Programming Without Code: An Approach And Environment For Conditions-On-Data Programming

Philippe Larvet

***Abstract*—**This paper presents the concept of an object-based programming language where tests (if... then... else) and control structures (while, repeat, for...) disappear and are replaced by conditions on data.

According to the object paradigm, by using this concept, data are still embedded inside objects, as variable-value couples, but object methods are expressed in the form of logical propositions ("conditions on data" or COD).

For instance:

variable1 = value1 AND variable2 > value2 => variable3 = value3

Implementing this approach, a central inference engine turns and examines objects one after another, collecting all CODs of each object. CODs are considered as rules in a rule-based system: the left part of each proposition (left side of the "=>" sign) is the premise and the right part is the conclusion. So, premises are evaluated and conclusions are fired. Conclusions modify the variable-value couples of the object and the engine goes to examine the next object.

The paper develops the principles of writing CODs instead of complex algorithms. Through samples, the paper also presents several hints for implementing a simple mechanism able to process this "COD language".

The proposed approach can be used within the context of simulation, process control, industrial systems validation, etc. By writing simple and rigorous conditions on data, instead of using classical and long-to-learn languages, engineers and specialists can easily simulate and validate the functioning of complex systems.

***Keywords*—**conditions on data, logical proposition, programming without code, object programming, simulation, system validation.

I. The Usefulness of a Simple Programming Language

Within the context of complex system simulation, process control verification/improvement or specification validation, adapted tools are needed to help engineers to model the process of the future system or the functioning of the present system to be improved. By using these tools, engineers have to express and combine the functioning of subsystems, procedures, mechanisms, etc. by describing and modeling diverse industrial objects or concepts, then putting and connecting them together in order to represent the final functioning of the system.

In order to achieve this purpose, the use of modeling tools [1] is possible, but they have to be manipulated by experts. A semantic description of all subsystems in terms of semantic components [2] is another way, but this approach is difficult to handle for engineers and specialists who are not necessarily IT professionals. A description of the system is also possible through the implementation of a collection of interacting objects, within the scope of OO programming [3], but this powerful way is once more reserved to computer engineers.

This paper presents another approach, combining the power of object paradigm with the simplicity of manipulating only **data** and **conditions** on these data, without the difficulty of writing complex code or algorithms. So, this approach – that could be considered a programming-without-code way – can be used by non-IT professionals.

Philippe Larvet was research engineer at Alcatel-Lucent Bell Labs, Villarceaux (France). Author of numerous publications and patents, he works now as independent consultant in Bergerac (France). e-mail: phlarvet@ gmail.com. LinkedIn https://www.linkedin.com/in/philippe-larvet-9ba391/

II. What is a Program?

Within the scope of modern development [4], a program is a set of interacting **objects** [5]. Each object embeds

- **data**, under the form of variable-value couples, e.g.
`variable1 = value1`

- and **methods**, which can be seen as services rendered by the objects.

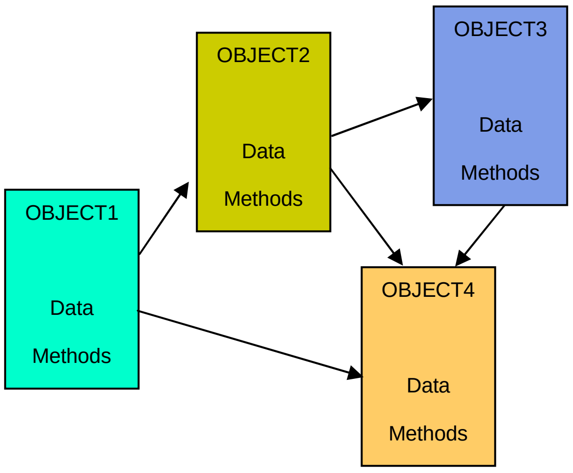

Fig. 1 A Program is a Set of interacting Objects

Analyzing the system, modeling the concepts and splitting them into interacting objects are tasks accessible to non-IT professionals [6]. But the real difficulty, the complexity of the dynamic functioning of the system, is hidden inside objects, within their methods. There, have to be written algorithms and software mechanisms, expressed in “code” (Java, C++, C#, PHP, etc.). And there, computer specialists are needed.

So, a means of achieving a complete and executable model of the system could be very interesting, mainly if only system specialists could express the model by themselves, without the help of IT-professionals. It is well known that computer engineers generally turn the specifications into their own "sauce", and most of the time the system finally implemented does not meet the initial requirements. Building a system within the scope of a precise activity field must be a matter for specialists in this field, not for IT specialists.

Consequently, a way allowing non computer engineers to describe the complexity of object methods and algorithms could be very relevant.

Let us see with a little more detail the contents of these methods or algorithms.

Written in a given programming language, the code of an algorithm is made of process statements (instructions). Statements can be partitioned into four well-defined categories [7]:

- Accessors
- Calculations
- Method calls
- Tests

These categories are characterized in terms of the purpose of the instruction and the use it makes of object data.

Accessors are statements whose only purpose is to access data in a single object. Accessors can read or write (= update) data of a single object.

Examples:

```
1) cont = TANK.content
```

means the data (= variable) `content` of the object `TANK` is read and set to the variable `cont`.

```
2) TIMER.timeout = 50
```

means the value 50 is allocated to the variable `timeout` of the object `TIMER` ; in other terms, the value 50 is written to the internal data `timeout` of the object `TIMER`.

Calculations (or transformations) are statements whose purpose is one of computation or transformation of data.

Examples:

```
1) i = i + 1 (or i++)
```

This very classical "dynamic equation" means the value of `i` is incremented and the new result is written to `i`.

```
2) delta = (b*b) – (4*a*c)
```

Method calls are statements that call methods onto other objects.

Example:

```
tax = COMMON.compute_tax(param)
```

means the method `compute_tax` of the object `COMMON` is called, and the result of the computation is set to the variable `tax`.

Tests are statements that test conditions on some data and control the process flow according to the test results. There are two kinds of Tests: simple (if… then… else) and repetitive (while, repeat… until, for, do… loop).

Examples:

```
1) if (condition) then action_1
   else action_2

2) for (j=0; j<limit; j++)
    action_3
   end for

3) while (condition)
     action_4
   end while
```

where the different `action_i` are sets of other statements.

According to these definitions, a program can be seen as a set of objects, each object embedding a set of complex statements.

III. How to Reduce the Programming Complexity?

We assume that writing Method calls and Tests represents the true complexity of programming activity. With these two categories of statements, we are at the heart of the difficulty of writing programs.

In order to reduce this difficulty, it is necessary to think about the main characteristics and purposes of programs, at the highest level:

- Programs transform data into other data (also called "results").
- Programs are dynamic (in the common language, one says "the program turns").
- Programs have to keep data and intermediate results during all the time of their execution, and have to render final results at the end.

If programs can be considered as sets of objects, these assertions become (we will call them our "main assertions"):

- MA1: Objects keep data (and results).
- MA2: Objects transform data into results.
- MA3: Objects are dynamic.

Synthetically speaking, the main purpose of a program is to process and transform data. Looking at the detail of statement syntax, we observe that statements, except tests, are always written under the form of equations that express relationships between data. So, a good question could be: why not to express Tests as relationships - or conditions on data?

In mathematics, we write

```
A = B and B = C => A = C
```

that is semantically equivalent to

```
if A = B and B = C then A = C
```

So, the conditions on data A, B and C have the semantics of a Test in terms of process statement.

Consequently, the following Test

```
if (condition) then action_1
else action_2
end if
```

could be expressed as

```
condition => action_1
non(condition) => action_2
```

And the following one:

```
for (j=0; j<limit; j++ )
   action_3
end for
```

could be expressed as

```
j = 0 => action_3 AND j++
j < limit => action_3 AND j++
```

which is equivalent to

```
j = 0 OR j < limit => action_3 AND j++
```

In the same way, this Test

```
while (a = b )
   action_4
end while
```

is equivalent to the following COD:

```
a = b => action_4
```

In short, we assume that a program can be written in terms of

- data
- relationships on data (dynamic equations)
- conditions on data.

Then, a first level of reduction of the programming complexity is reached by expressing all the code in terms of data and conditions on data (COD): *code* becomes *data and COD*.

So, our main assertion MA1 becomes:

- MA1: Objects keep data and CODs.

CODs formulate Tests, but they have a disadvantage: they do not express repetitions and loops, i.e. the dynamics of the processing.

IV. How to Simplify the Problem of Dynamics?

A simple possibility to solve this problem is to put the dynamics outside the objects. For this, we can rely on the concept of expert systems [8] or rule-based systems [9].

This concept is interesting here for two reasons:

- CODs can be seen as production rules;
- the inference engine mechanism is the solution to place the dynamics outside the objects.

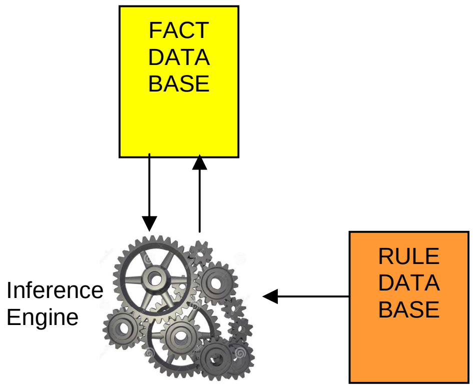

Fig. 2 Schema of an Expert System

In an expert system, the inference engine examines one after another the rules in the Rule Database.

Rules have the general form:

```
if (premise1) AND (premise2)
then (conclusion1) AND (conclusion2)
```

If the premises of the rule (i.e. the conditions) are present in the Fact Database, the rule is fired and the conclusions of the rule are written in the Fact Database. The engine turns automatically until no new conclusion is written.

TABLE I
Algorithm Of An Expert System Inference Engine

```
repeat
  newConclusion = False

  for each Rule in RuleDataBase
  if fired(Rule) = False
    nbPremisesFoundInFactDB = 0
      for each Premise of the Rule
       for each Fact in FactDataBase
        if Premise = Fact
        then nbPremisesFoundInFactDB++
        end if
       end for
      end for

    if nbPremisesFoundInFactDB = nbPremisesOfTheRule
    then for each Conclusion of the Rule
        add Conclusion to FactDB
       end for
       fired(Rule) = True
       newConclusion = True
    endif

  end if
  end for
until newConclusion = False
```

A COD has the semantics of a production rule:

```
(condition1) AND (condition2)
=> (action1) AND (action2)
```

The left part of each COD (left side of the "=>" symbol) is the premise(s) and the right part is the conclusion(s).

The main differences between the COD approach and the expert system paradigm are presented in the following table.

TABLE II
Differences Between COD Approach and Expert Systems

COD approach	**Expert systems**
CODs express conditions on equalities (i.e. equations)	Rules express predicates or “logical propositions”
CODs are distributed into all objects because a COD belongs to a given object	Rules are stored in a unique Rule Database
Data are formal variable-value couples	Facts can be informal
Data are distributed into all objects because a given variable belongs to a given object	Facts are stored in a unique Fact Database
The purpose of COD processing is to accomplish a given process and to achieve a result, i.e. a final state of the system, meeting initial requirements	The purpose is to get some conclusions, but no special result is necessarily targeted

In the COD approach, we keep the principle of the Inference engine. Due to this mechanism, our main assertions become:

- MA1: Objects keep data and CODs.
- MA2: The Inference engine transforms data according to CODs
- MA3: The Inference engine manages the process dynamics.

So, objects keep only the pertinent information from system requirements, all the complexity of processing and dynamics being deported to the Inference engine.

V. The COD Approach

We consider this approach to be very relevant for non-IT engineers, because it allows building a model of the system which is closer to the system requirements: indeed, data and conditions-on-data are necessarily described in the system specifications.

According to this approach, with an appropriate environment, **the description of data is sufficient to “run” the system.**

The first step of the approach, according to classical Object-Oriented Analysis methods [5], [7], is then to determine the “good” set of objects that will embed data and CODs.

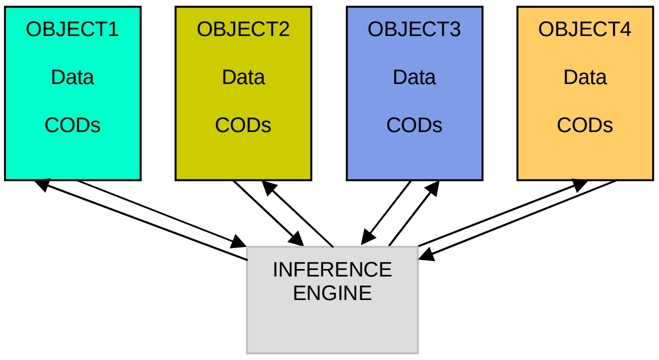

Fig. 3 Structure of a System in the COD Approach

Once the objects are defined, their data and CODs are described by using a simple "COD language".

For example, here is the description of an object TANK (parts of lines after "//" are comments):

```
TANK //Name of the object
  contents = 100               //data
  F_VALVE = open => contents++ //COD
  E_VALVE = open => contents-- //another COD
```

Once the objects are described in "COD language", the central inference engine, containing an equation solver, turns and examines objects one after another, collecting all CODs for each object. The left part of each COD, i.e. the left side of the "=>" symbol, is evaluated.

If the corresponding data, with the required values, can be found in some object, the "conclusion" of the COD, i.e. the right side of the "=>" symbol, is drawn: the conclusion modifies the variable-value couple of the object.

When all the CODs of the current object have been evaluated, the engine goes to examine the next object. The Table below shows the main algorithms for the inference engine and its useful functions.

TABLE III
ALGORITHMS OF THE COD APPROACH INFERENCE ENGINE

```
procedure moteur()
    new_conclusion = False
        For each Object
            For each COD of the Object
                premise = left_part(COD)
                conclusion = right_part(COD)
                if verified(premise)
                then draw(conclusion)
                     new_conclusion = True
                end if
            end for
        end for
    if new_conclusion = True then moteur()

function verified(premise)
    verif = False
    For each Object
     For each equality of the Object
      //An equality = a couple variable-value
      //call of the equation solver
      if solved(equality, premise)
      then verif = True
      end if
     end for
    end for
    return verif

procedure draw(conclusion)
    For each Object
     For each equality of the Object
        if left_part(equality) =
           left_part(conclusion)
        then allocate(variable_of(equality),
             value_of(conclusion))
        end if
     end for
    end for
```

Implementing these algorithms, associated to a user-friendly interface, allows having at disposal a powerful environment for test, simulation or validation of diverse kinds of complex systems.

VI. Developing an Example

Developing a simple example will show the detailed approach and will allow us to present the "COD language" in order to describe data and CODs inside objects.

We use here the specification of a very simple industrial process control system, presented by Paul T. Ward in one of his books [10].

Requirements of the Control System:

A thermal reaction consists in maintaining a given quantity of liquid reagent at a temperature H for a duration T.

The filling of the reactor begins when the "Start Filling" command is given by the operator.

When the level of reactant in the reactor reaches the value N, stop the filling (i.e. close the filling valve) and switch on the heating element.

Trigger the timer as soon as the temperature of the reactant in the reactor reaches the H value.

The timer is triggered by a specific command which is associated with the duration T of the reaction.

Maintain the temperature of the reagent at the constant value H for the period of time T.

As soon as the duration T has elapsed, turn off the heating element and evacuate the reaction product to an external storage tank.

Safety requirement:

Reactor filling can not start if the reagent level in the filling tank does not reach the minimum value M.

In this case, the system must detect it as soon as possible and produce an alarm in response to the "Start Filling" command.

The first step of the COD process is to determine the objects and their relationships. Links between ObjectA and ObjectB are essential to reveal if ObjectA knows the data of ObjectB.

Here is the Structure of the Control System (the logical name of each OBJECT is given in parentheses):

The System is made of a Reactor (REACTOR), two tanks (Filling tank FTANK and storage tank STANK), a Heating element (HEATER), valves (Filling valve of the Reactor FVALVE and Emptying valve EVALVE), a timer (TIMER) and alarm (ALARM).

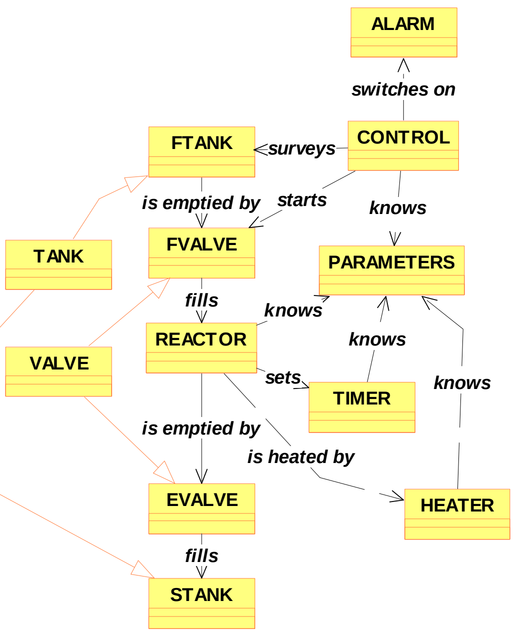

Fig. 4 Object model of the Control System

In addition to these concrete objects, two abstract ones are required: parameters (PARAMETERS), that keeps the values of the reaction parameters (M, N, H and T), and a controller (CONTROL) which surveys the minimum level in the FTANK and sends the "Start Filling" command. Fig. 4 shows the complete UML [11] object model of this simple Control System.

After defining objects and their relationships, the second step of the COD approach is to allocate data to each object, according to the specified requirements.

TABLE IV
OBJECTS AND DATA

OBJECT	**Data**	**Values**
ALARM	status	ON, OFF
CONTROL	Start Filling	
HEATER	status temperature	ON, OFF
PARAMETERS	M, N, H, T	
REACTOR	contents temperature	
TANK	contents	
TIMER	status time	SET, ON, OFF
VALVE	status	open, close

The following step is to complete this Table with CODs. Conditions on data for a given object are determined by its own data and by its relationships with its neighbors. For example, FTANK is a TANK (inheritance) and has a variable "contents". FTANK knows FVALVE, so, the COD concerning "contents" can be a condition including the data of FVALVE, that we write like this in "COD language":

```
FTANK
  contents = 100
  FVALVE = open => contents--
```

Table V shows the complete data and CODs for all objects. Note that "ALARM = ON" is a simplified notation for "ALARM.status = ON" and "temp" is equivalent to "temperature".

TABLE V
OBJECTS, DATA AND CODS

OBJECT	**Data**	**CODs**
ALARM	status = OFF	
CONTROL	Start Filling	FTANK.contents < M => ALARM = ON FTANK.contents >= M => FVALVE = open
HEATER	status temperature	status = ON => temp++ status = OFF => temp--
PARAMETERS	M, N, H, T	
REACTOR	contents temperature	FVALVE = open => cont++ cont = N => FVALVE = close AND HEATER = ON AND TIMER = SET TIMER = ON => temp = HEATER.temp temp < H => HEAT = ON temp >= H => HEAT = OFF TIMER = OFF AND HEATER = OFF => EVALVE = open EVALVE = open => cont--
TANK	contents	FVALVE = open => contents++ EVALVE = open => contents--
TIMER	status time	status = SET => time = T AND status = ON status = ON => time— time = 0 => TIMER = OFF AND HEATER = OFF
VALVE	status	

The last step is to write the COD language from this list (please see Table VI below).

Please note that the objects are listed in the Table in alphabetical order, and COD language will be processed by the Inference Engine in the same order, for example. In fact, the order in which the Engine examines the objects has no importance.

However, the process control demands a rigorous order of execution: first, the Reactor must be filled. Then, the heating system must heat the reagent until the temperature H is reached. Then, the timer is set. And the reaction must last the duration T. At least, the product of the reaction has to be poured in the storage tank.

One of the big interests of the COD approach is that the order of execution is rigorously kept by the CODs embedded in each object. The Inference Engine manages not only the dynamics of the process, but also the order of execution, because it cannot trigger a "conclusion" on data if the conditions on these data are not filled.

TABLE VI
CONTROL SYSTEM COMPLETE COD NOTATION

```
//Control System COD Notation
//---------------------------
ALARM
  status = OFF
//----------
CONTROL
  FTANK.contents < M => ALARM = ON
  FTANK.contents > M => FVALVE = open
//---------------------------------
EVALVE
  status = close
//------------
FTANK
  contents = 100
  FVALVE = open => contents--
//-------------------------
FVALVE
  status = close
//------------
HEATER
  status = OFF
  temp = 18
  status = ON => temp++
  status = OFF => temp--
//--------------------
PARAMETERS
  M = 50
  N = 40
  H = 150
  T = 30
//----------
REACTOR
  contents = 0
  temp = 18
  FVALVE = open => contents++
  contents = N => FVALVE = close AND HEATER = ON AND
TIMER = SET
  TIMER = ON => temp = HEATER.temp
  temp < H => HEATER = ON
  temp >= H => HEATER = OFF
  TIMER = OFF AND HEATER = OFF => EVALVE = open
  EVALVE = open => contents--
//-----------------------------------------
TIMER
  status = OFF
  time = 0
  status = SET => time = T AND status = ON
  status = ON => time–
  time = 0 => TIMER = OFF AND HEATER = OFF
//====================================
```

This "program without code" can be executed with the environment we develop and propose for free download at: **www.programmingwithoutcode.org**

As you can see, the complete notation for this example contains less than 50 lines (including comments). This short content has to be compared with the complete Java program solving the same example we also develop as a comparison (the Java program can be downloaded on the same web site). The Java program has been generated from an object modeler and simulator, and contains 10 classes (Alarm, Control, Heater, mainClass, System, Reactor, Tank, Parameters, Timer and Valve) for a total of 560 lines of code – the class Reactor being of course the fattest!

We hope we have demonstrated through this example the power and simplicity of the COD approach, where data and their conditions are sufficient to model and make "run" complex systems.

VII. Conclusion

We have presented in this paper an approach, a language and an environment that allow non-IT engineers to program the functioning of a system without writing complex lines of code, by using only the data required by the system and their conditions of use. We believe this approach, powerful and simple, provides a complete tool for system modeling and verification or simulation.

The sequel of this work, which is currently in progress and will be presented in a next paper, consists in a complementary approach to analyze directly the text of a system specification and to extract from this text the description of objects with their data and CODs. By this way, we hope to be able to generate the "COD language" in order to feed directly the COD inference engine. We think this means will be an interesting help for system modeling and simulation.

References

[1] Modeling tools like Objecteering : http://www.objecteering.com (access date: May 2017), UML Designer: http://www.umldesigner.org/ (access date: May 2017), Rhapsody: http://www-03.ibm.com/ software/products/ (access date: May 2017), or Interactive Models: http://www.sciencedirect.com/science/article/pii/S1877050915002513 (access date: May 2017)

[2] Ph. Larvet, *Semantic Application Design*, Bell Labs Technical Journal, Volume 13, Issue 2, Summer 2008, Pages 75–91.

[3] John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, ISBN 0-521-78098-5, p.278.

[4] Pierce, Benjamin, *Types and Programming Languages*, MIT Press, 2002, ISBN 0-262-16209-1, section 18.1 "What is Object-Oriented Programming?"

[5] Grady Booch, *Object-Oriented Analysis and Design With Applications*, Addison-Wesley, ISBN 0-8053-5340-2, l5th Printing, December 1998.

[6] Ph. Larvet, *Analyse des Systèmes, de l'Approche fonctionnelle à l'Approche objet*, InterEditions, ISBN 2-7296-0430-8, Paris, 1994.

[7] Sally Shlaer, Stephen J. Mellor, *Object Lifecycles, Modeling the World in States*, Yourdon Press Computing Series, Prentice Hall, 1992, ISBN 0-13-629940-7 p.125, *Forming and Assigning Processes.*

[8] Ph. Larvet, *Systèmes Experts en Turbo-Pascal*, Eyrolles, Paris, 1987

[9] Jocelyn Ireson-Paine, *What is a Rule-Based System?*, Feb. 1996, http://www.j-paine.org/students/lectures/lect3/node5.html (access date: May 2017)

[10] Paul T. Ward, Stephen J. Mellor, *Structured Development for Real-Time Systems*, Yourdon Press, Prentice Hall, 1986, ISBN 0-13-854787-4, Vol.I, *Introduction & Tools.*

[11] UML, *Unified Modeling Language*, http://www.uml.org/ (access date: May 2017)