Software Development for Non-Expert Computer Users

Teodor Rus and Cuong Bui
The University of Iowa
Iowa City, IA 52242, USA

May 15, 2010

The Goal of the Project

Provide Cloud Computing to

Masses of Computer Users!

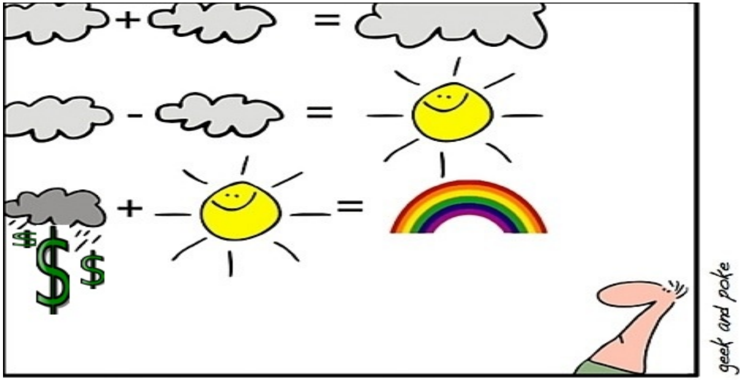
Rationale

Contradictory requirements of the technology:

1. Software development tools are dedicated to (few) experts!
2. Ubiquitous computing requires computers to be used by everywhere!
3. Efficiency requires CC systems to be used by masses!

These set unusual constraints on computer businesses.

Cloud Computing Algebra



A Cloud Based Software Business

- Find a Problem of Interest, (PI).

  Example PI: Make rain during drought!
- Develop a solution.

  Example: Develop a "Rain-Making Dust, (RMD)".
- Implement the solution in the cloud.

  Example: place RMD in the cloud.
- Wait for a rain while hoping to:

  Get dollars instead of water drops !!!

Problem Solving Process

Facts:

- During Problem Solving Process (PSP), domain experts USE the Natural Language of the Domain (NLD) as a problem solving tool.
- During Business Process (BP), domain experts DO NOT require business partners to know their NLD.

Examples

- Mathematicians use language of set theory to express problems, theorems, and proofs. They DO NOT require engineers to know set theory.

Examples

- Mathematicians use language of set theory to express problems, theorems, and proofs. They DO NOT require engineers to know set theory.
- Mechanical engineers use language of differential equations to model vehicle's behavior. But they DO NOT require drivers to know diff. equations.

Examples

- Mathematicians use language of set theory to express problems, theorems, and proofs. They DO NOT require engineers to know set theory.
- Mechanical engineers use language of differential equations to model vehicle's behavior. But they DO NOT require drivers to know diff. equations.
- Business people use language of forms to express business data processing. But they DO NOT require office-secretaries to know HTML.

Contrast

- Computer scientists use programming languages to express problem models and algorithms.
- Computer business REQUIRES computer users to program their applications.

Computer Based Problem Solving Process

- Unlike other domains, computer business REQUIRES COMPUTER USERS
  to be computer educated
  in order to program their computers.

Computer Based Problem Solving Process

- Unlike other domains, computer business REQUIRES COMPUTER USERS
  to be computer educated
  in order to program their computers.
- To align computer business to other businesses
  CBPSP needs be liberated from
  PROGRAMMING REQUIREMENT.

Liberate Computer User from Programming

This can be achieved by using

Natural Language (NL) as a

Programming Language (PL).

Feasibility

- NL is a human convention. It lacks the formalism required by PL implementation. But,

Feasibility

- NL is a human convention. It lacks the formalism required by PL implementation. But,
- PSP uses NLD during problem solving process.

Feasibility

- NL is a human convention. It lacks the formalism required by PL implementation. But,
- PSP uses NLD during problem solving process.
- Unlike NL, NLD can be formally specified. Thus,

Feasibility

- NL is a human convention. It lacks the formalism required by PL implementation. But,
- PSP uses NLD during problem solving process.
- Unlike NL, NLD can be formally specified. Thus,
- NLD can be used as a problem solving tool.

Why Using NLD with CC Systems?

1. CC hides the complexity of computer systems from their users.
   Hence, CC-s are appropriate for non-expert users.
2. Virtual machines in CC insulate different application domains.
   Therefore can be naturally dedicated to domains.
3. Infrastructure of CC is well defined.
   Hence, it should easily accommodate NLDs.

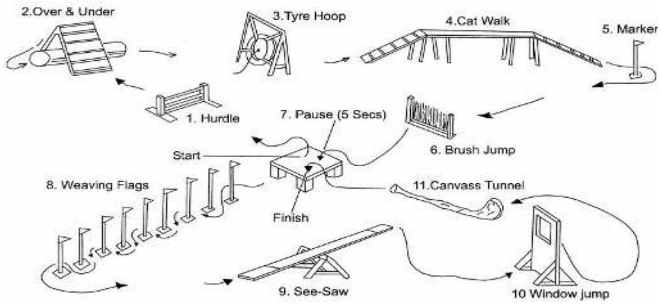
Benefits of Cloud Computing

Computer efficiency (speed + parallelism)

This Christmas, Santa goes green with new hybrid sleigh technology...
search ID: jhen29
© Original Artist
Reproduction rights obtainable from
www.CartoonStock.com
Professional driver on a closed course. Do not attempt this stunt
of this image, but they were bewildered and discombobulated. Sleigh and harness courtesy

More benefits

Business agility (adaptability to changes)



Efficiency of a CC Enterprise

CC system used by masses of computer users!

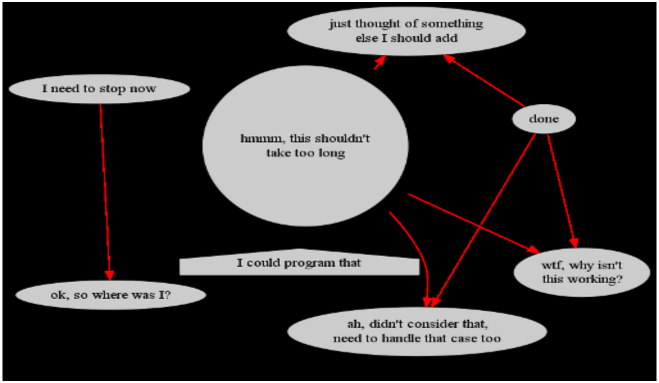
Computer Users

are of two categories:

1. computer educated (experts) and
2. non computer educated (non-experts).

Experts:

few who can program their computations.


just thought of something else I should add
I need to stop now
hmmm, this shouldn't take too long
done
I could program that
wtf, why isn't this working?
ok, so where was I?
ah, didn't consider that, need to handle that case too

Non-experts:

masses of users who can click buttons.

Conclusions

1. To be efficient a CC System needs to be used by non-expert users.

Conclusions

1. To be efficient a CC System needs to be used by non-expert users.
2. To be used by non-experts a CC system must be user convenient.

Conclusions

1. To be efficient a CC System needs to be used by non-expert users.
2. To be used by non-experts a CC system must be user convenient.
3. To be user convenient a CC system needs to be based on NLDs!

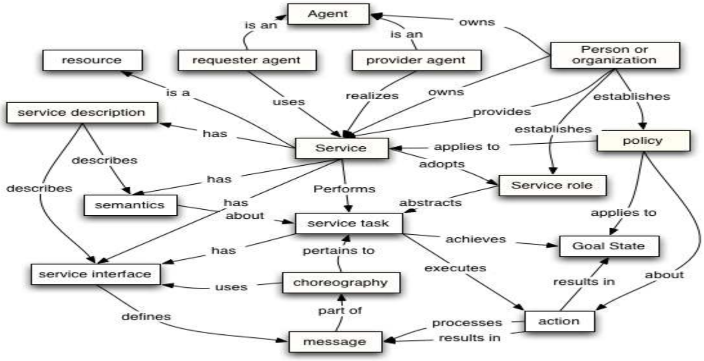
Can a CC system be efficient?

CC needs to address its convenience

Note: PSP fits naturally to everybody's activity

However

W3C standards SOAP, WSDL, UDDI,
(the bricks of current SaaS) are dedicated to experts.



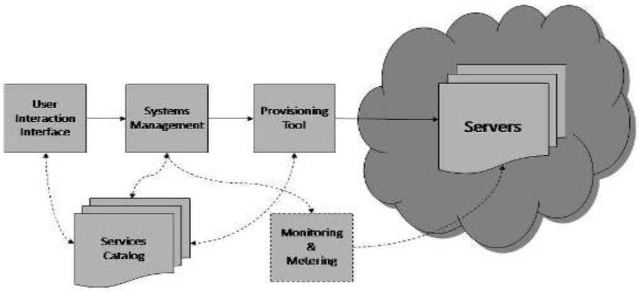
Making a CC Enterprise Efficient

Develop software for non-expert users where:

- Users employ NLD during PSP ! GPS gadget does it.

While

- CC use computer language during PSP! Current CC architecture allows it.



Is such a software possible?

Example Software for Non-Expert Users

HTML technology using Language of Forms (LF):

- HTML tags are LF vocabulary provided with attributes representing form processing events;
- Business documents are structured as dynamic objects ;
- Attributes are associated with computer programs (agents that perform form manipulation);

Convenience of HTML

While business users use BROWSERS

to perform actions on the forms

The agents in the cloud use SERVERS

to perform business computations.

Example Applications

- Airline reservation systems
  (are classic examples);
- Here I would suggest
  A System Selling Iowa Popcorn to Singapore Theaters!

Our Idea:

- Move Document Object Model in the Cloud;
- Generalize LF to the
  Natural Language of Application Domain, NLD
- Use tags as process names instead of
  using them as layout instructions!

Implementation

1. Replace HTML with XML;

Implementation

1. Replace HTML with XML;
2. Replace HTML tags with an XML tag set that characterizes the Application Domain (AD);

Implementation

1. Replace HTML with XML;
2. Replace HTML tags with an XML tag set that characterizes the Application Domain (AD);
3. Associate each XML tag with a computer artifact that implements tag concept meaning;

Implementation

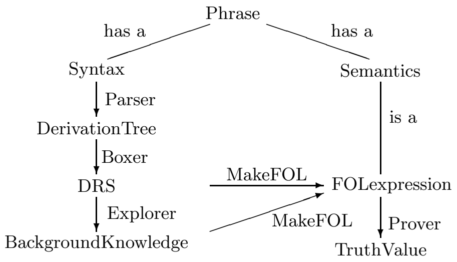
1. Replace HTML with XML;
2. Replace HTML tags with an XML tag set that characterizes the Application Domain (AD);
3. Associate each XML tag with a computer artifact that implements tag concept meaning;
4. Use tag's attributes as computation specification.

Methodology

- Computationally Emancipate Application Domain (CEAD).
- Develop the Natural Language of the Domain (NLD).
- Develop problem models and solution algorithms using NLD.
- Perform NLD algorithm execution in the Cloud using a Domain Dedicated Virtual Machine (DDVM).

CEAD Process: 1

Organize concepts using a Domain Ontology (DO);
Example: Recognizing Textual Entailment (RTE).



CEAD Process: 2

Associate each concept in the DO with a computer artifact that implements it in the cloud;

Note: Computer artifacts are Web services and their URI-s are used in the ontology.

Implement and Use NLD

Example: (no URI-s shown)

```
RTEdecider:
   input: phrase T, H;
   output: phrase Result;
   perform:
   treeT := Parser(T); treeH := Parser(H);
   drsT := Boxer(treeT); drsH := Boxer(treeH);
   bk := Explorer (drsT,drsH);
   ET :=  MakeFOL(drsT); EH := MakeFOL(drsH);
   Result := Prover((bk and ET) implies EH))
```

Implement DDVM

$DDVM = \langle CC, AP, Next \rangle$ where:

1. CC is a concept counter
2. AP is an abstract processor whose instructions are URI-s of Web services in the DO;
3. Next() is a mechanism that selects the next action to be performed by DDVM.

DDVM Execution

For `A` an NLD algorithm and D a DO, `DDVM(A,D)` is simulated by following C-pseudocode:

```
CC := Start Concept of A in D;
while (CC not End)
     {Execute (AP, CC); CC := Next(CC);}
```

Optimize DDVM

- NLD algorithms execution implies ontology search.
- To optimize this process we have created the Software Architecture Description Language (SADL).
- SADL expressions are processes expressed by XML elements.
- Tags of XML elements are ontology terms and tag attributes are computer artifacts implementing these terms.
- Computer user maps NLD algorithms into SADL expressions to be executed by SADL interpretor using the cloud. This can be done by hand or automatically by Map2SADL.

SADL Processes

1. Simple processes represented by empty XML elements:

```
<tag atr_1 = "val_1" ... atr_k = "val_k" />
```

SADL Processes

1. Simple processes represented by empty XML elements:

```
<tag atr_1 = "val_1" ... atr_k = "val_k" />
```

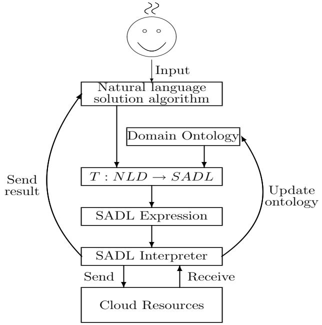
2. Composed processes represented by content XML elements:

```
<tag atr_1 = "val_1" ... atr_m = "val_m">
   process_1
   process_2
   . . .
   process_n
</tag>
```

Example: SADL Expression of RTEdecider

```
<?xml version="1.0" ?>
<sadl> <RTEdecider>
  <Perform manner = "in_sequence">
    <Input input="URI(T) URI(H)" output="URI(result)" />
    <Parser uri="URI(Parser)"
            input="URI(T)" output="URI(treeT)" />
    <Parser uri="URI(Parser)"
            input="URI(H)" output="URI(treeH)" />
    <Boxer  uri="URI(Boxer)"
            input="URI(treeT)" output="URI(drsT)" />
    <Boxer  uri="URI(Boxer)"
            input="URI(treeH)" output="URI(drsH)" />
    <Explorer uri="URI(Explorer)"
        input="URI(drsT),URI(drsH)" output="URI(bk)" />
    <MakeFOL uri="URI(MakeFOL)"
            input="URI(drsT)" output="URI(ET)" />
    <MakeFOL uri="URI(MakeFOL)"
            input="URI(drsH)" output="URI(EH)" />
    <And uri="URI(and)"
       input="URI(bk) URI(ET)" output="URI(antecedent)"/>
    <Implies uri="URI(Implies)"
            input="URI(EH)" output="URI(wff)" />
    <Prover uri="URI(Prover)"
            input="URI(wff)" output="URI(result)"/>
   </Perfrom>
  </RTEdecider>
```

A Software System for CC Use by Non-experts



Computational Emancipation of Application Domain

- CEAD is a natural process that characterizes human knowledge evolution.
- CBPSP forces CEAD into a conscious activity that transcends natural evolution of knowledge.
- Domain dedicated software requires CS to make CEAD process an interdisciplinary methodology that makes problem domains suitable to CBPSP.

CEAD Process

CEAD(D) is a dynamic process that consists of:

1. Identify the characteristic concepts of D which are: universal over D, standalone, and composable.

CEAD Process

CEAD(D) is a dynamic process that consists of:

1. Identify the characteristic concepts of D which are: universal over D, standalone, and composable.
2. Organize the characteristic concepts of D into a DO where terms are associated with computer artifacts.

CEAD Process

CEAD(D) is a dynamic process that consists of:

1. Identify the characteristic concepts of D which are: universal over D, standalone, and composable.
2. Organize the characteristic concepts of D into a DO where terms are associated with computer artifacts.
3. Develop the NLD of D as a notation used by domain experts to create problem models and algorithms.

CEAD Process

CEAD(D) is a dynamic process that consists of:

1. Identify the characteristic concepts of D which are: universal over D, standalone, and composable.
2. Organize the characteristic concepts of D into a DO where terms are associated with computer artifacts.
3. Develop the NLD of D as a notation used by domain experts to create problem models and algorithms.
4. Create a DDVM that executes domain algorithms on the domain ontology.

Example Domain Characteristic Concepts

Consider the domain I of integers in high-school algebra. The operation $+ : I \times I \rightarrow I$ is a characteristic concept because:

1. $+ : I \times I \rightarrow I$ is universal because $\forall n_1, n_2, \in I,\ +(n_1, n_2) \in I$;
2. $+ : I \times I \rightarrow I$ is standalone because $\forall n_1, n_2 \in I,\ +(n_1, n_2)$ depends only on n_1 and n_2;
3. $+ : I \times I \rightarrow I$ is composable because $\forall n_1, n_2, n_3$ $+(+(n_1, n_2), n_3) \in I$.

Note: $+ : INTEGER \times INTEGER \rightarrow INTEGER$ is not a domain characteristic concept in Fortran.

Domain Ontology

DO(D) is a repository of knowledge which plays a double role during the PSP:

- DO(D) supports consistent usage of domain knowledge during problem modeling and algorithm development;
- DO(D) provides the framework to be used by the DDVM(D) for domain algorithms execution.

DO Specification and Representation

W3C standards suggest:

- DO(D) specification:
  a tag set defined by XML schema
- DO(D) Representation:
  Resource Description Framework (RDF).

Knowledge Evolution

Sets of concepts grouped into
domains and sub-domains

Hence, more appropriately:

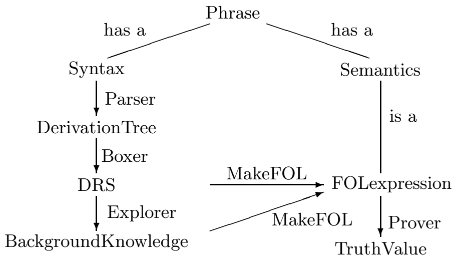
- DO(D) Specification:
  a tag set defined by XML schema;
- DO(D) Representation: a higraph
  (nodes are sets of knowledge, edges are knowledge nesting.)

Labeling a DO Higraph

- Nodes are labeled by terms denoting data concepts
- Edges are labeled by terms denoting actions ($\longrightarrow$) or properties (—).

Example Domain Ontology

Domain: natural language processing.
Problem: Recognizing Textual Entailment (RTE).



Consequences

1. Knowledge is handled unambiguously by PSP using Domain Characteristic Terms (DCT-s).

Consequences

1. Knowledge is handled unambiguously by PSP using Domain Characteristic Terms (DCT-s).
2. DCT-s are universal, standalone, composable.

Consequences

1. Knowledge is handled unambiguously by PSP using Domain Characteristic Terms (DCT-s).
2. DCT-s are universal, standalone, composable.
3. Implementations of DCT-s are associated with them in the DO using URI-s.

Additional Benefit

Bridging Semantics Gap

During CBPSP:

- Domain experts use domain terms while
- Computer experts use URI-s of terms used by domain experts.

NLD Specification

NLD is specified on three levels:

1. Domain Vocabulary, V_D (terms denoting concepts);
2. Simple phrases (NLD expressions of actions or properties);
3. Phrases (expressions of NLD algorithms).

Domain Vocabulary

$V_D = C_D \cup A_D$, where:

- C_D terms that denote domain characteristic concepts;
- A_D terms inherited from IT or other domains.

Note: each $t \in V_D$ is associated with a tuple $(arity, sig, type)$:
$arity \geq 0$, $sig \in V_D^*$, $type \in V_D$.

Simple Phrase

Simple phrases are NLD constructions that represent actions or properties and have the form $t_0 \; c_1 \; \ldots \; c_k$; where:

1. $t_0, c_1, \ldots, c_k \in V_D$,
2. $arity(t_0) = k$, $k \geq 0$,
3. $sig(t_0) = t_1, \ldots, t_k$, and
4. $type(c_i) = t_i$, $1 \leq i \leq k$.

Phrase

A Phrase of NLD is a simple phrase (an action or a property) or an NLD construction of the form

$t_0 \; p_1 \; \ldots \; p_k$ where:

1. $arity(t_0) = k,\ k \geq 0$,
2. $sig(t_0) = t_1, \ldots, t_k$ and
3. $p_1, \ldots, p_k$ are phrases of type $t_1, \ldots, t_k$.

Note: semantically a phrase is an NLD algorithm!

BNF Specification

```
S ::= "AlgName:" [I";"][O";"][Local";"] ActionList
I ::= "input:" DL O ::= "output:" DL Local ::= "local:" DL
DL ::=  D | D "," DL
D ::= "conceptType" VarList ["where" BooleanExpression]
VarList ::=  Var | Var "," VarList
ActionList ::= Action | Action "compose" ActionList
Action ::= ["perform:"] PhraseList
PhraseList ::= Phrase | Phrase ";" PhraseList
Phrase ::= Concept | Concept ArgList | "itOp" Phrase
ArgList ::= "("TermList")"
TermList = Term | TermList "," Term
Term ::= Var | Phrase | Concept
Var ::= "userId" Concept ::= "noun" | "verb"
```

Example NLD Algorithm

```
RTEdecider:
   input: phrase T, H;
   output: phrase Result;
   perform:
     treeT := Parser(T); treeH := Parser(H);
     drsT := Boxer(treeT); drsH := Boxer(treeH);
     bk := Explorer (drsT,drsH);
     ET :=  MakeFOL(drsT); EH := MakeFOL(drsH);
     Result := Prover((bk and ET) implies EH))
```

Domain Dedicated Virtual Machine

Informally: DDVM is a VM that behaves as a pocket calculator provided with the picture of a DO on which:

- the user can select and
- the user can click

buttons labeled by the actions she wants to perform.

DDVM, Formally

$DDVM = \langle CC, AP, Next \rangle$ where:

1. CC is a concept counter
2. AP is an abstract processor, and
3. Next() is a mechanism that selects the next action to be performed by DDVM.

NLD Algorithm Execution

For a CEAD-ed domain D and NLD algorithm $\mathcal{A}$, $DDVM(\mathcal{A}, D)$ performs as follows:

1. CC selects the concept of $\mathcal{A}$ in D to be executed next;

NLD Algorithm Execution

For a CEAD-ed domain D and NLD algorithm $\mathcal{A}$, $DDVM(\mathcal{A}, D)$ performs as follows:

1. CC selects the concept of $\mathcal{A}$ in D to be executed next;
2. AP execute computations associated with $CC(\mathcal{A}, D)$, if any;

NLD Algorithm Execution

For a CEAD-ed domain D and NLD algorithm $\mathcal{A}$, $DDVM(\mathcal{A}, D)$ performs as follows:

1. CC selects the concept of $\mathcal{A}$ in D to be executed next;
2. AP execute computations associated with $CC(\mathcal{A}, D)$, if any;
3. Next() determine next concept of $\mathcal{A}$ in D to be executed.

Simulating DDVM

The behavior of a DDVM can be expressed by the following C-like program:

```
CC := Start Concept of A in D;
while (CC not End)
     {Execute (AP, CC); CC := Next(CC);}
```

Contrasting DDVM with a Computer

DDVM mimics the behavior of a computer which:

- operates on the DO instead of memory,
- operations are processes defined by the URI-s associated with the concepts in the DO.

DDVM Summary

1. The input to DDVM is an NLD algorithm, not a program;

DDVM Summary

1. The input to DDVM is an NLD algorithm, not a program;
2. CC point to a concept in the DO not to a memory location;

DDVM Summary

1. The input to DDVM is an NLD algorithm, not a program;
2. CC point to a concept in the DO not to a memory location;
3. Concept pointed to be CC is evaluated by AP which may create and execute a process in the cloud;

DDVM Summary

1. The input to DDVM is an NLD algorithm, not a program;
2. CC point to a concept in the DO not to a memory location;
3. Concept pointed to be CC is evaluated by AP which may create and execute a process in the cloud;
4. Next(CC) represent the action performed by computer user.

PSP using DDVM

A computer user solves problems using DDVM following Polya four steps methodology:

1. Formulate the problem as an NLD expression;

PSP using DDVM

A computer user solves problems using DDVM following Polya four steps methodology:

1. Formulate the problem as an NLD expression;
2. Develop an NLD algorithm that solves the problem;

PSP using DDVM

A computer user solves problems using DDVM following Polya four steps methodology:

1. Formulate the problem as an NLD expression;
2. Develop an NLD algorithm that solves the problem;
3. Input (type) the algorithm to DDVM;

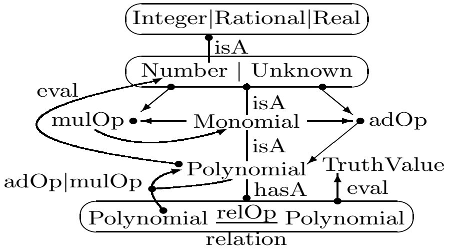
PSP using DDVM

A computer user solves problems using DDVM following Polya four steps methodology:

1. Formulate the problem as an NLD expression;
2. Develop an NLD algorithm that solves the problem;
3. Input (type) the algorithm to DDVM;
4. Execute the algorithm, i.e. set CC to the first concept.

Example DDVM Use

Consider the domain of high-school algebra whose ontology is in the following higraph (no URI-s shown):



Example problem solving process

Problem: Solve second degree equations:

1. Problem model: the relation $ax^2 + bx + c = 0$, $a \neq 0$;

Example problem solving process

Problem: Solve second degree equations:

1. Problem model: the relation $ax^2 + bx + c = 0$, $a \neq 0$;
2. High-school solution algorithm:

```
Solver:
input real a, b, c where a is not zero;
local real t := b^2 - 4*a*c;
if t is positive or 0
output x1 := (-b + sqrt(t))/2*a;
x2 := (-b - sqrt(t))/2*a;
else
output "there are no real solutionss".
```

Example problem solving process

Problem: Solve second degree equations:

1. Problem model: the relation $ax^2 + bx + c = 0$, $a \neq 0$;
2. High-school solution algorithm:

```
Solver:
input real a, b, c where a is not zero;
local real t := b^2 - 4*a*c;
if t is positive or 0
output x1 := (-b + sqrt(t))/2*a;
x2 := (-b - sqrt(t))/2*a;
else
output "there are no real solutionss".
```

3. Set CC to Solver;

Example problem solving process

Problem: Solve second degree equations:

1. Problem model: the relation $ax^2 + bx + c = 0$, $a \neq 0$;
2. High-school solution algorithm:

```
Solver:
input real a, b, c where a is not zero;
local real t := b^2 - 4*a*c;
if t is positive or 0
output x1 := (-b + sqrt(t))/2*a;
x2 := (-b - sqrt(t))/2*a;
else
output "there are no real solutionss".
```

3. Set CC to Solver;
4. Type data when requested.

Optimizing DDVM

- NLD algorithms execution implies ontology search.
- To optimize this process we have created the Software Architecture Description Language (SADL).
- SADL expressions are processes expressed by XML elements.
- Tags of XML elements are ontology terms and tag attributes are computer artifacts implementing these terms.
- Computer user maps NLD algorithms into SADL expressions executed by SADL interpretor using the cloud.

SADL Processes

1. Simple processes represented by empty XML elements:

   ```
   <tag atr_1 = "val_1" ... atr_k = "val_k" />
   ```

SADL Processes

1. Simple processes represented by empty XML elements:

```
<tag atr_1 = "val_1" ... atr_k = "val_k" />
```

2. Composed processes represented by content XML elements:

```
<tag atr_1 = "val_1" ... atr_m = "val_m">
   process_1
   process_2
   . . .
   process_n
</tag>
```

Example: SADL Expression of RTEdecider

```
<?xml version="1.0" ?>
<sadl> <RTEdecider>
  <Perform manner = "in_sequence">
    <Input input="URI(T) URI(H)" output="URI(result)" />
    <Parser uri="URI(Parser)"
            input="URI(T)" output="URI(treeT)" />
    <Parser uri="URI(Parser)"
            input="URI(H)" output="URI(treeH)" />
    <Boxer  uri="URI(Boxer)"
            input="URI(treeT)" output="URI(drsT)" />
    <Boxer  uri="URI(Boxer)"
            input="URI(treeH)" output="URI(drsH)" />
    <Explorer uri="URI(Explorer)"
        input="URI(drsT),URI(drsH)" output="URI(bk)" />
    <MakeFOL uri="URI(MakeFOL)"
            input="URI(drsT)" output="URI(ET)" />
    <MakeFOL uri="URI(MakeFOL)"
            input="URI(drsH)" output="URI(EH)" />
    <And uri="URI(and)"
       input="URI(bk) URI(ET)" output="URI(antecedent)"/>
    <Implies uri="URI(Implies)"
            input="URI(EH)" output="URI(wff)" />
    <Prover uri="URI(Prover)"
            input="URI(wff)" output="URI(result)"/>
   </Perfrom>
  </RTEdecider>
```

Mapping NLD algorithms into SADL

- Mapping of NLD algorithms into SADL can be done by the domain expert by hand (This is feasible for toy problems).
- For more sophisticated problems it is beneficial to automate this process.

Note: `Maps2SADL NLDalgorithm` does it.

Maps2SADL

The development of `Maps2SADL` is facilitated by:

1. The lexicons of NLD and of SADL are finite and one-to-one connected.
2. NLD language has a simple syntax that avoids usual ambiguities present in NL.

Note: Computer user is not aware of SADL or Maps2SADL!