1 Introduction
Still from begin the new methods which were proposed as remedy for the programming engineering suffered of the communication problem.

For approach the communication problem we shall kept a relation between AD and IT on lifecycle software. In [6] they considered the Model Driven Architecture MDA, and Extreme Programming as approach for the improvement communication in the process of the engineering software. The methodology suggested [6] is based on the domain computation emancipation of the problem.

In application domain ontology these abstractions developed by IT experts are domain characteristic terms attached with the processes that implement them. Software developed process is based on well-define concepts that characterize application domain of the software thus developed. Language processing as domain application, illustrates this new methodology of problem solving.

The current method of computer usage requires that the human logic of problem solving be encoded into a program in computer memory that is executed as described by the following loop referred to further by program execution loop, PEL: while((PC).Opcode ≠ Halt){Execute(PC); PC: = Next( PC)}. Here PC is the program counter register holding the memory address of the current instruction of the program, Execute(PC) performs the operation encoded in the current instruction, and Next(PC) determines the address of the next instruction of the program. Computer systems set this loop as the foundation for their usage in problem solving A solution of the problem for be executed by computer, doing abstraction of the level of language represented, must described application as loop present before.

2 Syntax and semantics of CTL model
CTL is a branching time temporal logic meaning that its formulae are interpreted over all paths beginning in a given state of the Kripke structure. Definition of a Kripke model [2] let \( AP \) is a set of atomic propositions. A Kripke model \( M \) over \( AP \) is a triple \( M = \langle S, R, F:S \rightarrow 2^{AP} \rangle \) where \( S \) is a finite set of states, \( R \subseteq S \times S \) is a transition relation, \( F:S \rightarrow 2^{AP} \) is a function that assigns each state with a set of atomic proposition.

A model is defined [1] as a directed graph \( M = \langle S, E, P:AP \rightarrow 2^E \rangle \) where \( S \) is a finite sets of states also called nodes, \( E \) is a finite sets of directed edges, and \( P \) represents proposition labeling function which labels each nodes with logical proposition. For each \( s \in S, \) use the notation \( \text{succ}(s) = \{s' \in S | (s,s') \in E \} \). Each state in \( E \) must have at least one successor, that is \( \forall s \in S, \text{succ}(s) \neq \emptyset \). A path in \( M \) is an infinite sequence of states \((s_0, s_1, s_2, ...)\) such that \( \forall i, i \geq 0, \) we have \((s_i, s_{i+1}) \in E \). The labeling function \( P \) maps an atomic proposition in \( AP \) to
the set of states in $S$ on which sentences is true.

**Syntax definition of a CTL model checker** [3] A CTL has the following syntax given in Backus near form:

$$f ::= \forall \wp \exists [\neg f_1] | f_1 A[f_2] | f_1 U[f_2] | f_1 \subset f_2 | AX f_1 | EX f_1 | AG f_1 | EG f_1 | AF f_1 | EF f_1 | A[f_1 U f_2] | E[f_1 U f_2] | E[f_1 \subset f_2] \text{ where } \wp \in AP.$$ 

A CTL formula is evaluated on a Kripke model $M$. A path in $M$ from a state $s$ is an infinite sequence of states $\pi = [s_0, s_1, \ldots, s_i, \ldots]$. If we express a path as $\pi = [s_0, s_1, \ldots, s_i, \ldots]$ and $i < j$, we say that $s_i$ is a state earlier than $s_j$ in $\pi$ as $s_j < s_i$. For simplicity, we may use $\text{succ}(s)$ to denote state $s_0$ if there is a relation $(s, s_0)$ in $R$.

Syntactically, we divide CTL formulae into three categories:

1) those whose outermost operator, if any, is not a temporal operator,
2) those whose outer most operator is a temporal operator ($X$ (next), $U$ (until), $F$ (eventually) or $G$ (always)) prefixed with the existential path quantifier E, and
3) those whose outer most operator is a temporal operator prefixed with the universal path quantifier A.

The formulae in category (i) comprise atomic propositions on the states of the Kripke structure, as well as logical combinations of other CTL formulae from categories (i), (ii) and (iii). Specifically, if $f_1$ and $f_2$ are CTL formulae then so are: $\neg f_1, f_1 A f_2, f_1 U f_2, f_1 \subset f_2$.

The formulae in category (ii) express properties which are true on at least one path of the Kripke structure $M$ starting from $s$. For example, $EX f_1$, where $f_1$ is a CTL formula, states that for at least one path starting from the state $s$, $f_1$ holds in the next state. Similarly, $EF f_1$ states that for at least one path starting from $s$, $f_1$ holds until some state where $f_1$ holds. Also, $AX f_1$ states that for at least one path starting from $s$, $f_1$ eventually holds, and $EG f_1$ states that for at least one path starting from $s_0$, $f_1$ always holds.

The formulae in category (iii) express properties which are true on all paths of $M$ starting from $s$. For example, $AX f_1$, where $f_1$ is a CTL formula, states that for all paths starting from the state $s_0$, $f_1$ holds in the next state. Similarly, $A[f_1 U f_2]$ states that for all paths starting from $s$, $f_1$ holds until some state where $f_2$ holds. Also, $AF f_1$ states that for all paths starting from $s$, $f_1$ eventually holds, and $AG f_1$ states that for all paths starting from $s$, $f_1$ always holds.

**Semantics definition of a CTL model checker** [3]. Let $M = (S, R, F : S \rightarrow 2^AP)$ be a Kripke model for CTL. Given any $s$ in $S$, we define if a CTL formula $f$ holds in state $s$. We denote this by $(M, s) \models f$. The satisfaction relation $\models$ is defined by structural induction on fourteen

### 3 Problem formulation

Software systems are developed inner problem solving instruments that are independent of specific problem domain. With these tools, a problem solution is program conversion by a conventional programming language. The conventional consequence of following route is the absence of problem abstract domains in the process of problem solving. More, these approximations realize substantial increase in soft complexity.

Ontology engineering is the needed link because it can be used to bring the computational power of the computer to the computer end-user. Ontology provides the abstractions AD experts need to express naturally their problems and solution algorithms while computer artifacts associated as semantics with the concepts of the ontology allow IT experts to develop software that maps AD systems into computer processes that implement them.

The frame for description and development software based on applications is supported by: (1) methodology development for application-domain structuring using engineering tools; this will enable the design of domain-driver software architecture description language (SADL) that capture elements of the problem-solving process as expressions in the natural language of the problem domain; (2) implement SADL interpreters that integrate computing abstractions (used in SADL expressions) and generate computing processes that execute the algorithms expresses in SADL; (3) use this methodology to conduct problem solving experiments in such diverse problem domains as Internet Agents, Geography and Hydrology.

These objectives are accomplished using problem solving methodology where problem domain experts use natural language to express problem solving system while computer experts develop tools that map domain expert solution in the frame of computer processes. No programming as usual is involved.

Next follows a problem domain application that will be solving using a methodology that executes two tasks: (1) computational emancipation of problem domain and (2) software system development that describe the algorithms of oriented-domain in a computer process.

### 3.1 Computational emancipation of problem domain

By computational emancipation, two things should be understood:

1) a particular AD is produced by a computational structure of oriented-domain, that means the computational emancipation process of a domain
application can be obtained by AD structuring using ontology; (2) computational structure of the model is given by IT semantics, that means it is made by delivery of ontology concepts with IT semantics that has a computational mean [6].

The elements of the vocabulary of the language ontology in Fig. 1 are:

- **alphabet**, which is a finite set of distinguishable symbols
- **lexical entities**, which are strings of symbols over the **alphabet**, usually split in classes that may not be disjoint.
- **discourse** is the collection of well-formed language-constructs classified as syntax categories by the specification rules
- **objects** and **actions** are elements of the universe of discourse (UoD) denoted by the lexical entities of the language.
- **properties** represent knowledge about the UoD and are expressed by language phrases.
- **syntax** and **Semantics** are related by two operators **Value : Syntax → Semantics** and **Discourse : Semantics → Syntax** defined by equations of the form:

\[
\text{Value(Syntax:element)} = \{\text{UoD:elements denoted by Syntax:element}\}
\]

\[
\text{Discourse(UoD:element)} = \{\text{Syntax:elements denoting the UoD:element}\}
\]

In fig.1 this tree represents the vocabulary of the ontology, interior nodes represent concepts constructed in terms of other concepts, and the arrows and the dotted lines represent relationships between the concepts in the ontology.

In fig.1 this tree represents the vocabulary of the ontology, interior nodes represent concepts constructed in terms of other concepts, and the arrows and the dotted lines represent relationships between the concepts in the ontology.

![Fig. 1 Language Ontology](image)

The computational meaning of the concepts used by the domain ontology is universal algorithms that solve classes of problems, are characteristic to the domain, and are mathematically proven correct. The problem solving process transforms these assumptions into the following characterization of the computational meaning of the AD concepts:

- Computational meaning associated with an ontology node is stand-alone computation processes further referred to as components.
- The behavior of a computational process is completely defined by its input/output performance.
- The interaction between computational processes is achieved by an appropriate combination of one or more of the mechanisms: calling patterns, sharing appropriate data or procedures, messaging systems.
- Composition of computational processes is performed by filters that map the output generated by a component into the input expected by another component.

The goal of computational emancipation of an AD is to allow AD experts to use computer technology as a problem solving tool dedicated to their AD without requiring the AD expert to develop IT representations. Assuming that the computational emancipation of the AD has been performed, the AD concepts are already associated with computations performed by the problem solving tool, the computer.

We illustrate the use of computational emancipation in language processing with the following problem: \( M, s \vDash f? \) where \( M \) is a model of analyzed system, and \( s \) is a state of model and \( f \) is a CTL model formula that must be satisfied by the system.

### 3.2 Ontology of domain

We use the ontology terms for show formalization of domain knowledge. That is, ontology is a collection of AD terms organized on based of properties them and the relations among these. Structured the application domain, the domain expert is responsible same of the create process of domain ontology collecting and organizing the concepts from domain.

Each application domain is characterized of a collection of terms (terminology), which have same semantic interpretation for all the domain experts. The association symbolic names to these terms inside the table deliver an easy way of interpretation (translation) of these as long as they are keeping the semantics.

For the aim utilization ADS to the solution of our problem, we are interested by the little subsequence of terms from domain which represent the computation process characterized by of the behavior of their input/output date as follows:

We simplify the our application using a reduced version of connecters CTL \( \Gamma = \{\bot, \neg, \land, \lor, AF, EU, EX\} \)

\[
\]
where:
- $\bot, \land, \lor$ are used for the propositional part.
- $A,F,EU,EX$ are used for the temporal part.

The function of this algorithm is to:

a) decompose formula $f$ in pieces (sub-formulas) and apply a structural induction to label the graph with sub-formulas of $f$ (the intuition is that a formula that labels a state is true in that state).

b) for each such sub-formula, parse the graph to infer the truth in a state according to the meaning of the connectives and the truth values of its sub-formulas.

In 2, one may need to know the values of sub-formulas in possibly many different states; this is the case for temporal operators, but not for the propositional ones.

The ontology development for the application domain represent the identification of these terms and their organization which to permit the implication in reasoning’s process. The ontology for application domain (in our case the language programming) can be very large. However, ontology terms can be classified on a hierarchy of sub-domains and the problem solving can be center just on the sub-domain ontology appropriate.

For the purpose of using ADS to solve our problem, we are concerned with a small subset of terms in the domain whose meanings are computational processes characterized by their input/output behavior as follows:

The function SAT takes a CTL formula as input and returns the set of states satisfying the formula. It calls the function SAT_EX($f$), SAT_EU($f$) and SAT_AF($f$), respectively, if EX, EU or AF is the root of the input’s parse tree.

\[
\text{function SAT(f) /\!* determines the set of states satisfying f */}
\]
\[\text{Input: a CTL model } M = (S, \rightarrow, F) \text{ and a CTL formula } f \text{ (in } \Gamma\text{-format)}
\]
\[\text{Output: the set of states of } M \text{ which satisfy } f, (M, s_0) \models f, s_0 \in S\]

function SAT(p) /\!* determines the set of states satisfying p */
\[\text{Input: } p\]
\[\text{Output: the set of states of } M \text{ which satisfy } p, (M, s_0) \models p\]

\[
\text{function SAT_not(f) /\!* determines the set of states satisfying } \neg f \ */
\]
\[\text{Input: } \neg f\]
\[\text{Output: the set of states of } M \text{ which satisfy } \neg f, (M, s_0) \models \neg f\]

The function SAT_EX computes the states satisfying $f$ by calling SAT. Then, it looks backwards along $\rightarrow$ to find the states satisfying EX $f$.

\[
\text{function SAT_EX(f) /\!* determines the set of states satisfying } EX f \ */
\]
\[\text{Input: formula } f\]
\[\text{Output: the set of states of } M \text{ which satisfy } EX f\]

The function SAT_AF computes the states satisfying $f$ by calling SAT. Then, it accumulates states satisfying AF $f$ in the manner described in the labeling algorithm.

\[
\text{function SAT_AF(f) /\!* determines the set of states satisfying } AF f \ */
\]
\[\text{Input: formula } f\]
\[\text{Output: the set of states of } M \text{ which satisfy } AF f\]

\[
\text{function SAT_EU(f1,f2) /\!* determines the set of states satisfying } EU f1 \cup f2 \ */
\]
\[\text{Input: formula } f1, f2\]
\[\text{Output: the set of states of } M \text{ which satisfy } EU f1 \cup f2\]

\[
\text{function SAT_EG(f) /\!* determines the set of states satisfying } EG f \ */
\]
\[\text{Input: formula } f\]
\[\text{Output: the set of states of } M \text{ which satisfy } EG f\]

\[
\text{function SAT_all(Y) = } S - \text{pre_exist}(S-Y)\]
\[
\text{function SAT_exist(Y) = Y}\]

Both functions compute a pre-image of set of states. pre_exist($Y$), instrumental in SAT_EX and SAT_EU, takes of subset $Y$ of states and returns the set of states which make a transition into $Y$. The function pre_all($Y$) is used in SAT_AF, takes a set $Y$ and returns a set of states which make transitions only into $Y$.

3.3 Computational emancipation of AD
The process of supplying the application domain ontology [3] with the implementation of IT artifact are named Computational Emancipation of the Application Domain, in short name CEAD.

Applying this approach to the language sub-domain, CEAD is done by associating each term in the language ontology with a Uniform Resource Identifier (URI) pointing to the IT component implementing it. We
organize these terms in a tree giving us the fully emancipated language processing ontology shown in Figure 2.

We get away from our example we shall build follow-up the ontology of emancipated language processing ontology.

### 3.4 Computational emancipation of AD

For any problem, the solution is formulated in the domain of discourse from which the problem originates. As language processing experts we provide the following formal solution to our problem, formulated in terms of the computationally emancipated language processing domain.

**Input:** formula \( f \)

**Output:** \( M \models f \)

```plaintext
switch (f)
    case cond1 : RESULT := SAT;
    case cond2 : RESULT := SAT_not;
    case cond3 : RESULT := SAT_and;
    case cond4 : RESULT := SAT_EX;
    case cond5 : RESULT := SAT_AF;
    case cond6 : RESULT := SAT_EU;
    default: return Error;
end switch
```

where \( cond_1 \) means \( p \); \( cond_2 \) means \( \neg f_1 \); \( cond_3 \) means \( f_1 \wedge f_2 \); \( cond_4 \) means \( EX \ f_1 \); \( cond_5 \) means \( AF \ f_1 \); \( cond_6 \) means \( EU \ f_1 \). Functions SAT, SAT_not,...,SAT_EU are function with have a result a value true if the formula satisfy M model.

Follow up shall execute the SADL solution. ADS methodology for the execution of domain solutions is a two step process: translation and interpretation. Before a solution can be executed it must be first translated into a process interpretable language called the Software Architecture Description Language (SADL) [5]. This translation replaces each domain concept with its associated IT artifact and each operational instruction (if, while, etc.) with an appropriate SADL operator. After translation to SADL, the solution process is carried out by the SADL interpreter.

### 3.5 SADL

SADL is a process-description language used in ADS to represent domain solutions in an intermediate form. The structure of the SADL language is built on the principles:

a) The lexical elements of the language are AD terms and SADL operators. The semantics of AD terms are the software artifacts associated with them in the AD ontology. The semantics of SADL operators are specified by the SADL interpreter.

b) The SADL primitive processes specified by the signature of the component used in the AD ontology or by SADL operators that compose such processes in SADL.

c) The SADL composed process which consists of sequential and parallel process compositions of one or more SADL processes that implement an AD solution algorithm.

The syntax of SADL is built on the extensible markup language (XML) syntax. The two types of SADL processes are represented by the two types of XML elements:

- **SADL primitive processes are represented by empty XML elements of the form** \(<op\ atr_1 = val_1 \ldots atr_n = val_n />\) **where op is an SADL operator which performs a process and atr_1, \ldots, atr_n defined the properties of that process such as the URI of the code that implements it, input, output, etc.**

- **SADL composed processes are represented by content XML elements of the form** \(<op\ atr_1 = \ldots = val_1 ; \ldots ; atr_n = val_n />\).
\[ \text{val}_{1} \ldots \text{atr}_{n} = \text{val}_{1} \ldots \text{p}_{n} \text{ </op> where op is a SADL operator that composes the processes p}_{1} \ldots \text{p}_{n} \text{ using atr}_{1} \ldots \text{atr}_{n} \text{ determine the behavior of the composition.} \]

The process performed by the SADL for each element is determined by the SADL operator. For our solution the interpreter recognizes the following set of SADL process operators.

Interpretation of the SADL language is handled by the SADL interpreter acting as a virtual processor. That is, the SADL interpreter mimics the behavior of a physical processor using a virtual program counter that points to the emancipated nodes of the domain ontology. The processes performed by the SADL interpreter are based on SADL semantics defined above and each process defined by the software artifact used in the ontology is performed on the computer platform in the network where it exists. The challenge for the interpreter is in performing process composition. In sequential composition this means executing software artifacts, waiting for execution to terminate and performing the next composition. This challenge is compounded by the fact that solutions contain sub-compositions as is the case of the \texttt{switch} with \texttt{case} process. The assumption here is that software artifacts associated with ontology nodes are correct and terminating.

The solution given in Fig. 3 is the SADL form of our domain solution given in Section 3.4. The process performing this solution is the composition of processes between the \texttt{<system>} and \texttt{</system>} tags.

```sadl
<system name="evaluator">
  <execute component="&it:semipc_formula" input="input" output="formula"/>
  <switch atr="formula">
    <execute component="&it:semipc_testSAT" input="input" output="SAT"/>
    <node name="RESULT" term="SAT" />
    <execute component="&it:semipc_SAT" input="input" output="SAT"/>
    <execute component="&it:gcc" input="pt, &it:c_SATParser" output="output"/>
    </case>
    <execute component="&it:semipc_testSAT_not" input="input" output="SAT_not"/>
    <node name="RESULT" term="SAT_not" />
    <execute component="&it:semipc_SAT_not" input="input" output="SAT_not"/>
    <execute component="&it:gcc" input="pt, &it:c_SAT_notParser" output="output"/>
    </case>
  </switch>
</system>
```

<execute component="&it:semipc_testSAT_and" input="input" output="SAT_and"/>
<case test="cond3">
  <node name="RESULT" term="SAT_and" />
  <execute component="&it:semipc_SAT_and" input="input" output="SAT_and"/>
  <execute component="&it:gcc" input="pt, &it:c_SAT_andParser" output="output"/>
</case>
.......
<execute component="&it:semipc_testSAT_not" input="input" output="SAT_not"/>
<case test="default">
  <message text="Error generating parser."/>
</case>
</switch></system>
</sadl>

Fig. 3: SADL solution

4 Conclusions

We show that the application domain expert can step away from computer implementations focusing on the application domain. For that, the application domain must be computationally emancipated by providing it with domain ontology. This ontology can then be used as basis for the development of a domain-driven methodology for problem solving with a computer where processes characterizing the application domain are composed rather than composing code representing software itself.

References: