New research issued on Component and Service Oriented Software : *Towards new ways of creating Software Intensive Systems*

Juan José Moreno-Navarro
Universidad Politécnica de Madrid
jjmoreno@fi.upm.es

Software basado

en Componentes

Curso 2006-2007

Summary

- **Software Intensive Systems:** new paradigm

  Difficult to develop with conventional techniques (complicate verification or validation, combining new and legacy code, etc.)

- **Software Architecture**

  Only solves a part of the problem.

- **Software Urbanism**

  New concept: Need for early planning knowing no complete details about future use, heterogeneous systems, advanced ways of combining software based on semantics interoperability.

- **We pose more questions than solutions: Provocative discussion.**

Software production nowadays

- **Software market in the world:** around 200 billion €

  Europe: 1/3 of this market, around 4.5% of GPB

  Spain: around 45 M€, 2.5% of GPB

  Industries bet for their digital conversion, and software is the key.

- **Software systems in all our environment:**

  They control essential aspects of our lives: banking systems, communications, transport, medicine

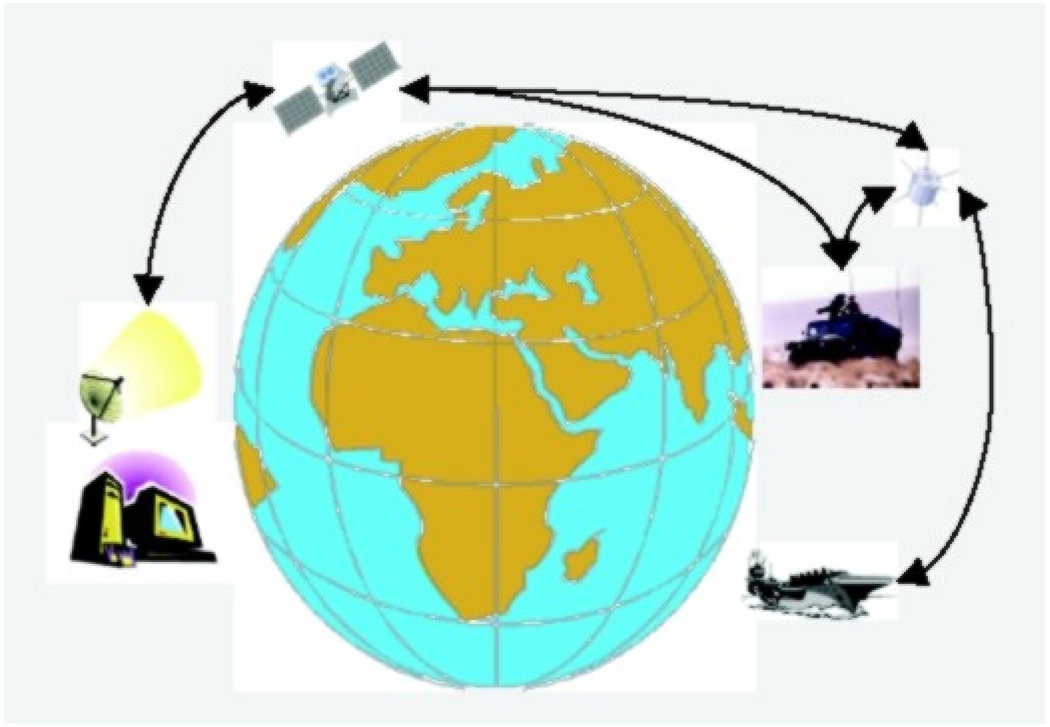
- **Continuous need for more and more software**

  Not always followed by an adequate foundational model.

Motivation: Information Society

The ever growing demand:

For *quality*, *reliability*, *safety*, *efficiency*, ...



Reality:

fragility, *unreliability*, *lack of trust*, ...

Several examples of severe Software failures

Software Failures

- **Well known examples**
  - Ariane 501 (ESA report)
  - Mars Climate Orbiter (Nasa 98)

    Problems mixing british and decimal measures
  - Wrong alarms about electricity in USA (First Energy infrastructure) with 3 victims
  - Accident of a American Marines flight mission in 2000 (no survivors)
  - 2000 effect
  - Euro problem

Software Failures

- **(More) Well known examples**
  - Impact of a Patriot missil on a friend plane in Irak
  - Severe failures in the telecommunication systems in France, summer 2004
  - Panama incident: Non reliable software used in a treating cancer device caused the death of patients for over-exposition to radiation. The authors have been condemned by a jury.
  - http://www.acm.org/technews/articles/2004-6/0322m.html#item17

•Effect of formal methods

–Detection of errors in X-21 y TCP

Gap between theory and practice

- **Software quality as a social problem: *The Economist* (June 2003)**

  http://www.economist.com/displaystory.cfm?story_id=1841081

  “To achieve predictable quality in software-making, just as in carmaking; the more you automate the process, the more reliable it is”.

  “The Panama incident causes some industry experts to consider the possibility that more stringent regulation of SW development is necessary”

Gap between theory and practice

- **Cost of failures:**

  USA 60 B$ per year, means a 0.6% of GPD. 80% of soft development cost invested in solving failures.

  A 80% of total cost of a software development are invested in identifying and correcting errors.

  NIST Study: Software Bugs Take Bite Out of Nation's Economy, www.nist.gov/director/prog-ofc/report02-3.pdf

The Babel Group

Software Intensive Systems

- **Software Intensive Systems:** *A system in which software is the dominant, essential, and indispensable element*
  - Complex programmable systems, based on embedded technolog (sensors and integrated controllers).
  - Integrated with the service-oriented computing paradigm.
    - Dynamic and evolving systems,
    - Adaptative and anticipatory behaviour.
    - Process knowledge and not only data

The Babel Group

Software Services

- **Autonomous, platform-independent computational entity that can be described, published, categorised, and dynamically discovered. Services can be dynamically assembled for developing massively distributed, interoperable, evolvable systems and applications.**

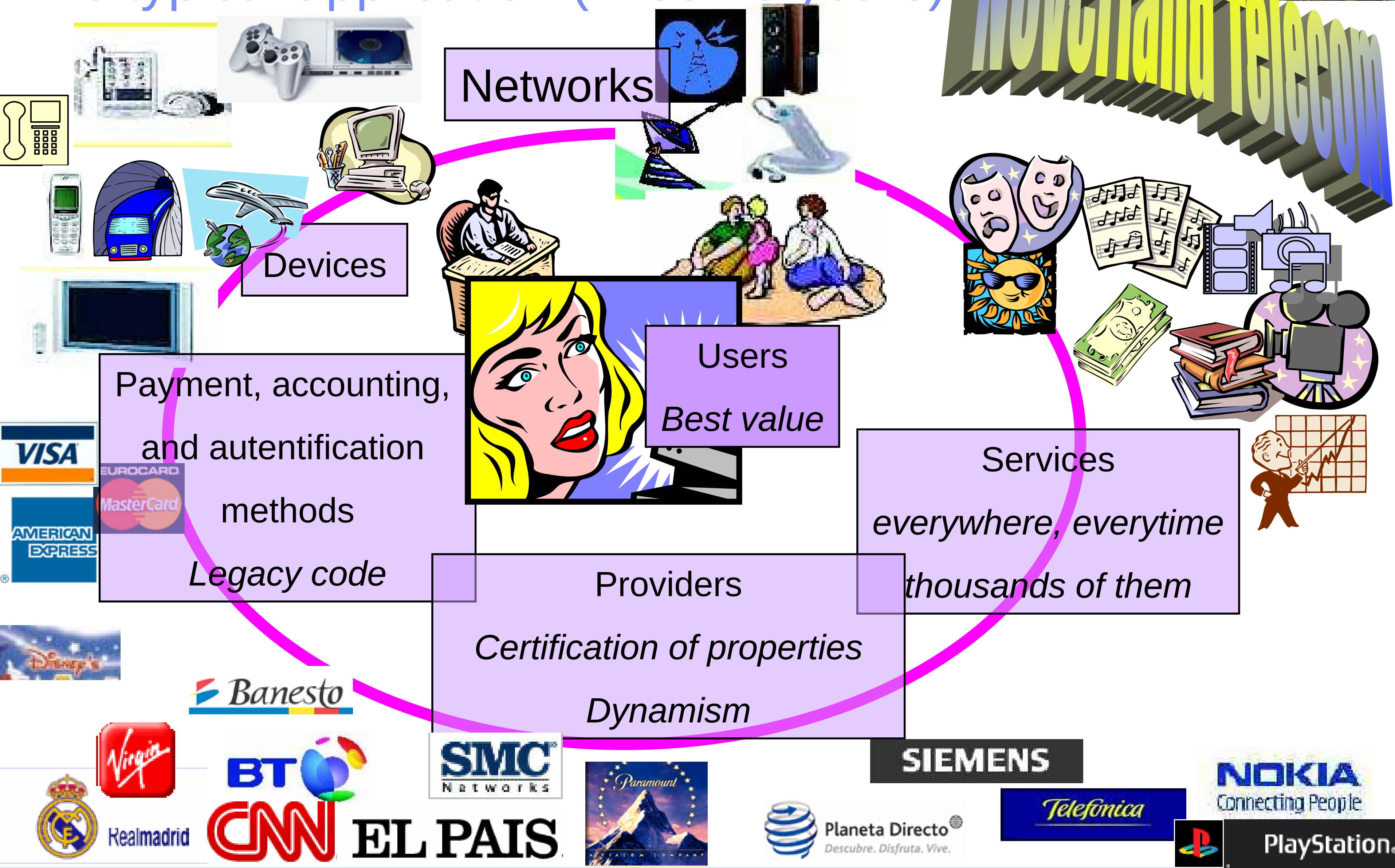
  Design

  –Future uses cannot be predicted

  –We need to dynamically ask services about the functionality it provides

The typical application (in some years)

Neverland Telecom



Networks

Devices

Users

Best value

Payment, accounting,

and autentification

methods

Legacy code

Services

everywhere, everytime

thousands of them

Providers

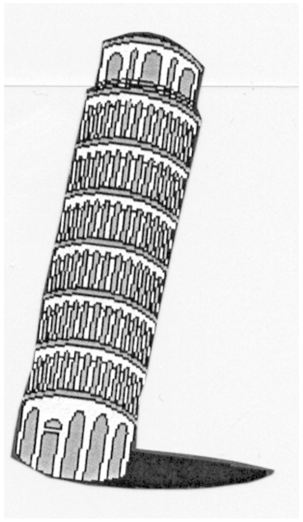
Certification of properties

Dynamism

SW Intensive Systems Validation/Verification

- **Conventional techniques are no longer useful:**
  - –Sensors and controllers: no room for patches, upgrades or new versions; software is part of long lived goods (cars, planes, mobile phones, etc.);
  - –A software system using software services cannot be verified:
    - we ignore the code.
    - it even cannot be tested as we can only use the software service in operation (and maybe paying for its use).
  - –Bertrand Meyer pointed out that exhaustively and formally verifying a software service that will be used hundred or thousand of times could be extremely worthy.

The problem with requirements



•Bad analysis of requirements is a general problem.

Responsible of 50% of failures:

- Standish Group 1995 - 350 USA, 8000 projects
- European Software Institute 1996, 3800 companies, 17 countries

•Severe cost for late corrections or new version (up to 200 times greater - Boehm, 1981)

New challenges, new models

A change in the models is a must.

- **Forcing the use of only those elements semantically rigorous in all the stages:**
  - Analysis (Specification Languages)
  - Modelling (UML?)
  - Design ([Formal] Languages for architecture description, design patterns -when formalized)
  - Implementation (Programming Languages with clear semantics: Declarative languages, Imperative languages with strict rules and well documented)
- **Tools for automatezing (part of) the process.**

Two ways of handling the problem of software reliability

1. **Providing tools for checking properties of existing code (model checking, abstract interpretation, verification tools, ...).**

   We depart for declarative programming and now applying those ideas for GCC (Eureka project).

2. **Providing a methodology for generating correct code from the early steps of a system:**

   The SLAM system.

New models

- **Software Architecture is not enough**
- **Specification of components need an essential be part of it.**

Architecture defined

- **Architecture n (1563)**

1. The art or science of building or constructing edifices of any kind for human use
2. The action or process of building
3. Architectural work; structure, building
4. The special method of 'style' in accordance with which the details of the structure and ornamentation of a building are arranged
5. Construction or structure generally
6. *The conceptual structure and overall logical organization of a computer or computer-based system from the point of view of its use or design; a particular realization of this*

Oxford English Dictionary

Software Architecture

- **The software architecture of a program or computing system** is the structure or structures of the system, which comprise:
  - software components,
  - the externally visible properties of those components, and
  - the relationships among them
- **Offer a global vision of the system**
- ***Dynamic Architectures:*** Basis for the systematic and autonomous evolution of software in execution.

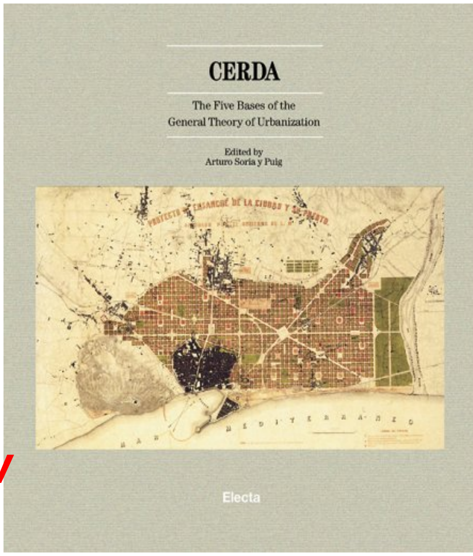
  Main ideas:
  - Separation of concerns between components and connectors.
  - Identify the desired characteristics of connectors for facilitating change
  - Use of architecture description languages for separating the structure of the system and the dynamic reconfiguration.

Software Architectures are not enough

- They just supply a part of the need of evolution of the systems. The structural approach does not necessary contribute to the capacity of evolution of the system, even more the system could be “too much structured”, avoiding the adaptation to new situations. E.g. many current systems are monolithic and centralized, not easy to adapt for their highly distributed management.
- Quality attributes are rarely part of requirement specification. Later when they appear in the development they are not adequately understood.
  - Architecture: Refers to a house
  - Urbanism: Refers to quarters and cities.

Urbanism

- **Urbanism:** comes from the Latin word *urbanitas* meaning « *from city, having the characteristic way of life of city dwellers* ».
- In 1867, **Ildefonso Cerda**, a Spanish engineer, published « *Teoría de la Urbanización* » where *urbanización* denotes the process of space arrangement, that could be planned or not, and the underlying laws. The job of the «*urbanizador*» is to discover hidden and unconscious laws in order to understand and to use them, knowingly, for conception and arrangement of constructed spaces. Barcelona's "El Ensanche" ***integrating old and new quarters*** to improve the city.

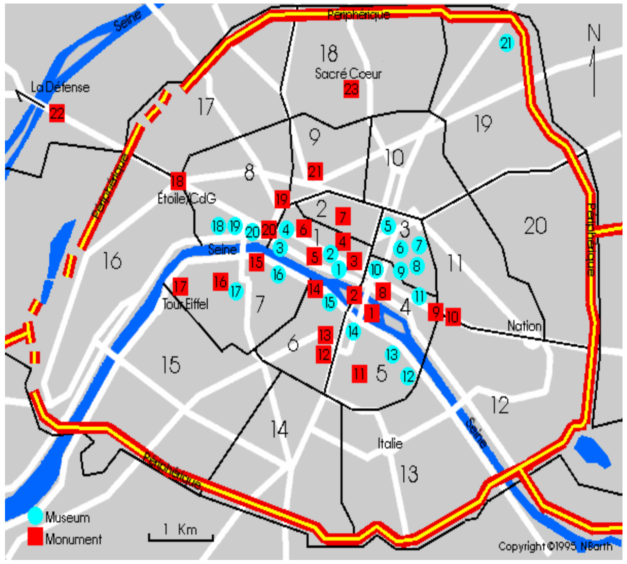

CERDA
The Five Bases of the General Theory of Urbanization
Edited by Arturo Soria y Puig
Electa

- Unlike the notion architecture which refers to the structure and style of an edifice, urbanism refers to the integration of the edifice into the environment according to the ***needs of the society***. Urbanism is defined as the art ***to reason and plan*** about urban agglomerations to adopt them to the needs of human beings who live and work there.

Urbanism

- **Hausmann** (prime minister Napoleon III) reorganized the city of Paris. The intention of this re-organization was to transform the middle age town of Paris into a modern city which fulfils new requirements like the growing traffic at that time, better supply infrastructures and others. For doing so, it was required to document the existing situation to provide information for later changes, i.e. a ***cartography of the city*** was needed. Such cartography is not a simple map of streets and blocks. It has to take different points of view into account, e.g. form outside, i.e. how to get into the city, or existing supply infrastructures such as water, electricity, and garbage collection and so on.

- The city cartography forms the basis for re-organization of the city. In Paris there exist an east-west axis which cuts the city and along which many important buildings and sights are located. Many avenues follow the direction to former town gateways (*portes*). Some quarters were entirely re-organized, others (*Le Marais*) remained almost unchanged.



Urbanism

- When re-organizing a city in the described way city architects and engineers are confronted with a lot of problems which concern many levels of the city such as water supply, heating or security. Large wide avenues are usually securer than small tiny lanes, because they are easier to supervise. A restructuring of the streets implicates a reconstruction of the supply network. These problems also require solutions when re-organizing a city. Some of these problems, so-called *Core-Level-Concerns* can be encapsulated into single functions, others not. These are called *System-Level-Concerns* or *Crosscutting-Concerns,* because they are interwoven with each other and concern the whole systems. Consequently, a city re-organization also needs solutions for these ***crosscutting problems*** which cut through all layers of the logical city structure.

- **Arturo Soria** proposed the *Ciudad Lineal*. The goal was to build a new and healthy city, improving the quality of life of its inhabitants. The main axe was the public transport and services as a sustainable cohesion element. The ***integration of the society*** in the city was introduced as an important element of urbanism.

Compañía Madrileña de Urbanización (C.M.U.)
Fundadora de la
Ciudad Lineal
(1894)

Software Urbanism: Approach

- **Software Urbanism:**

  Urbanism in computer science is an approach for a systematic evolution of software systems. It is a metaphor which establishes an analogy between building software and building cities that has been introduced to improve our insight on how to obtain a good system design and, especially, helping to re-engineer existing software systems. It is based on similarities between urban systems and large software systems:

  –Both possess a great complexity with numerous facets and corresponding view points reflecting the human organization which created them.

  –Both types of systems are characterized by a great dependency of their history, by a construction over the course of years depending of the constraints of the given time, and by various adoptions to different needs without re-designing the systems in a coherent manner.

Software Urbanism: Six axes

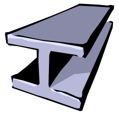
- **Cartography: Evolution and re-engineering**

- **New quarters: Autonomy and change**

- **Old quarters: Involving legacy code**

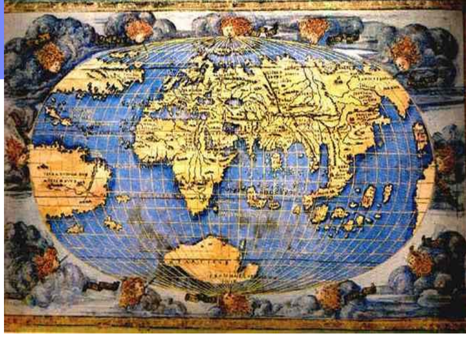
- **Crosscutting issues: Non-functional aspects**

- **Reasoning and planning: Semantical foundations**



- **Integrating society: Automatization**

Cartography: Evolution and re-engineering



- Need for a high capacity for evolution:

  Good software architecture $\Rightarrow$ Low cost for changes

  $\not\Rightarrow$ New uses in new contexts

  Need for models to support for system changing and evolution in aspects like business model, requirements, technological platforms, etc.

  – Separation of pure applicative parts from the software infrastructure (communications, interoperability layers, middleware, etc.)

  – Architecture standarization as part of service information. Allowing for service composition reasoning and (semi)-automatization.

  – Ability for change as part of modelling and design: e.g. automatic configuration of components compatible with the architectonical style.

Cartography: Evolution and re-engineering

- Urbanism is based on a functional decomposition which provides a functional cartography of the system with the following properties:
  - Hierarchical classification of activities discovered during the system analysis.
  - It reflects the services offered within the system. This is one of the pillars of a strategic alignment for supervising and controlling the system.
- **Derivation of the system cartography** is based on the modelling of different view points to make the system structure visible:
  - *External view point*: services provided by the system, the external user and the relation between them, the usage of the service, and the data exchanged.
  - *Internal view point*: processes and work flows running in the system.
  - *Informational view point*: Data exchanged between the processes as well as between the system components and specifies the applied data formats.
  - *Architectural view point*: building blocks and the structure of the system, i.e. its components, its subsystems, the hierarchical structure, and the infrastructures.
- The derived cartography is used to re-engineer or evolve the system in a systematic way: definition of the building blocks of the new system (or system version), re-design of data exchange infrastructure, creation of the new system architecture. In addition, the traceability of the changes must be ensured.

Cartography: Evolution and re-engineering

- Derivation of system cartography: iterative process.
  - *To chart the existing system* by determining the structural elements of the studied system, defining their roles and their associations.
  - *To consolidate the cartography* by validating the coherence of different view points and by simplifying them according to given rules.
  - *To define the evolution targets*: The evolution targets define the objectives of the evolution process or of certain evolution steps, respectively. The evolution targets determine the evolution or re-engineering strategy applied and possibly the migration paths towards that target. The evolution step will lead to a new cartography.

New quarters: Autonomy and change

- Autonomy of services and systems, and the ability for self-organizing.

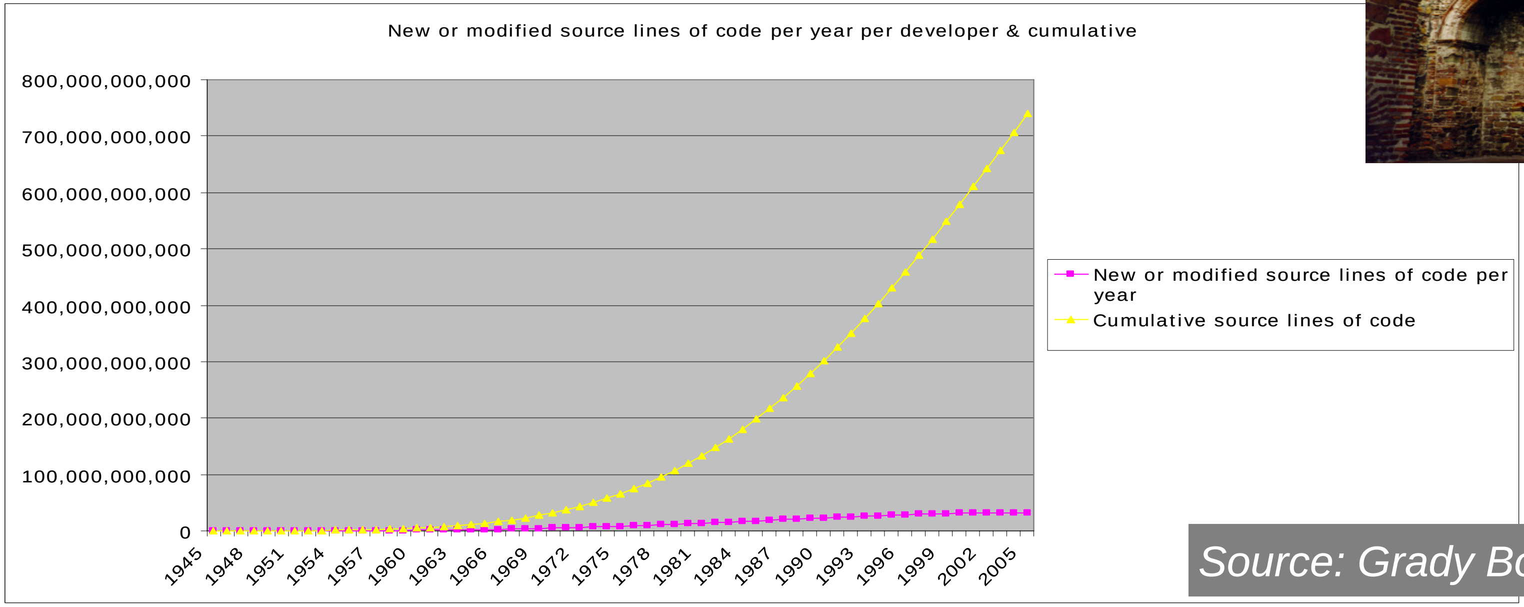
  Systems could take decision based on the context, and adapt to it (by means of re-organizing knowledge, models, code, ...)

  Autonomy is the capacity of “living in a society”, so it is intimately linked with urbanism.

- Need for strategies and methods for changing large distributed software systems and services and/or integrating new components and/or to migrating to others system architectures, allowing to extending and adopting existing systems to meet new demands and to make the system evolution more efficient.

Old quarters: Involving legacy code

- Hard topic



Source: Grady Booch

- First option: Leave the systems are they are, but specify in detail the interface, making the verification and validation tasks that were not done in their development.
- Second option: Audit the systems to identify those parts that need a re-engineering to fulfil needed properties.

Why God could create the world only in 7 days?
Because He has no legacy system to integrate

Crosscutting issues: Non-functional aspects

- Need for strategies for handling crosscutting issues: quality of service, security, resilience, extensibility, and manageability in urban strategies, allowing to supporting complete re-engineering processes.
- Non-functional aspects specification, representing quality attributes.
- Part of the modelling from the very beginning.
- Quantification of non-functional requirements. A single value is not usually enough. Idea: use of soft-constraints and ranges of values:
  - Allowed users: [10..15]
  - Fuzzy aggregation operators: conjunctive (less or equal the minimum), disjunctive (greater of equal than maximum), average (between minimum and maximum), negation.
    - Availability: Minimum operator
    - Efficiency: Product operator.

Reasoning and planning: Semantical foundations

- Need for developing methods and formal language concepts for modelling urban and architectural aspects allowing to describing different views, exchange scenarios, and architectural elements to support the re-engineering process in a more formal way.
- **Packed service:** It is important to know i) its behaviour, and ii) a certification of its quality properties.
- All the layers need rigorous and semantically defined elements: requirements, models, architectures, patterns, validation techniques, etc.
  - A must for reasoning with systems with huge number of services: a formal models of the behaviour, how it can be developed, validated, ...
  - A must for building tools that automates (part of) the process.
- **Semantic interoperability or *How* to understand *what* a software service offers:** Description languages (WSDL, syntactic, BPEL4WS not expressive enough) allow that applications have the ability to "talk" to each other but they do not "understand" what they are talking about.

  A formal specification language for web services is needed, with

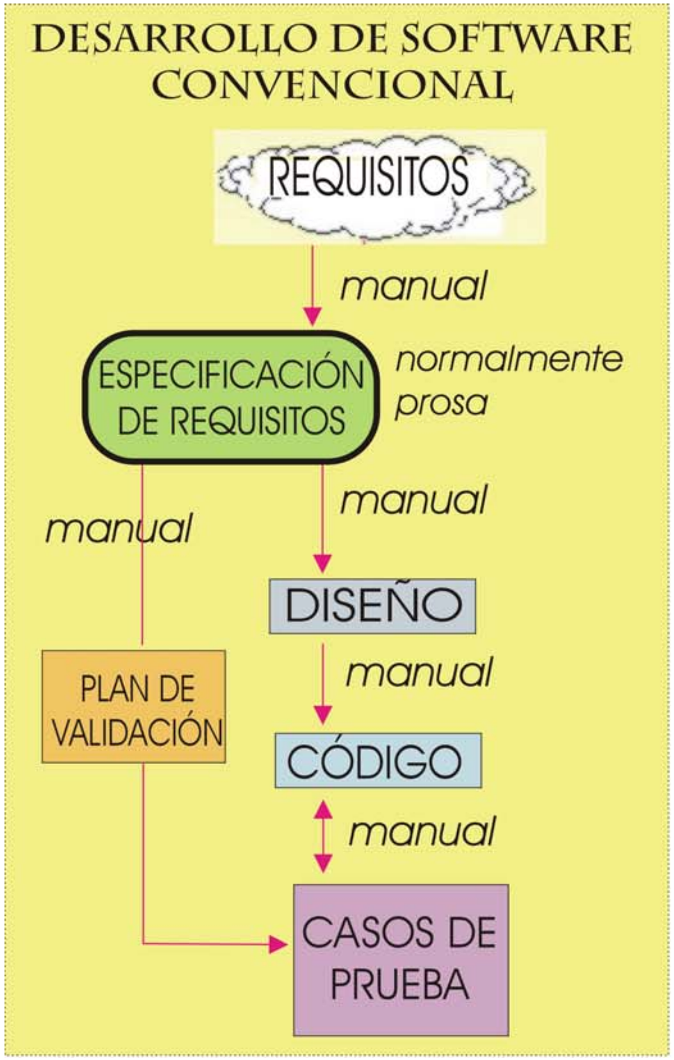
  i) no ambiguities, and

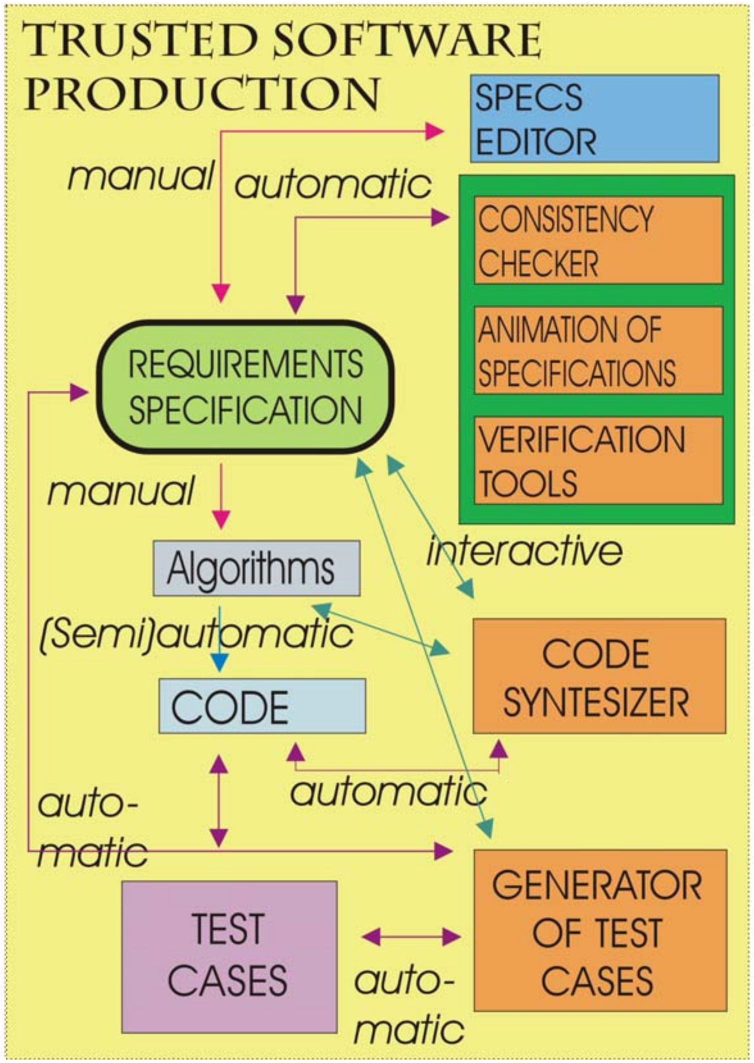
  ii) allowing automatic management.

Integrating Society: Automatization

- Need for automatizating (part of) the generation and software development:
- **TOOLS**:
  - Ensuring predictability of quality attributes (notably reliability) by construction.
  - Formal components is a must.
  - Decrement of development time.
  - Increment of reuse.
- A Software development process where
  - most verification and validation tasks are done in the analysis phase.
  - Code generation, consistency of specifications, testing, ... more and more automatized.

Integrating Society: Automatization



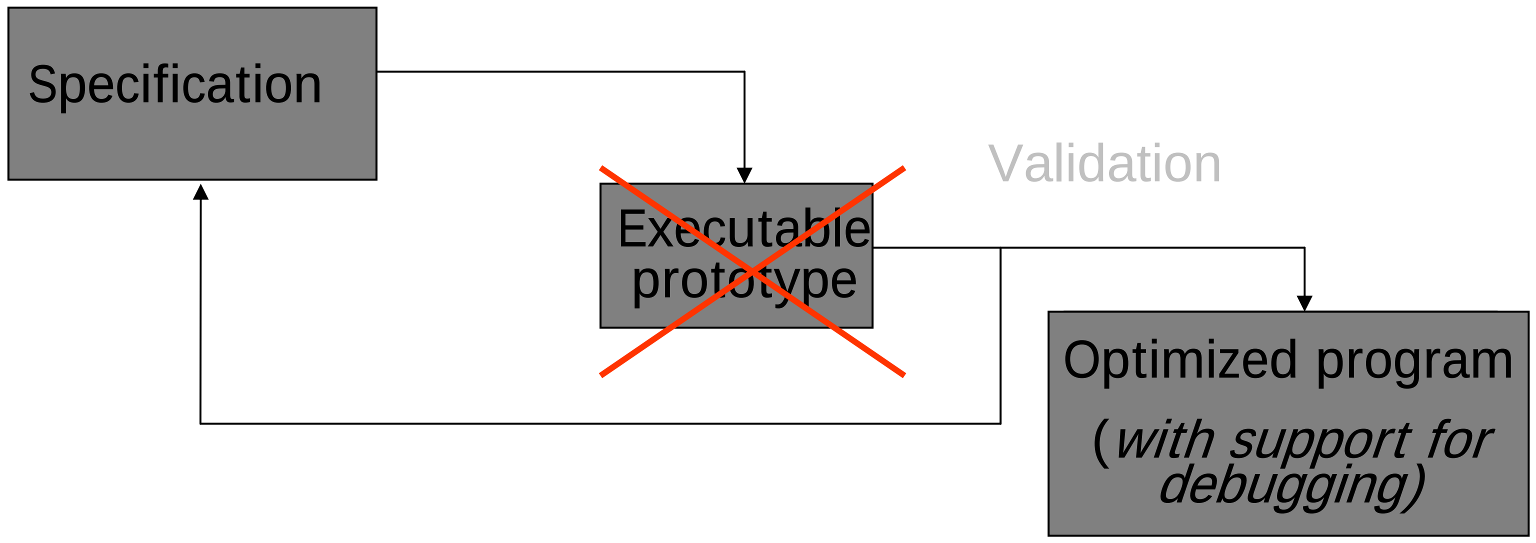


Production of a high quality specification of software behaviour (incl. non-functional properties)

Efficient, predictable and reliable correct code generation

Dealing with specifications: The SLAM project

- **Automatizing code generation**



The SLAM project

- **(Invisible) formal and rigorous methods for software development.**
- **Programming from specifications: From a description given in an specification language (SLAM) we are able to generated verified, readable and reasonably efficient code (Java, C++, ...).**
- **Environment support for reasoning and validating changes in code : Checkable post-conditions as assertion in the code.**
- **Many applications in different contexts.**
- **Useful for a complete systems, but the best use is for component specifications.**

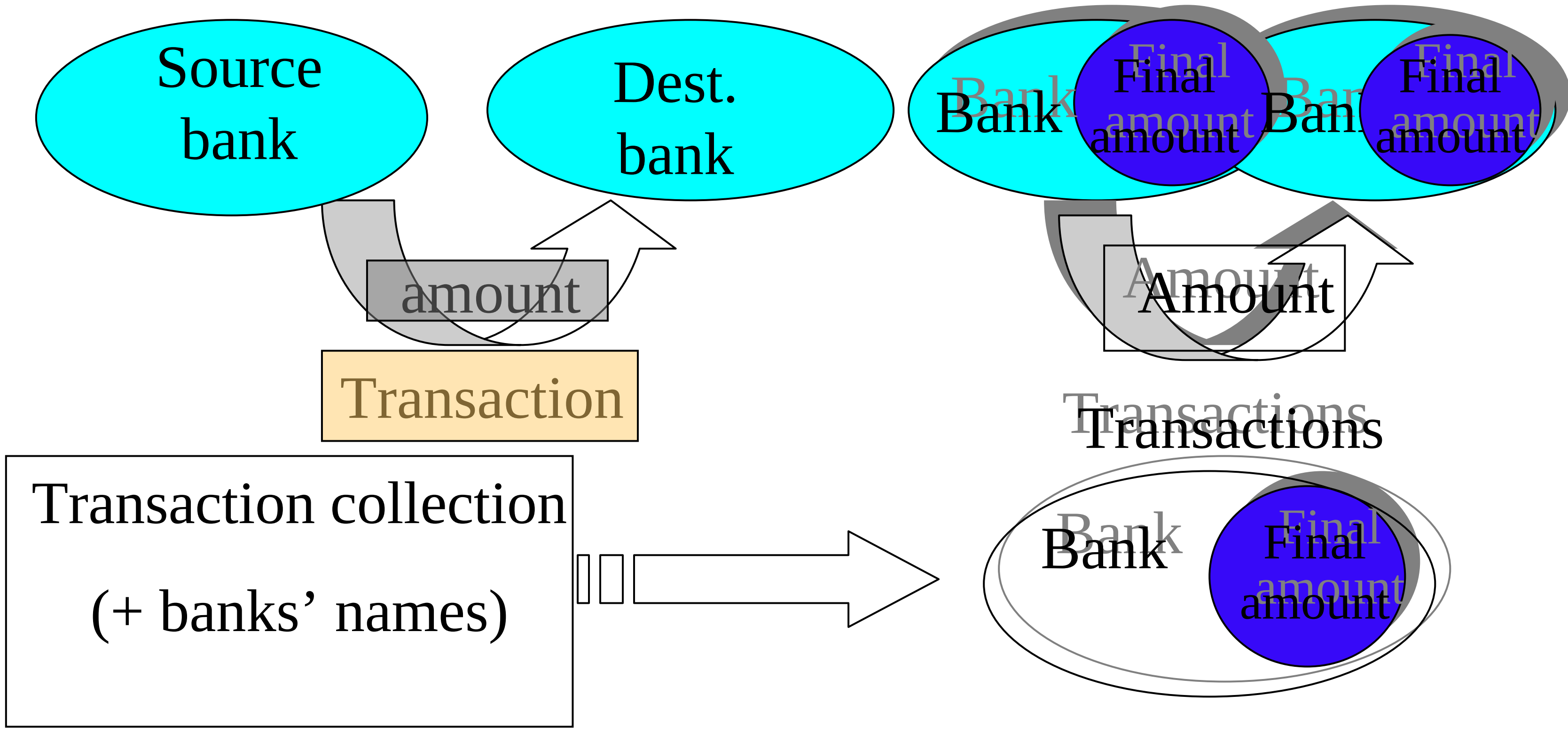
The SLAM project

- **An expressive specification language: *object oriented*. Compromise in order to being able to generate code.**
- **Translation schemas to imperative code (Java) based on (formal) program transformation techniques (coming from declarative programming).**
- **Key idea: To separate the specification (post-condition) from the mechanism to implement it (solution), but define both in the same language.**

http://babel.ls.fi.upm.es/slam

SLAM: An example

- **Bank transactions:**



SLAM: Bank transactions

```
class Transaction
  public state source : String, dest : String, amount : Real

class Banck
  public state banck (public name: String, public amount: Real)
    observer Name: String
      bank (nombre, total).Name = name
    constructor createBank : (String, Real)
      createBank (n, t) = bank (n, t)
```

Class declaration

Attribute declaration

Predefined construction for records

Data constructor definition

Operation/Method declaration

Explicit function definition

X: SLAM syntax

X: SLAM predefined

X: User defined

An Example: Bank Transactions

```
class TransactionCol
   state [Transaction]
   observer finalAmount: [String]: [Bank]
     pre:- length (names) > 0
     ctrans.finalAmount (names)
     post:-  check
     ( length (result) = length (names) and
       all i in {1..length(result)}.
         result(i).name = names(i) and
         result(i).total =
             (sum t in ctrans| t.source = names(i).t.amount) -
             (sum t in ctrans| t.dest = names(i).t.amount) )

     sol map n in names.
         makeBank (n,
                            (sum t in ctrans| t.source = names(i).t.amount) -
                            (sum t in ctrans| t.dest = names(i).t.amount))
```

Class declaration

Attribute/method declaration

Operation/method declaration

Predefined sequence construction

Precondition: *(Part) of the ... that must hold to ensure ... behaviour*

Postcondition: *Relates output and input values of the result*

Solution: *Effective method to compute the result*

Assertion annotation: *(Part) of the postcondition that can be dynamically checked*

X: SLAM syntax

X: SLAM predefined

X: User definition

Advantages of SLAM

- **In contrast with other specification languages, SLAM generates readable and modifiable code.**

  No throw-away prototype in the IRPP.

- **Code generation relatively easy provided that the target language supports some kind of object orientation (Java, C++, Haskell, Prolog).**
- **Key feature: O.O. + traversal + quantifiers → loops**
- **Formula classification: Executable specifications (valid for debugging) and solutions (valid for code generation).**

Compilation to Java

- Every class into a file.
- Private Java constructors (*C (...)*) + public class members that constructs objects (*makeC (...)*).
- Instance members for attribute access.
- Iterator technology for traversals: any traversable object returns an iterator.
- Quantifiers are translated by code schemes.
- Classes implement a serializable interface (automatic generation of read and write operations)

Compilation to Java

```
import java.io.Serializable;
import java.io.IOException;
public class Bank implements Serializable {
  private String name;
  private float total;
  private Bank (String name, float total) {
       this.name = name;
       this.total = total;    }

 public static Bank makeBank (String name,
                                          float total) {
     return new Bank (name, total);    }

  public String  name () {
   return name;    }
}
```

```
class Bank

state bank (String, Real)

constructor makeBank :
                    (String, Real)
makeBank (n, t) =
               bank (n, t)
observer Name: String
bank (name, total).Name =
                        name
```

Compilation to Java

Code Generation: *Program Transformation*

```
class List (Elem) inherits Collection
    state empty
    state cons (Int, Lista)
```

```
public class List {
  private static final int EMPTY = 1;
  private static final int IS_CONS = 2;
  private int state;
  private Object head;
  private List rest;
  private List () { }
  public static List empty () {
     List result;
     result :=new List ();
     result.state = EMPTY;
     return result;
}
```

Compilation to Java

Code generation: *Equational reasoning*

```
class List (Elem) inherits Collection
      state empty
      state cons (Int, Lista)
```

```
function traversal:List (Elem)→[Elem]
empty.traversal = []
(cons (c, l)).traversal = [c] + l.traversal
```

```
select x in l where prime (x)     ≡
select x in traversal (l) where prime (x) ≡
   ⎧ s(1)                                      if prime (s(1))
   ⎩ select x in s.suffix where prime (x)     otherwise          where s = traversal (l) ≡
   ⎧ y                                           if prime (y)
   ⎩ select x in traversal (r) where prime (x)  otherwise          where l = cons (y,r) ≡
   ⎧ y                                           if prime (y)
   ⎩ select x in r where prime (x)              otherwise          where l = cons (y,r) ≡
```

Tail recursive version.
Easily translated to a loop ⟹

```
x := l.head();
while(!prime (x)) {
  l := l.tail;
  x := l.rest ();     }
```

Compilation to Java

```
import java.util.Iterator;import java.io.Serializable;
public class TransactionCol implements Traversable, Serializable {
   private Transaction[] ctrans;
   private TransactionCol (Transaction[] ctrans) {
        this.ctrans = ctrans;
    }
    public static TransactionCol makeTransactionCol (Transaction[] ctrans) {
        return new CTransaction (ctrans);
    }
    public Iterator iterator () {
       return new Cursor (ctrans);
    }
```

Compilation to Java

```
public Bank[] finalAmount (String[] names) {
    Bank[] result = new Bank[names.length];
    int i;  float sum1, sum2;
    Iterator iterator;Transaction t;
    for (i = 0; i < names.length; i++) { // map
        iterator = this.iterator ();
        sum1 = 0;
        while (iterator.has_next ()) {     // sum
            t = (Transaction) iterator.next ();
            if (t.source().equals(names[i])) {
                sum1 += t.amount();             }
        }
        iterator = this.iterator ();
        sum2 = 0;
       ...
       result[i] = Bank.makeBank (names[i], sum1 - sum2);
    }
    return result;
}
```

Two traversals!

Hand-coded optimization

```
public Bank[] finalAmount (String[] names) {
    ...
    for (i = 0; i < names.length; i++) {
        iterator = this.iterator ();
        sum = 0;
        // ONE LOOP!!!
        while (iterator.has_next ()) {
            t = (Transaction) iterator.next ();
            if (t.source().equals(names[i])) {
                sum += t.amount();          }
            if (t.destination().equals(names[i])) {
                sum += t.amount();                  }
            //      ^-> This is a hand coded bug!! Must be -=          }
        result[i] = Bank.makeBank (names[i], sum);
    }
    return result;
}
```

SLAM: Extensions

- *Distributed* extension:
  - Focused on specification of component and (web) services.
- **One of the needs is to allow for a semantic description of software services.**
- **Current web services standards are still weakly defined from the semantical point of view.**
- **Main element: shared resource**
- **This information can be consulted by another component (by using the reflective capabilities).**

The Babel Group

Distributed Slam

Example

```
manager := mk_auctionManager

process Seller is
   awarded: Boolean := False;
   newPrice : Price;
begin
  <Init lot>
  manager.init_auction(lot.Item,lot.Price);
  loop
    <generate newPrice>
    <wait some minutes>
    awarded := manager.updatePrice(newPrice);
    exit when awarded;
  end loop;
end Seller;
```

Shared resource

Code using this service

Seller: Check if price updated until estable price

Distributed Slam

Example

Code using the service

```
process Buyer is
 fs : Item;  price : Price;
 newPrice : Price;  finalPrice : Price;
 awarded : Boolean := False; open : Boolean;
begin
 (open,fs,price):= manager.join_auction(open,fs,price);
 if open and <Interest fs price> then
  while not awarded and open loop
   (open,newPrice):= manager.hear_value(price);
   price := newPrice;
   if open and <Want_to_buy> then
    (awarded,finalPrice) := manager.bid;
   end if
  end loop
  if awarded then  <Item_Got!>  end if
 end if
end Buyer
```

Buyer: Check price and makes a bid.

Shared resources

Example

```
public resource AuctionManager

private state (market_ : Auction, open_ : Boolean)

public constructor mk_auctionManager
call mk_auctionManager=((0,0),false))

public action init_auction(Item, Price)
pre not open_
call init_auction (item,price)
post effect= (market_←setItemPrice(item,price),true)

public action join_auction:(Boolean,Item,Price)
call join_auction
post effect = (market_,open_) and
     result = (open_,market_←getPrice,market_←getItem)
```

Specified as a class

Actions for interfacing.
- *Atomic*
- *Resource modifying*

Shared resources

Example

```
public action hear_value (Price) : (Price, Boolean)
cpre market_←price < last_price or not open_
call  hear_value (last_price)
post effect = (market_, open_) and
     result = (market_←getPrice,open_)

public action bid : (Boolean,Price)
call bid
post open_ implies
      effect = (makert_,false) and
      result=(open_,market_←getPrice)

public action update_price(Price) : Boolean
call update_price (newPrice)
post effect=(market_←setPrice(newPrice),open_) and
     result = not open_
```

Concurrent precondition:

Conditional synchronization

Extending WSLD by adding semantics

- *WSLD*: Web services definition language:
  - Basically, merely syntactic
- Our proposal for extension:
  - Insert Slam-Sl rules by means of *WSLD* annotations
- Triple:

  <pre-condition, concurrent pre-condition, post-condition>
- *WSLD* 1.1: Annotations are added as extensible attributed
- WSLD 2.0: Annotations incorporated via new type elements

Annotated WSLD 2.0

Example

```
<types>
 <xs:schema
  <xs:element name="initAucRequest" type="tInitAucRequest"/>
    <xs:complexType name="tInitAucRequest">
      <xs:sequence>
        <xs:element  name="item" type="slam:Item"/>
        <xs:element  name="price" type="slam:Price"/>
        <xs:element  name ="self" type="slam:AucManager"/>
      </xs:sequence>
    </xs:complexType>
  <xs:element name="initAucResponse" type="tInitAucResponse"/>
    <xs:complexType name="tInitAucResponse"/>
      <xs:sequence>
        <xs:element name="effect" type="slam:AucManager"/>
      </xs:sequence>
    </xs:complexType>
   <...>
  </schema>
</types>
```

WSLD description of our service

Annotated WSLD 2.0

WSLD 2.0:
new element

Example

```
<interface  name = "auctionManagerInterface" >
 <operation name="initAuction" pattern="http://.../in-out" >
   <input messageLabel="In"  element="initAucRequest" />
   <output messageLabel="Out" element="initAucResponse" />

   <wsdls:slamRule>
    <wsdls:precondition = "not initAucRequest←self←is_open">
    <wsdls:cprecondition = "true">
    <wsdls:postcondition = "initAucResponse←effect←market_ =
      (initAucRequest←self←market_←setItemPrice(
              initAucRequest←item,initAucRequest←price))
     and initAucResponse←effect←open_ = true">
   </wsdls:slamRule>


 </operation>
    <...>
</interface>
```

New
subelements

Extending OWL-S with Slam-Sl

- *OWL-S*: Significant effort to add semantics to web services. Our proposal for extension:
- OWL-S does not indicates a preferred language (candidates are SWRL & DRS).
- Expressiveness problems: f.i. Outputs mean a type, no synchronization

OWL-S

```
public action hear_value (Price) :
                         (Price, Boolean)
cpre market_←price < last_price
     or not open_
call  hear_value (last_price)
post effect = (market_, open_) and
     result = (market_←getPrice,open_
```

Example

```
<process:AtomicProcess rdf:ID="HearValue">
 <process:hasInput>
  <process:Input rdf:ID="LastPrice">
    <process:parameterType rdf:resource="slam:#Price"/>
  </process:Input>
  <process:Input rdf:ID="AuctionIn">
    <process:parameterType rdf:resource="slam:#AucManager"/>
  </process:Input>
 </process:hasInput>
 <process:hasPrecondition rdf:resource="#HearValueCPrecondition"/
```

Property hasInput:

Input data types

Now: Precondition

OWL-S

Example

```
  <process:hasOutput>
   <process:UnConditionalOutput rdf:ID="#OpenOut"/>
    <process:paramerterType rdf:resource="&xsd;boolean">

    **<process:Output rdf:resource="HearValuePostCondition"/>**

   </process:UnConditionalOutput>
  </process:hasOutput>
</process:AtomicProcess>
```

Property hasOutput:

Output data types

Now: Postcondition

OWL-S

Example

```
<!— Conditions, Effects and Outputs —>

<slam:cprecondition rdf:ID="HearValueCPrecondition">
 <slam:expression = "#AuctionIn←market_←price < #LastPrice
                     or not #AuctionIn←is_open"/>
</slam:cprecondition>

<slam:output rdf:ID="HearValuePostCondition">
 <slam:expression = "#PriceOut=#AuctionIn←market_←getPrice
                    and #OpenOut=#AuctionIn←is_open"/>
</slam:output>
```

New element for concurrent conditions:

IMDEA Software

- A Comunidad de Madrid Research Institute around these topics.
- Part of the IMDEA network: Madrid Advanced Studies Institutes: *Water, Material sciences, Energy, Social sciences, Nano-sciences, Telematic networks and services, Mathematics, Food, Biomedicine*
- Model: Private non-profitable foundation, agreements with the Universities for sharing personnel, buildings, infrastructures, etc.

IMDEA Software

- **Integrated with Universities: UPM, URJC, UCM, CSIC**
- **Scientific excellence as main goal**
- **Focused on attracting researchers, outside the CAM**
- **Strong connection with industry: Telefónica I+D, Atos Origin, HP, BBVA**
- **Four initial research lines:**
  - Correctness by construction.
  - Tool-based rigorous modelling and validation.
  - New generation languages and optimizing and validating compilers.
  - Free software and its development methods.

Conclusion

- **Software intensive systems are growing:** a *complex* area. They are based upon services that were complex when they were built. This is an essential complexity that cannot be simplified. This complexity cannot be perceived by the user.
- A new discipline is needed: *Software Urbanism*.
- Key elements:
  - Capacity of evolution and autonomy of services further than the concrete application they were conceived for.
  - Enforce reuse.
  - Certification of quality properties, including non-functional aspects.
  - Solid semantical foundation in all the layers, specially in the specification of services.
- Preference for development methodologies ensuring quality and correctness by construction.
- **Software ubiquity** is not adequately understood. We need theories to design and reasoning with (complex) software. It is the only way for recon ciliate market software demands with its rigorous construction.

Making predictions is difficult, specially for the future

Niels Bohr (Physics Nobel Prize, 1922)

I am not enough young to know everything.

James M. Barry

¡Ozú! !Qué miedo saber tanto¡

Lola Flores

Thank you