Proposal for a Multilanguage Teaching Programming Environment

Rafael Fontao, Gustavo Goñi, Guillermo Kalocai y Gustavo Ramoscelli[1]

***Abstract* - This paper describes a proposal for a programming methodology which has been applied by the authors in their own programming work and in the teaching process as a pedagogical resource for first year courses on computer programming for electrical and electronics engineering students at Universidad Nacional del Sur, Argentina.**

The original idea was set up for sequential processes, as it was for typical calculus engineering problems or simple data administration. Currently the idea follows modern concepts of computer programming methodologies. In particular, "extreme-programming" principles, like customer involvement, which may naturally accomplished by this methodology.

The central core idea is describing any task by specifying its structure (what has to be done) separately from the implementation (how it is accomplished). The notion of finite state automaton, which is a conceptual device well fitted for electronics engineering students, is taught from the beginning and the end user or client may be part of the programming team while designing a structured solution.

A version of this environment is being applied for the new academic curricula programs. We first present a learning environment to describe the behavior of any given task separately from its implementation and then we apply a methodology to guide this description for a particular implementation.

Index Terms - Task conception, teaching programming, description language, e-learning.

INTRODUCTION

Current software development is strongly influenced by the agile methods [2][3][9] like *extreme programming*, however we agree with D.Parnas (cited in [3]) on the idea that the so called "software crisis" has derived into a chronic problem. As educators we need to understand clearly the new developments at the same time that we teach the basic foundations of computer programming somehow independently of fashionable circumstantial soft/hard developments.
It is a hard work to do.

In this paper we consider that the way a given task is conceived is essential for understanding its inner workings, in order to describe it later in a programming language. However before any formal coding is done there should be a clear understanding of the nature of the work independently of its implementation. The connection between the task description and its implementation depends on the tools that will be used, like an artist conceives his/her work on the basis of the tools to be applied into the artwork. The inspiration found in the bibliography has influenced our approach to teaching programming.

The methodology we show in this paper, SOL (from Structure Oriented Language), was originally set up for the intention to document some existing gaps between de verbal or textual description of a task and the formal description in a programming language [5]. The approach to computer programming teaching that we propose here is oriented to engineering students from non-computer science careers. The methodology is based on the notion of **work**, which is previous to the conception of any program [5]. Even though the initial intention was to separate clearly the way a program is conceived from its implementation, it became necessary to use a precompiler to show (via Pascal) the assignments for the typical engineering problems [7].

In past years an unforeseen problem has appeared: the lack of intellectual training of the students coming from the high schools [8]. However, in the new study program curriculum we are planning to teach Java through this teaching methodology. The idea of a unifying programming environment where different programming languages may coexist, may be of special interest as a means for the reuse and portability of programming systems.

While such applications are natural for the environment presented in this paper, our main goal is the development of a methodology for teaching computer programming to non-informatics students. From the standpoint of education in engineering it may seem as another disciplined approach to teach software development. No formal statistic has been collected so we cannot assess its performance, but the experience collected in its use during the last years indicates that it has worked very well, especially in challenging classes of recent high school graduates. It will be applied in the new curricula study program on electrical and electronic engineering. Moreover, for advanced courses, an e-learning enviromment is under study.

Describing a work: Even a complex work can be described in a sequential way, as for instance in written

natural language. Later, a reader should be able to understand how it works and even to foresee its behavior. It seems that all behavioral knowledge can be transmitted in this way.

There are no limits to the possibility of describing a work by means of a sequential language, even when there are simultaneous behaviors and interactions among the parts of a given task. Analogously, in a movie/novel the author has to describe the arguments as to be understood by means of a sequential view/reading of its work.

Traditionally, at least for non informatics professionals like us (pure electronics engineers), the links between the specification of a task and the program that carries it out, are only expressed as natural language comments. However, few clues survive, in general, that may help to figure out how the programmer conceived the task. The history of the intermediate steps towards the final solution is difficult to trace.

This methodology, oriented to engineering students, tries to fill the gap between the conception of an idea and its implementation starting form the basis that every finite state behavior can be modeled through the behavior of a finite state automaton: a concept well understood by engineering students.

The description of a program as a finite state automaton: From the observation of how a work is done, at least one of a mechanical nature, it may be concluded that a work is the interaction among *actors* in the same way that in a movie the history is played out by actors. The actors do their jobs by applying a sequence of working tools (those that change the properties of the object being transformed by the work) followed by a sequence of tests (using those tools that evaluate the properties of the object or by checking logical conditions on the input). Eventually, both sequences of actions may be empty, but not simultaneously, otherwise the task performed will be the null task [5][6].

Our starting hypothesis was to consider a program as the *description of a work* We find this statement sound and there is not reason to assume it is false. This statement leads us to ask about the nature of a work The more acceptable definition in a dictionary says something like "... the transformation of a thing by the action of applied forces."

By observing how work is done, we may distinguish al least five essential components, i.e. components that must be present in the conception of a work. These are: **Primary material**: it is the initial "thing" in the definition, i.e. the crude material before the transformation is carried out. **Final product**: it is the final "thing", i.e. into what was the primary material turned on after the transformation has been done. **Tools**: these are the artifacts or utensils that have to be applied on the material to do the "transformation". In a broad sense only two kinds of tools are conceivable: work and test tools. Work tools are applied to "transform" the properties of the "thing", and test tools are applied to check out the properties of the "thing" being transformed. **Description of the task**: It is a description or sketch in some language (may be verbal, graphical or mental) of how the tools should be applied to transform the primary material into a final product. And finally, **Mechanism**: it may be conceived as the artifact or device (humans are also included somewhere), which, according to the description of the task, applies the tools to do the work.

Let us imagine a man (say, a caveman) doing a known task such as sharpening the tip of an arrow. Let us try to imagine how he applies the tools (working and test tools) to sharpen an initial stone (primary material) in order to give it such form as to be considered useful as an arrow tip (final product). Even though the simplicity of this notion, it suggests that the nature of a program shares something in common with the task that is being carried out by our primitive hunter. It may be observed that his task consists of an alternate succession of strokes and observations. Strokes to *transform* the stone and observations to *test* how the stone shape resembles an arrow tip.

This allows us to conceive any task as decomposed in a sequence of *elementary tasks*, in which each one is, in turn, a sequence of applications of work tools followed by a sequence of applications of test tools. In the context of a computer (i.e. a running computer) the primary materials are the data in the variables and the tools are the instructions in a computer language. Here again we may distinguish two kinds of instructions: those which transform the contents of variables (such as assignments) and instructions, which test something (conditional statements).

So, by analogy we may call *elementary program* in some language L to a sequence of assignment-like instructions followed by a sequence of conditional statements. Both sequences may be empty but not simultaneously (to avoid the null task).

In fact, our primitive artisan behaves like an actor in a movie called “Show me how you work” but in general there will be more than one actor in its cast. Therefore we may consider a work as carried out by a set of actors communicating between each other.

The rather simple concept of elementary program is the bridge towards the field of finite state machines, since we model an elementary program as a state in some machine. Note that the property of being elementary does not depend, in some level, so much on the task as on the tools that are available. For instance: "search for a given number in an array", is an elementary program only if the language to represent this task allows search operations like this one.

We now have an intuitive idea of what is a program, but we have to turn to discuss what actually is programming. If we say that programming is the task of writing a program, then what is the description of the task of writing a program?

Since human beings are the "mechanisms" that produce such a wonder, we may recognize the initial specifications as the primary material, the human intellect as the tools and the training in orderly thinking (somehow mysteriously) the description of the task of writing a program. In this sense we may state that the different programming techniques are rational intentions to describe the task of writing a program based on some paradigmatic model.

A Finite State Automata Program Model: The teaching experience in *Switching and Sequential Machines Theory* has shown us that the finite state automaton (FSA) is a model well understood by engineering students. Noticing that relatively complex automata are designed and implemented very efficiently on hardware components supports this fact.

The behavior of a FSA is represented by a table in which rows are states (specifying some output to be done) and the columns are inputs or stimulus. There is an initial state (or row) and for each possible input (column) the entry shows the next state of the FSA. Additionally there will be a final state, that when reached stops the FSA.

To explore the advantages of modeling finite state sequential behavior, we want to introduce a model in which a program may be modeled by a FSA. It seems natural to conceive any given task as decomposed into elementary tasks composed of the application of work tools followed by the application of test tools. Work tools are applied to modify the properties of raw materials and test tools allow testing how these properties are. Tools in general may be very sophisticated but the above decomposition will still work under a convenient interpretation of tools and primary materials.

In the case of a computer, the assignment-like statements may be viewed as work tools modifying the contents of memory cells interpreted as raw materials. The conditional statements are test tools, which allow the computer to decide what to do next. Therefore, we may define an elementary program in language L as a sequence of assignment instructions in L, followed by a sequence of conditional statements in L. Note that the property of being elementary does not depend upon the problem itself, but on the level of the language in which the program is written. Let us now conceive the task of an actor in some language L as a FSA in the following way:

i) There is an initial state

ii) Each state is represented by an elementary program in L. A sequence P of assignment-like instructions (work tools) followed by a sequence C of conditional statements (Test tools).

iii) There are two inputs True and False (Y and N). They represent the logical values of the condition set at the end of each elementary program. (For the sake of simplicity we will restrict the model to sequences of at most only one logical conditional statement).

iv) The current state is the one being executed and the next state will be determined by the logical input entered at the end of each elementary program.

It is known, from the structured programming principles on writing programs [1][4][10][11][12][13], that a program design is made by successive steps or refinements. At each step a module is refined by explicating its behavior. This refinement process will continue until the whole program is written in a practical programming language. We will now state a similar procedure to write a program as a FSA. Let us call PPL the *practical programming language* in which the program will finally be written. The programming procedure is as follows (let us assume an imperative language):

Step 0: Choose the language you better understand, for expressing your ideas. Conceive the whole program in this language (L) as a one state FSA. Customers have to be engaged in some way.

Step 1 to n: If there is a state which cannot be clearly written in PPL, then conceive it in L as a FSA. (Therefore a state in the automaton is replaced by new states forming themselves a FSA. As a clarity rule conceive each FSA having as few states as possible.)

Step n + 1: (At this step all the states should be "clearly" written in PPL).

Write the code in PPL for all states. (This step is for developers only).

In this model the control structure of a program (steps 1 to n) is written separately from the rest of the statements (step n+1). The control structure, in turn, is modeled by the state transitions of a FSA. Each state, stating in some language the purpose of its task, is refined either by statements in PPL (at step n+1) or by a new FSA. Therefore, a program may be seen as a hierarchy of automata where the dependence relation can be represented by a tree. The root is at one state automaton describing the global purpose of the program. Then each refinement adds several nodes to the tree and this process continues until all the states (or leaves of the tree) can be modeled by a sequence of statements in PPL. In the next section we present a language definition suitable to deal with our proposed model for programming.

Therefore, in this model a program may be written in successive steps translating the description of a task from a natural language into a formal language oriented to a computer. Note that a program conception is a chain of refinements written first in L then in PPL.

When does the change occur? If the PPL is applied from the beginning we are just doing the standard coding in PPL. Otherwise, the shift of the language programming, from L to PPL, will obey to the learning process suggested by this methodology.

A Meta tool for an existing programming language. The language SOL is oriented to show explicitly how a programmer conceives the structure of a program. Let us present the SOL syntax and semantics and then

examples of its application. The language SOL will be at first conceived as a system to produce code into an existing PPL.

Syntaxis and semantic of SOL. Using BNF production rules the syntax and semantics may be given as follows.

Preliminary notes:

FSA: Stands for Finite State Automata

PPL: Stands for Practical Programming Language

<comment> are documentation comments ended with "end of line" o any other delimiting symbol as used in the standard programming languages. It will depend on the precompiler design.

The semantic rules corresponding to the syntax sentences are being written double quote delimited.

<SOL>::=SOL <comment > < Work_description > <Implementation> END SOL

"The solution of a task has two well differentiated parts. First, the description of the task and then the implementation in some PPL."

< Work_description >::=<Actor> FIN |<Actor> < Work_description >

"The task description is a list of characterizations of the work to be done by each actor (possibly only one) ended by the word FIN. The first actor is the principal actor, or the work starter. The rest may be added arbitrarily. For programming style use alphabetical order."

<Actor>::= ACTOR < Actor_ID >
<actor_control_structure> END ACTOR

"The coding of an actor is delimited by the words ACTOR and END ACTOR"

< Actor_ID >::= unique word identifying the actor.

<actor_control_structure>::=<state_refinement>FIN | <state_refinement><actor_control_structure>

"The actor description is an ordered list of refinements ended by the word FIN. The creation order of the refinements follows, necessarily, a descendent route. So, a refinement tree grows down. The leaves of this tree are the final refinements, which do not require further detail and are implemented by the PPL directly."

<state_refinement>::= REF <depth_identifier><FSA_definition> END <depth_identifier>|
REF <depth_identifier> IS ACTOR < Actor_ID>: <depth_identifier> <comment> |
REF <depth_identifier> IS ACTOR < Actor_ID>: CREATE |
REF <depth_identifier> IS ACTOR < Actor_ID>: DESTROY [ALL]

"The state refinement starts with REF and an identifier and then there are four interpretations. 1) If this refinement belongs to the actor that is being refined, then it follows the FSA description of the refinement ended with the word END and the same identifier. 2) If the work to be refined belongs to other actor action, then it follows IS ACTOR with the identifier of that action. 3) If it refers to the creation of a new version of an actor then it follows IS ACTOR then the identifier for this actor followed by CREATE. 4) If it refers to cancel or destroy an actor (previously created) then it is followed by the word IS ACTOR then the word DESTROY. The option ALL refers that all copies of the actor are being destroyed. Note that in each reference IS ACTOR both the emitter and the receiver are perfectly identified. All actors have to be created by other actor. The only exception is the initial actor."

<depth_identifier>::= null | <identifier>

"A depth identifier follows the classical rules for tree depth identifier (Dewey notation). In particular if there is not an identifier it means that it is the first refinement"

<identifier>::=<positive_integer > | <positive_integer> . <identifier>

<FSA_definition>::=<state_description>|<state_description><FSA_definition>

"A FSA is described by a list of state descriptions of its states. The states are consecutively numbered up from 1"

<state_description>::= <positive_integer>[DO]<task_description> THEN <next_state_list>

<task_description>::= it is a description in natural language of the purpose of this elementary task

"The task description is identified by a positive integer (starting from 1) followed optionally by the word DO and the description in natural language of the purpose of the state task"

<next_state_list>::= <next_state_identifier> |<next_state_identifier> OR <next_state_list>

<next_state_identifier>::= <positive_integer>| EXIT <positive_integer>|STOP <positive_integer>::= <digit> |<digit><positive_integer>

<digit>:: = 0 | 1 | 2 | 3 | 4| 5 | 6 | 7 | 8 |9

"After the word THEN it is specified the next action to be taken by the FSA. 1) If it is a positive integer it refers to that state in the same FSA. 2) If it contains the word EXIT it refers to that identifier in the previous FSA (parent node). 3) If it is the clause STOP it forces to end."

Up to this point the <Work_description>, hopefully, describes the global behavior of the work. Note that only a few special tokens are used to picture all the work structure (ACTOR, CREATE, DESTROY, DO, END, EXIT; IS, FIN, OR, REF, SOL, STOP, THEN).

The programmer's team has now a formal description of the task to be done by all the actors of the work without having written any PPL implementation. This version of the work can be shared among other personnel for discussion and can be even shared in a learning environment without having any clue about how to implement it. The global description is obviously not unique but it is a good start for sharing among non-informatics professionals since the syntax of SOL is very simple. The customer may share this description and test it on a special module that replaces the unfinished refinements (leaves of the tree) by their

simulation. This feature allows running partial completed programs to be approved by the customer.

Syntax and semantic related to the PPL

<Implementation>::=<Global_implementation><Actor _implementation_list >

"In particular for teaching it is advisable to start with Pascal. So, this part may look somewhat tied to this language. This implementation has two parts: the global implementation and a list of the actor implementations."

<Actor_implementation_list>::=<Actor_implementation> | <Actor_implementation> <Implementation>

<Global_implementation>::= GLOBAL <global _declarations>

<global_declarations>::= global declarations required by the PPL

<Actor_implementation>::=ACTOR ACTION <actor_ID> <actions_description>FIN

<actions_description>::= DEC <data_declarations> < actions_list >

<data_declarations>::= particular declarations of the actor (if they exist at all) required by the PPL

<actions_list>::= <action> FIN | <action> <action_list>

<action>::= ACT <identifier> <output_state> <exit_condition>

"The action identifier follows the same rules for refinement identifiers. In fact, an action can be considered as a refinement in the tree (a leaf) similar to the previous stage but this time written in a programming language"

<output_state>::= PPL sentences that implement the ouput of this state

"This is the traditional writing of any part of a program. The output of a state is the set of all the required sentences in PPL that implement the task of the state. At this stage the programmer has the freedom to implement parts which in the previous stage refinements are described globally and not refined."

<exit_condition>::= NEXT | NEXT (<logic_condition>)

"The question about what to do next should be documented in the refinement stage. The NEXT clause should not be confused with the same reserved word that eventually may be used by the PPL. Otherwise use another token instead of NEXT.

<logic_condition> ::= logic condition allowed in the PPL.

The SOL Environment: A first version of the SOL environment has been implemented in DELPHI and it consists of some modules, which are used in the introductory courses on programming languages. We are teaching Pascal as introduction and we are planning to teach Java in second courses for engineering programming. In particular, we are planning to offer some of these courses as an e-learning resource for teaching. The modules are: 1) A SOL code editor, 2) A PPL code editor where there is the generated code, 3) A syntactic analyzer for SOL writing errors, 4) An actor simulator 5) A SOL precompiler.

The precompiler has two parts: the first one generates the state transition tables (or refinement tables) with the information of each actor given in the control structure. The second part generates the code according to the instruction given in the action description adding the control flow defined by the tables.

The actor simulator allows the execution of undefined actors (those that have not yet being written) simulating its behavior like in the classic "under construction" sign. The user has information about the task to be accomplished by the particular actor and a list (Next state list) of options to choose one and manually continue the task.

Example: let us use the methodology to show how to describe the behavior of a known system, which we use, almost everyday: A small university restaurant attended by 2 bartenders, a cashier and 3 chefs.

The courtesy and/or personal necessities are not considered. This example is for didactical purposes only to show the description power of SOL syntax, since for a restaurant only the cashier may be implemented on computer systems.

```
SOL   The work at a simple Restaurant
ACTOR   Restaurant
REF
   1 Activate bartenders
  THEN 2
   2 Activate cashier
   THEN 3
   3 Activate kitchen
   THEN 4
   4 Open Restaurant
   THEN 5
   5 Close Restaurant
   THEN STOP
END
REF 1
    1 Activate bartender Juan
    THEN 2
     2 Activate bartender Pedro
    THEN EXIT 1
END 1
REF  1.1  IS ACTOR  Bartender: CREATE    Juan
REF  1.2  IS ACTOR  Bartender: CREATE    Pedro
REF  2  IS ACTOR  Cashier:  CREATE    Unique
REF 3
      1 Activate  first chef
       THEN
       2 Activate second chef
       THEN 3
       3 Activate third chef
       THEN EXIT 1
END 3
REF  3.1  IS ACTOR Chef:  CREATE    The first
REF  3.2  IS ACTOR Chef:  CREATE    The second
```

```
REF  3.3  IS ACTOR Chef:  CREATE   The third
REF 5
      1 Close kitchen
     THEN 2
      2 Desactívate bartenders
      THEN 3
      3 Ask for cashier report
      THEN 4
      4  Close  cashier
       THEN  EXIT 1
END  5
REF 5.1 IS ACTOR Chef:  DESTROY ALL
REF 5.2 IS ACTOR Bartender:  DESTROY ALL
REF 5.3 IS ACTOR Cashier: 4  Cash report
REF 5.4 IS ACTOR Cashier: DESTROY
END ACTOR   Restaurant
ACTOR   Bartender
REF
    1 Pay attention for client/cuisine requests
     THEN 2 or 3 or 4
     2      Attend new request from the client menu
      THEN 1
     3      Attend for cuisine ready dish request
       THEN 1
      4      Attend for closing the bill
        THEN  1
END
```

In this way, we can continue the refinement process specifying the behavior of the rest of the actors. The leaves of this refinement tree show the work or actions that the programmer considers can write down in the PPL. This tree is the common language among the clients and programmers.

Conclusions

The environment proposed here is just another idea for teaching computer programming at the first university levels. It summarizes our view of the programming teaching experience. As a team we have not been engaged in any huge software project, rather we have been involved in teaching the first steps of programming to a huge audience of engineering students. The students' feedback is analyzed by means of their final exam assignments. The philosophy behind this approach to teaching has guided our own professional programming work for more than 20 years. Individually we have been engaged in the software development of medium size engineering problems and business applications. For the first time this paper introduces the actor conception and puts together the concepts proposed in previous work of the group.

ACKNOWLEDGMENT

The authors are indebted to Dr. Fernando Tohmé and Dr. Claudio Delrieux from Univ. Nacional Del Sur and Dra. Virginia Cano from WBL Open & Distance Learning Consultants, Scotland, for a critical reading of the original paper.

REFERENCES

[1] *ACM Computing Surveys: Special Issue on programming*. Vol 9,4 (1974)

[2] Agerfalk,P.J. & Fitzgerald, B. (ed). "Flexible and distributed software processes", *CACM* Vol 49,10 (2006)

[3] Beck, K. "Extreme Programming Explained" Addison-Wesley (1999).

[4] Dahl, O.J., Dijkstra, E.W. & Hoare, C.A.R. - Structured Programming.Academic Press, 1972.

[5] Fontao, R.O., Ardenghi, J.R. y Arroyo, E.H. "Sobre una Metodología de Programación". Segundo Simposium sobre Aplicaciones de la Ingeniería Eléctrica y Electrónica, SIEEM 77, Monterrey, México 1977.

[6] Fontao, Rafael O, Kalocai, G., Ramoscelli, G. Y Goñi, G. Una propuesta de investigación sobre "A PROGRAMMING METHODOLOGY". Workshop on Programming Methodology - WG 2.3 - IFIP, Tandil September 2000.

[7] Fontao, Rafael O, Goñi, Gustavo. "SOL: Un ambiente de programación"
CACIC 2003. IX Congreso Argentino de Ciencias de la Computación. La Plata, Octubre de 2003

[8] Fontao, Rafael O., Repetto, Andrés P. Y Goñi, Gustavo M. "Una experiencia en la enseñanza universitaria de primer año". Jornada de Socialización de Experiencias-Programa de Articulación UNS-Polimodal, Bahía Blanca abril 28 de 2005.

[9] Jeffries, R.E. "Extreme Programming Installed". Addison-Wesley (2001).

[10] Presser, L. Structured Languages. *AFIPS Conf. Proced. 1975*. AFIPS Press Vol 44, pp 29l-292.

[11] Voigt, S. "Program Design by a Multidisciplinary Team". *IEEE Trans. on Soft. Eng.* Vol. 1,4 (1975) pp 370-376.

[12] Wirth, N. Systematic Programming: An Introduction. Prentice-Hall 1973

[13] Wirth, N. "On the Composition of Well-Structured Programs." *ACM Comp. Surveys*, Vol. 6,4 (1974) 247-259.

[1] Authors are at Dpto. de Ingeniería Eléctrica y de Computadoras, Universidad Nacioanl del Sur. Argentina. E-mail: fontao@uns.edu.ar, gmgoni@criba.edu.ar, ingelec@criba.edu.ar, ramoscel@criba.edu.ar. This work was made by a grant PGI 24/K028 in the framework of PROMEI C1J2 "Ciclos Generales de Conocimientos Básicos – Carreras de Ingenieria".