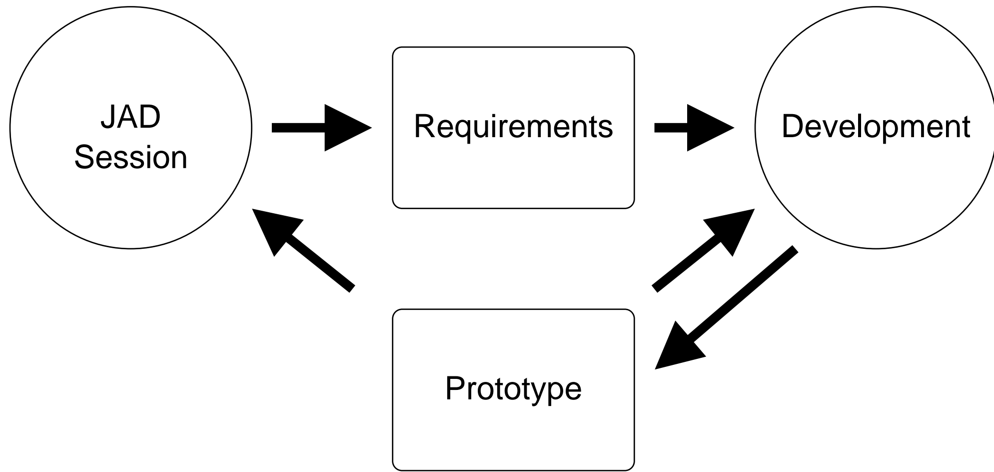
RAD

RAD

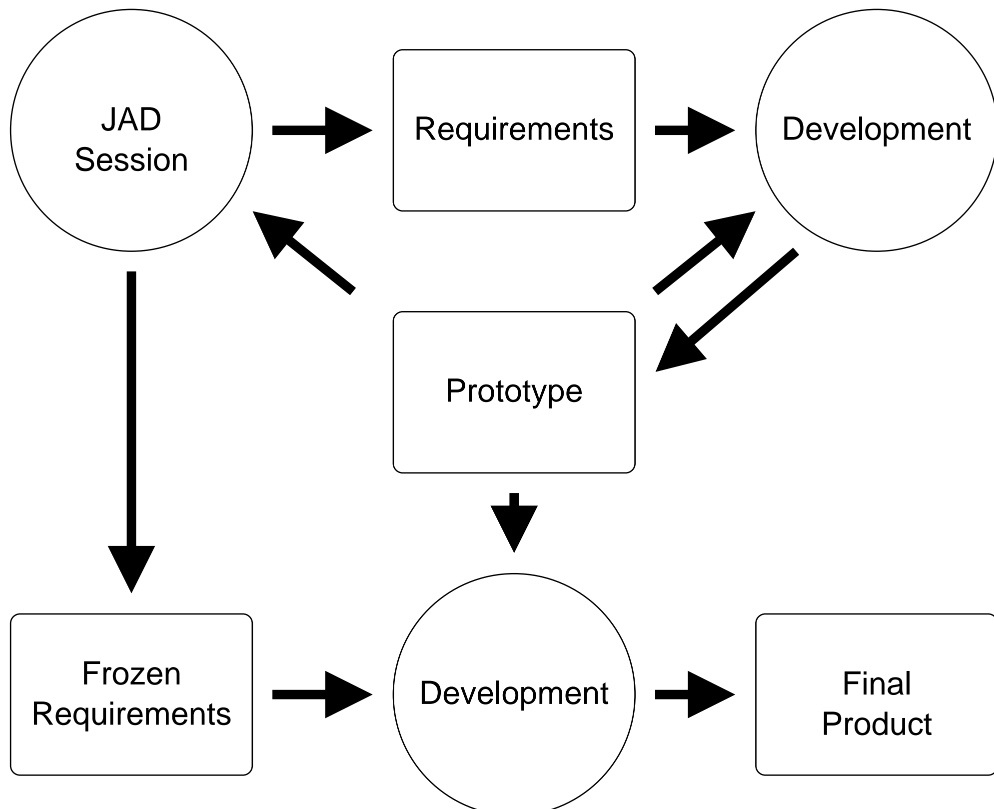


Rapid Prototyping and Incremental Evolution Using SLAM – p. 7/21

RAD

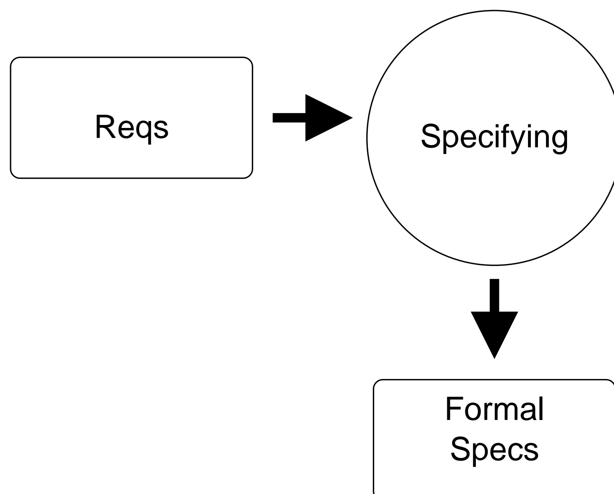


RAD

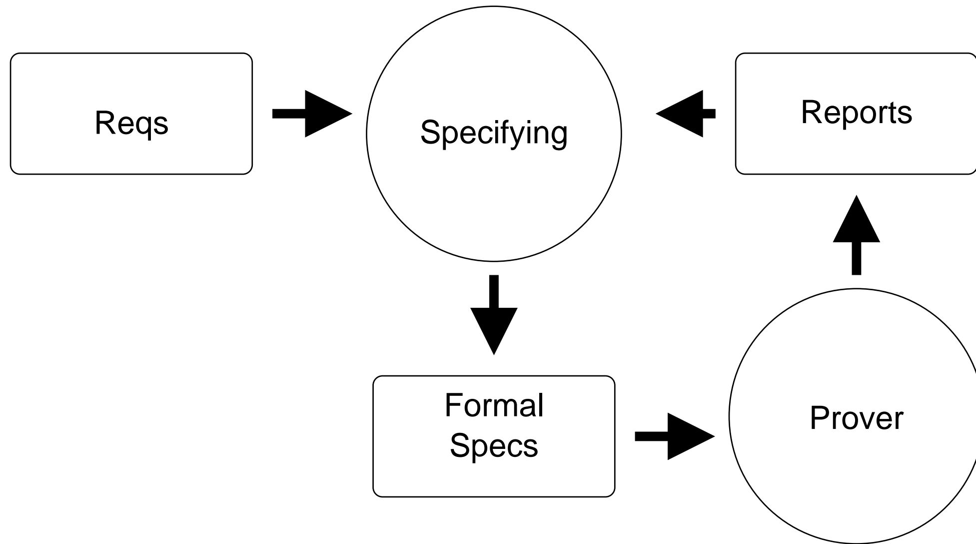


FM in RAD

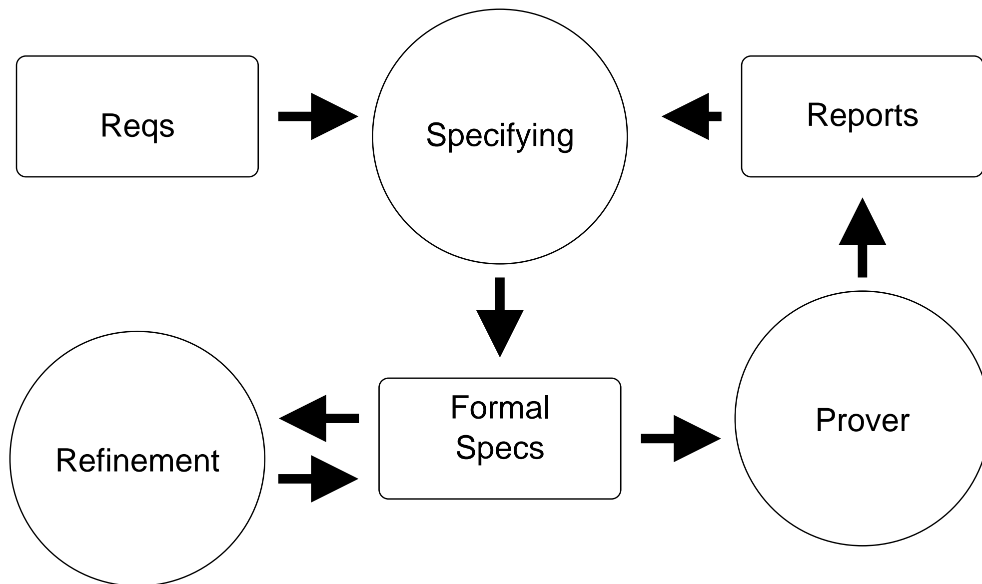
FM in RAD



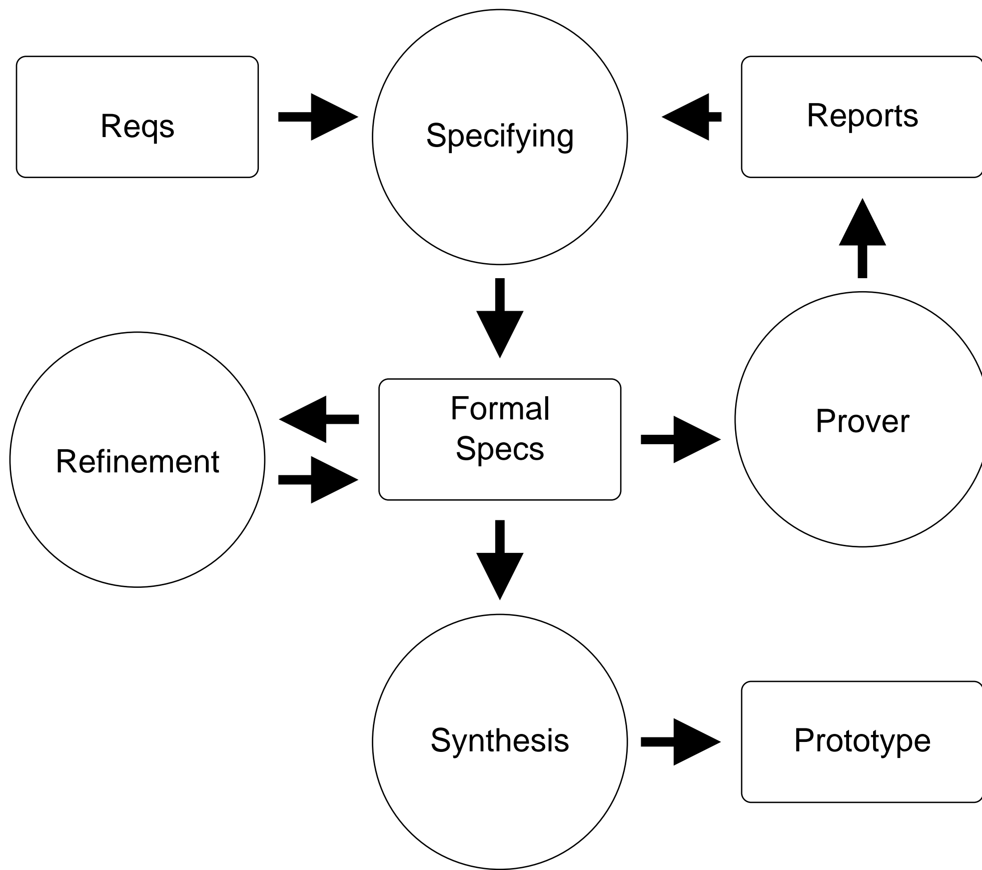
FM in RAD



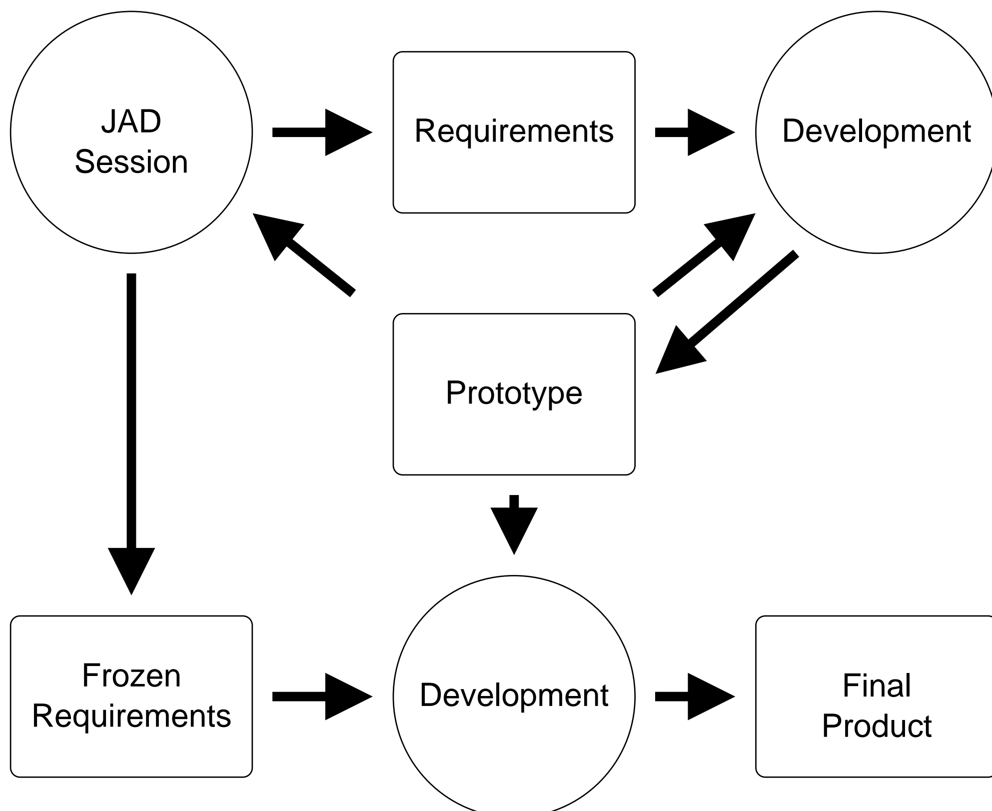
FM in RAD



FM in RAD



FM in RAD



Rapid Prototyping and Incremental Evolution Using SLAM – p. 8/21

Why SLAM-SL?

- SLAM-SL: an OO specification language valid for design and programming stages:
  - Abstract syntax dictated from *widespread* object-oriented languages (no alien!).
  - Trade-off between expressiveness and executability (human readable code synthesis).
- Lift programming practices to the world of formal specifications:
  - By using the OO as a metaphor.
  - By introducing design patterns and idioms in the synthesis.
- Allow programmers to improve generated code: evolutive prototype.

SLAM-SL

class Order

```
class Order {
  private OrderState orderState;
}

class OrderState {
```

- AST similar to Java or C#
- OO concepts: design patterns (State)

SLAM-SL

```
class Order

state pending (
  customer : Customer,
  product : Product,
  quantity : Positive)

state delivered (
  customer : Customer,
  product : Product,
  quantity : Positive,
  payment : Transfer)
```

```
class Order {
  private OrderState orderState;
}

class OrderState {

  private Customer customer;
  private Product product;
  private int Quantity;
}

class PendingOrderState
  extends OrderState {
}

class DeliveredOrderState
  extends OrderState {
  private Transfer payment;
}
```


SLAM-SL (cont'd)

```
class Color

class Point
```

SLAM-SL (cont'd)

```
class Color

class Point

class ColoredPoint inherits Color, Point
```

- Multiple inheritance.

SLAM-SL (cont'd)

```
class Color

class Point

class NoRedColoredPoint inherits Color, Point
invariant not self.is−red
```

- Multiple inheritance.
- Invariant from which checks are synthesised.

SLAM-SL (cont'd)

class A

Next spec is an instance method of A

SLAM-SL (cont'd)

class A

method m (T_1 , ... , T_n) **:** R

Method signature

SLAM-SL (cont'd)

class A

method m (T_1 , ... , T_n) **:** R

pre $P(self, x_1, \ldots, x_n)$
call m (x_1 , ... , x_n)
post $Q(self, x_1, \ldots, x_n, result)$

Method spec:

$$\forall s, x_1, \ldots, x_n (P(s, x_1, \ldots, x_n) \Rightarrow Q(s, x_1, \ldots, x_n, s.m(x_1, \ldots, x_n)))$$

SLAM-SL (cont'd)

class A

method m (T_1 , ... , T_n) **:** R

pre $P(self, x_1, \ldots, x_n)$
call m (x_1 , ... , x_n)
post $Q(self, x_1, \ldots, x_n, result)$

chk $T_1(self, x_1, \ldots, x_n, result)$
...
chk $T_m(self, x_1, \ldots, x_n, result)$

Extra properties for debugging

SLAM-SL (cont'd)

class A

method m (T_1 , ..., T_n) : R

pre $P(self, x_1, \ldots, x_n)$
call m (x_1 , ..., x_n)
post $Q(self, x_1, \ldots, x_n, result)$

chk $T_1(self, x_1, \ldots, x_n, result)$
...
chk $T_m(self, x_1, \ldots, x_n, result)$

sol $S(self, x_1, \ldots, x_n, result)$

Solution: *an executable postcondition* ($S \Rightarrow Q$)

SLAM-SL (cont'd)

```
method sort
call sort
post result.sorted and self.permutationOf(result)
```

A very abstract specification but there is information to produce debugging code (RT checks at least).

SLAM-SL (cont'd)

```
method sort
 call sort
post result.sorted and self.permutationOf(result)

chk  [].sort == []
chk  forall x : Integer with [x].sort == [x]
```

Some checks

SLAM-SL (cont'd)

```
method sort
 call sort
post result.sorted and self.permutationOf(result)

chk  [].sort == []
chk  forall x : Integer with [x].sort == [x]

sol   if self.isEmpty
     then result.isEmpty
     else result = self.tail.sort.insertSort(self.head)
```

An algorithm

SLAM-SL (cont'd)

- In standard logic, the meaning of a quantified expression $\forall x \in C(P(x))$ is

$$true \wedge P(x_1) \wedge P(x_2) \wedge ...$$

  with each x_i in C.
- The quantifier $\forall$ determines the *base value* $true$ and the binary operation $\wedge$.

SLAM-SL (cont'd)

- In SLAM-SL we have extended quantified expressions with the following syntax:

  q `quantifies` $e(x)$ `with` x `in` d

  Where q is a *quantifier* (a binary operation $\otimes$ and a starting value b), d is a collection x is the variable the quantifier ranges over, and e represents the function applied to elements in the collection previous to computation:

$$b \otimes e(x_1) \otimes e(x_2) \otimes e(x_3) \otimes \ldots$$

SLAM-SL (cont'd)

```
sum quantifies x * 2
  with x in empty.add_to_front(1).add_to_front(2)
```

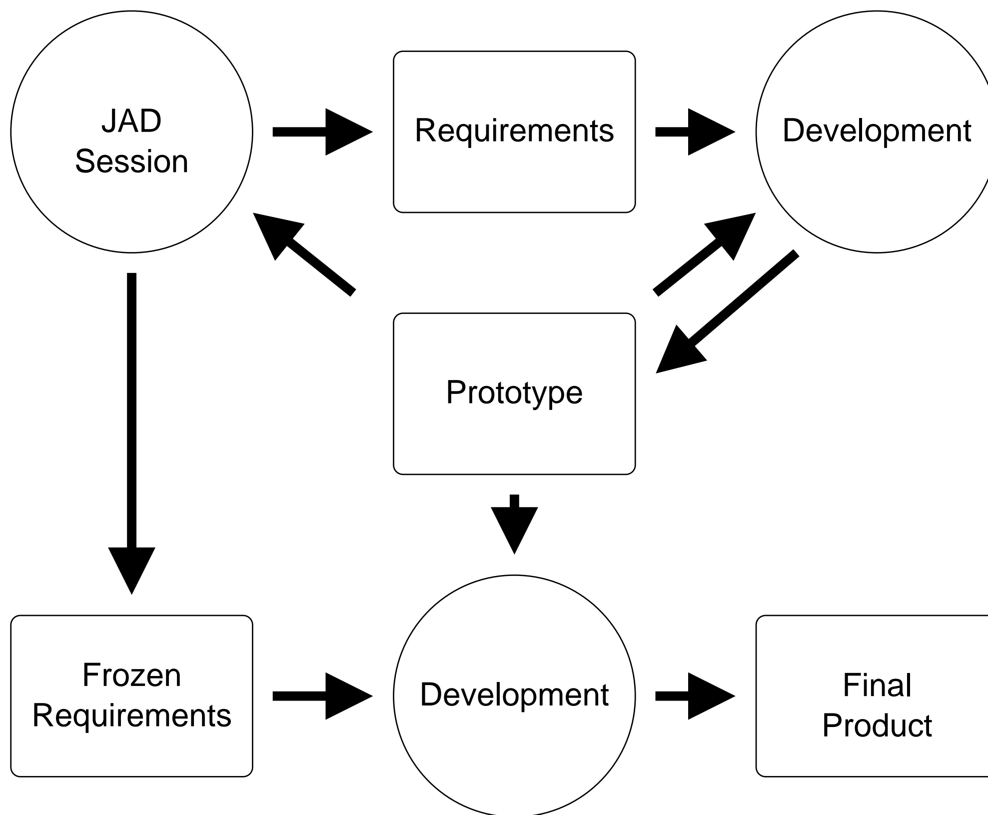
```
Sum q; { q = Sum.sum; }
Collection c;
{
  List l = List.empty();
  l.addToFront(new Integer(1)); l.addToFront(new Integer(2));
  c = l;
}
Integer r; Integer e;

Seq s = c.traversal();
for (int i = 0; i < s.length; i++) {
  e = new Integer (2 * (Integer)s.elementAt(i).intValue);
  q.next(e);
}
r = q.accumulate;
```


Conclusions

- FM and RAD synergy:
  - FM productivity can be improved if integrated in prototyping processes.
  - Prototyping risk can be reduced with SLAM:
    - Non-disposable prototypes.
    - Quality artifacts.
- Two exploratory papers about FM in agile and prototyping processes (the other in XP'03).
- Problem: generative approach vs. compositionality approach.

Conclusions (cont'd)



Future Work

- Status of SLAM:
  - Started in 2001. No environment.
  - SLAM-SL is semantically stateless.
  - Compiler:
    - Parser and typechecker.
    - Java code synthesis.
- Development of an environment and integration:
  - Tracking hand coded modifications.
  - Refinement.
  - Proving.
  - Industrial practices (2 contacts).
- Research in semantics and proving tech.

Final Remarks

- Prototyping with formal specifications is not new: VDM, Z, B and Maude.
- But non-disposable prototypes are not considered.
- FM adequacy has been demonstrated for critical systems.
- The example of the development of software for controlling the 14th line of the Paris Underground is spectacular.

Thanks

- The SLAM Project:

  `http://lml.ls.fi.upm.es/slam`

- Contact us:

`mailto:aherranz@fi.upm.es`
`mailto:jjmoreno@fi.upm.es`
`mailto:noelia@lml.ls.fi.upm.es`