PUC

Series: Monografias em Ciência da Computação,
No. 17/90

MICRO-ETHOS: AN EXPERIMENTAL METHOD-BASED ENVIRONMENT GENERATOR

Armando M. Haeberer   Jorge L. Boria
Adolfo M. Kvitca   Hernán Cobo
Daniel E. Riesco   Luis E. Roqué
Germán A. Montejano   Silvio B. Grilo
Edson G. Schubert

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453
RIO DE JANEIRO – BRASIL

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC RIO, Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
BRASIL

Tel.: (021)529-9386 TELEX:31078 FAX:(021)274-4546
BITNET: userrtlc@lncc.bitnet

PUC Rio - Departamento de Informática

Series: Monografias em Ciência da Computação, Nº 17/90
Editor: Paulo A. S. Veloso November, 1990

MICRO-ETHOS: AN EXPERIMENTAL METHOD-BASED ENVIRONMENT GENERATOR

Armando A. Haeberer Jorge L. Boria
Adolfo M. Kvitca Hernán Cobo
Daniel E. Riesco Luis E. Roqué
Germán A. Montejano Silvio B. Grilo
EDson G. Schubert

This work has been partially sponsored by FINEP

μ-ETHOS
AN EXPERIMENTAL METHOD-BASED ENVIRONMENT GENERATOR.‡

Armando M. Haeberer[1], Jorge L. Boria[2] , Adolfo M. Kvitca[3] , Hernán Cobo[2] , Daniel E. Riesco[2], Luis E. Roqué[4], Germán A. Montejano[4], Silvio B. Grilo[5], Edson G. Schubert[6].

Abstract
We describe μ-ETHOS, an experimental environment generator which supports the specification of method–based environments. We provide the definitions for the different methods' parameters such as the external syntax of tools, their inter-relationships and the rules governing the heuristic of their application to an specific problem domain. The kernel of μ-ETHOS is a graphical editor for graphs which allows for the definition of a semantic network for the instantiation of editors. This is achieved through changes in definitions, including those that support the relationships between tools. It is shown in this paper that many interesting method-based environments can be produced through the process embodied in μ-ETHOS. In particular the following environments are built: data-flow diagrams, data-flow diagrams with abstractions, control-flow diagrams, entity-relationship diagrams, state-transition diagrams, office-automation semantic network, Petri-nets, text editors, execution graphs, symbolic data-flow diagram processors, symbolic Petri-nets processors, and a data-flow virtual machine for matrix calculus.

1. Introduction

The μ-ETHOS project was a spin off of the ETHOS [TLH89] project, a part of the Argentina-Brazil Program for Research and Advanced Studies in Computer Sciences. It started in May 1987 and ended in February 1989, with a final report [Bor89]. Using a contemporary classification of software engineering environment architectures [Pen88] as a guidance μ-ETHOS can be seen as an attempt to support many methods by providing a consistent user interface to these methods while adopting a unifying software development model that covers most of the phases of the software development process. It can both be used as an open and as a closed architecture. Figure 1.1 shows the architecture of μ-ETHOS in the style proposed in [Pen88].

The software development process model used in μ-ETHOS involves four main activities: the real problem extension; the requirements specification (the informal description of the problem requirements); the formal specification (their formal description; and the program as the final product of the process. The development process divides the path from verbalization to program implementation into two "legs", called abstraction and reification, respectively in [Leh84].

In the context of the previously defined development process μ-ETHOS is concerned with methods supporting requirements specification. μ-ETHOS is therefore a tool for the generation of method-based environments. It is neither a method nor is it committed to a given method. Nevertheless, a bias introduced by some of the design choices, i.e. the kernel, allows certain methods to be more easily defined than others by the system tools.

The implementation approach taken in μ-ETHOS was object oriented [Ste85]. Its kernel is a graph editor, which constitutes the class of primitive editors. Why graphs? Because inheritance trees, semantic networks [Sow84], viewpoint abstraction trees, relationships between tools, and

‡ This project was supported by the Argentine-Brazilian Program of Research and Advanced Studies in Informatics (PABI).

1 Depto. de Informártica, Pontifícia Ubiversidade Católica do Rio de Janeiro. Rua Marquês de São Vicente 225, 22453, Rio de Janeiro-Brasil.
2 Depto. de Computación y Sistemas, Universidad Nacional del Centro de la Provincia de Buenos Aires. Chacabuco 399, 7000 Tandil, Argentina.
3 Depto. de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. Pabellón 1, Ciudad Universitaria, Buenos Aires, Argentina.
4 Universidad Nacional de San Luis. Chacabuco y Pedernera, 5700, San Luis.
5 Pro-Retoria de Pesquisa, Universudade Estadual de Campinas. Caixa Postal 6001, 13100, Campinas, Sao Paulo, Brasil
6 Universidad Federal de Rio Grande do Sul. Caixa Postal 1501, 90001, Porto Alegre, Brasil.

even end-user documents in many methods, can be modelled as graphs, and all of them are important concepts in this context.

The central architectural idea is conceptual integrity. The kernel of μ-ETHOS allows for the development of a meta-tool known as the basic environment. With it, the environment designer develops a CASE tool. With the CASE tool the software engineer designs the systems requirement specifications.

From the environment designer's viewpoint, all tools are developed by specifying [illegible]ir properties in the form of a semantic network. When networks are not sufficient the designe[illegible]ay resort to rules [Fro86]. This approach rests upon the execution of an (overloaded) instance [illegible] an editor. The semantic actions in all levels are expressed as operations on the editor whic[illegible] are syntactically similar. The semantic differences depend upon the particular instance of the[illegible]itor in use.

To investigate the potential of the above proposal, the following environments were de[illegible]oped using the basic facilities:

- editors for tools such as data flow diagrams, entity-relationship diagrams, P[illegible]-nets, state-transition diagrams, and for cliche-based text;
- symbolic processors for documents used in some of the above methods;
- integrated methods involving editors and symbolic execution processing facilities.

The particular implementation of μ-ETHOS described in this paper runs on a μ-VAX AI work-station with VMS and those (primitive) software constructs defined without the support of semantic network were written in CommonLISP, extended with our own version of objects and a Prolog inference engine[1].

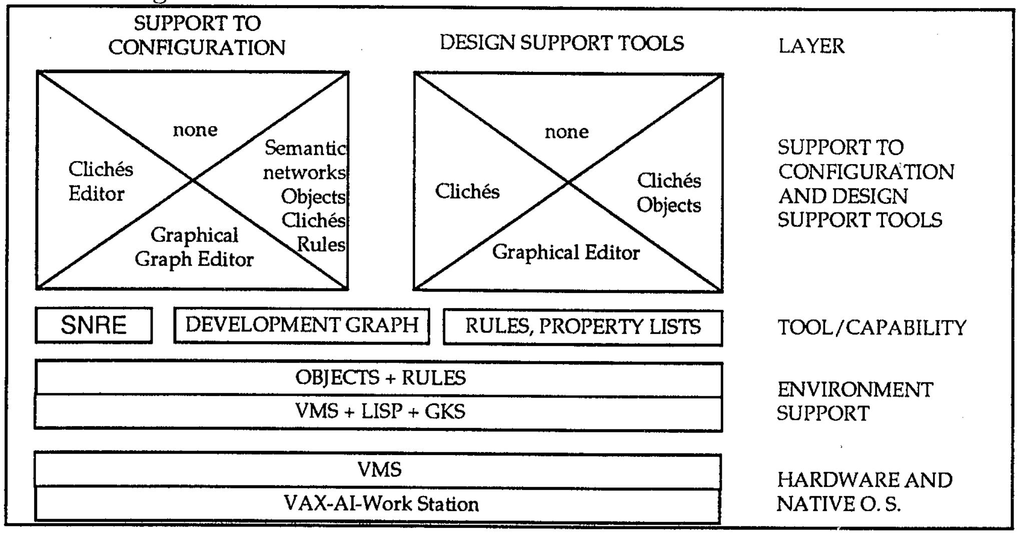

Figure 1.1- The μ-ETHOS SEE architecture.

In the next section the μ-ETHOS Basic Environment's components are presented, section 3 shows some examples of their use and section 4 shows how a given target environment is used .

2. The μ-ETHOS Basic Environment.

The basic environment's kernel was generated from the set of components of a graphical graph editor organized as an object hierarchy, *Egg* for short (figure 2.1.1) which was, in turn, used for defining the semantic network and rules editor (*SNRE*). Both of them where not written with the support of the semantic network. *SNRE* was then used to enrich *Egg* by adding a new relationship (*part-of*) to turn the object *Graph* into a full fledge graph editor. *Graph* constitutes the kernel of μ-ETHOS' environment and it is but the name of the super-class whose semantic is inherited by all objects related to *Graph* through the relationship *is-a* in a semantic network.

[1] The main difference with LOOPS [ ] is the way the rules are expressed externally, see figure 3.3 for an example of the rules in μ-ETHOS.

Together with the class *Graph* there is another primitive class called *Cliché*. A cliché is a frame-like object also written without the kernel support. This class is used for two purposes. On the one hand, the capability for generating graphs oriented editors is complemented by a capability for generating text editors, albeit quite simple ones. These are needed because some information related to nodes and arcs is better expressed by text. On the other hand, we needed the capability to relate actions to elements of a graph.

Although these tools do not constitute all the components of the the basic environment, they are already sufficient to support conceptual integrity. To justify this statement it suffices to point out that the remaining element of the basic environment, an Abstraction-graph editor (*Agraph*), was built from *SNRE* by extending the definition of *Graph* in the basic environment's semantic network.

In summary, the basic environment is made of *SNRE* and a semantic network together with a set of rules defining the classes *Graph*, *Cliché* and *Agraph*.

2.1. The Graphical Graph Editor Component Set, *Egg*. *Egg* is the set of software components that was used for the construction of the graphical graph editor and then reused in the basic environment.We shall describe the way these parts behave in terms of the operations supported by the elementary editor defined with *Egg*, and their effects upon the objects it defines. The editor has six windows, each with a different purpose (see figure 3.2). The main window shows the object to be edited, a top command window lists the accessible commands, in between a pop-up window sends comments from the editor to its user. To the right the icon window displays the figures that can be pasted on the main window. A global window shows the coverage of the main window with respect to the total figure, and on top of it a tiny window shows the file name being dealt with and the ratio of the size of the icons on the main window to a standard size.

The basic commands are CREATE, MOVE, DELETE, EXIT, PROPERTIES. When an editor is instantiated, the list can be extended to suit the new needs without re-programming. The semantic of all but the last commands is the usual one. PROPERTIES will be explained later in this paper.

The editor has two icons: a circle for nodes and an arrow for oriented arcs. In fact "circle" and "arrow" are reasonable choices since they suggest the underlying graph-theoretic concepts. Since nodes and arcs are implemented as objects, their corresponding icons are treated as properties in the property list of the object.

The editor expects the object being defined to be a well formed graph, and signals any attempt to save an object that violates that rule. In this context, graph stands for polygraph, that is, more than one arc connecting the same two nodes are allowed provided they have different labels. This kind of graph is required for the specification of many methods and tools.

The hierarchy of objects in *Egg* is shown in figure 2.1.1, it constitutes the basis for the semantic network that lies at the heart of the *SNRE*.

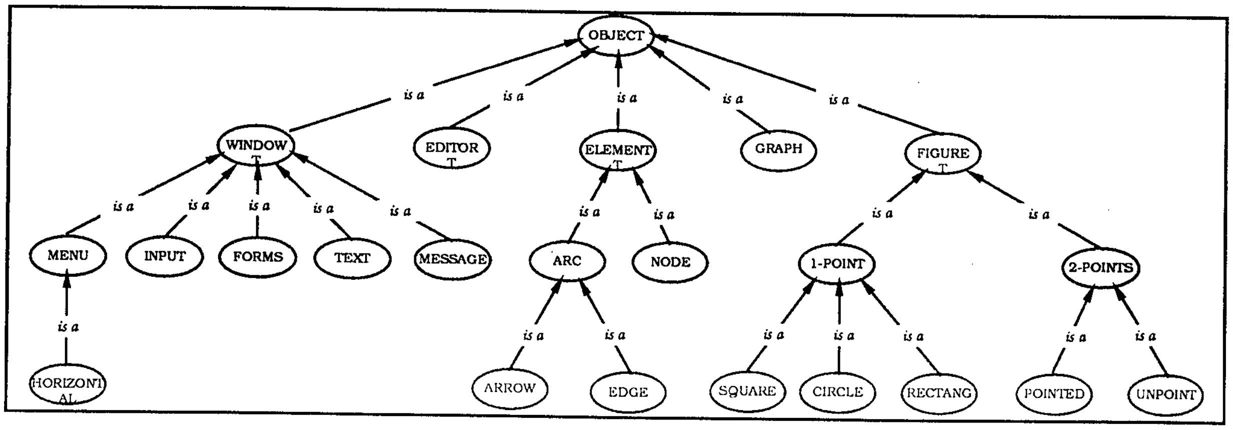

Figure 2.1.1- *Egg* object hierarchy.

2.2. The Components of *SNRE* Editor: Semantic Networks and Rules. As it has been said before *SNRE* is also a software artifact not generated by using semantic networks.The semantic networks editor and the rules editor, were written directly as programs in extended CommonLISP using *Egg* for both graph-theoretic typing and interactive support. In this sense *SNRE* is an extension of the elementary editor built from *Egg*.

In fact, *SNRE* is an editor for semantic networks extended by rules. The networks are themselves well-formed graphs -hence the relationship to *Egg*- where each node is associated to a meta-class in the objects' hierarchy. Creating a node implies the creation of a class or reuse of an existing class. Creating an arc represents the establishment of relationships between nodes restricted to *part-of* and *is-a*. There is, in fact, a third relationship that is not implemented by means of the graphic capabilities of *SNRE*, but that is achieved through the property list of each object created which is accessible from the PROPERTIES command. This design decision was taken so as to free the semantic network graph from having to handle visual complexities of the objects.

The objects in the property list (target objects) of another object (origin) share methods that are invoked from the origin when the corresponding message is received. The property list is a "mailing list" from the origin to the target objects. Typically, an object that can be shown has a figure in its property list.

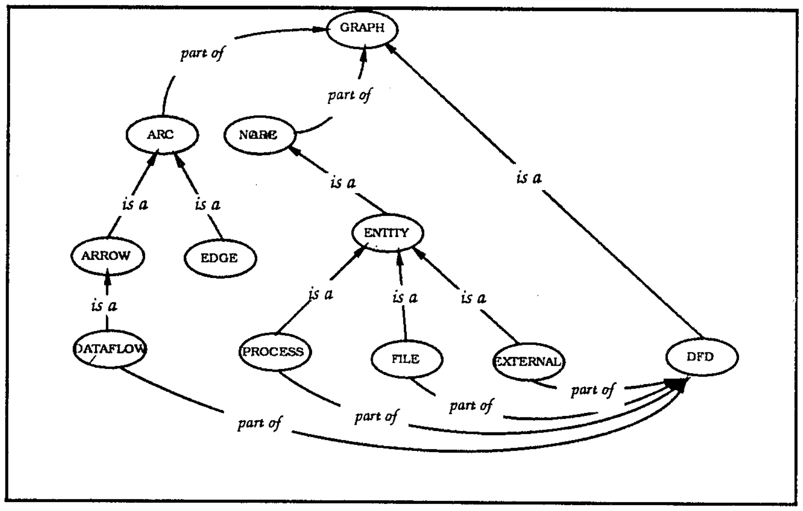

Figure 2.2.1- The semantic network of *DFD*.

The relationship *is-a* expresses simple inheritance. Actually, the same simple inheritance of the underlying object paradigm. When an object is created in a semantic network, *SNRE* assumes that an *is-a* relationship must be defined, and demands from the user the super-class to which the new object is to be linked. Therefore, the *is-a* relationship cannot be chosen from the icon palette (moreover, only under special circumstances can the *is-a* relationship be reassigned). Once an object is created, properties can be assigned to it in a property list. For example, a common property of most objects is the way in which they are to be shown on the screen.

The relationship *part-of* signifies that a given subclass is a component of other classes. For example, *process* ,*entity* and *storage* are subclasses of *node* (i.e., each «*is-a node* »), whereas «*DFD is-a Graph*», and *process* ,*entity* and *storage* are «*part-of DFD* ». As it can be seen in figure 2.2.1, the *part-of* and *is-a* relationships must be consistently used to avoid definition of an ill-formed object, with no semantic meaning.. For that reason *SNRE* prevents such type of definitions.

In a sense, definitions of objects with the *is-a* relationship describes their primitive behavior, yet this potential is only fully realized when the *part-of* relationship links the components to the whole. It is then that the objects can be edited.

Entities created by means of semantic networks inherit the well-formed properties of their super-class, but this could prove not to be enough. Rules are introduced to deal with new restrictions which are imposed to to make the entities well-formed. For instance, since «*DFD is-a Graph*», *DFD* inherits the graph-theoretic well-formedness of *Graph*, that is, its arcs join its nodes (figure 2.2.1). *Flows* are *DFD* ' s arcs, and *processes* ,*entities* and *storages* are also *DFD* ' s nodes. Nevertheless, graph-theoretic well-formedness allows for a *flow* linking a *storage* to a *storage* , and that should not hold. So, a rule must be introduced in this case to prevent it.

A rule should be "understandable" both by the user and by the environment. Hence, a rule is a syntactic structure with direct meaning in the particular domain the system is dealing with and whose semantic in the environment is the intended restriction over the inherited criteria of well-formedness.

To be able to have direct meaning in a particular application domain a rule must be expressed in a natural-language-like syntax and be expressed in terms meaningful in the domain. To allow the interpretation of a rule as a restriction on a pre-existing well-formedness criterion we provide a Prolog-executable grammar for that syntax. Its terminal symbols are meaningful in the domain either defined as part of a semantic network or by synonyms introduced through some rule.

2.3. *Graph.* So far, *SNRE* has been described as a general semantic networks and rules editor. But, since we are involved with the construction of the basic environment of a Method-based Environments Generator the accessible roots of all definable semantic networks should share a conceptual structure, whose underlying semantics is akin to the primitive editor built from *Egg*. In other words, *SNRE* by itself is not an environment generator, but *SNRE* plus the class *Graph* (to be used as the original super-class) is the Graph-oriented Editor Generator that comprise in the Basic Environment.

Since the relationship *is-a* at the *SNRE* level is the same as the one used at the programming language level, the user of the basic environment can access through it all the objects of the *Egg* hierarchy. However, many of these objects have no interest to the typical user.

At the *SNRE* level a semantic network must be defined (figure 2.3.1), relating *Graph* to its components, both in *Egg*'s object hierarchy.

To define a new editor, a user needs only to name it as an object and name *Graph* (or an object that -transitively- *is-a Graph*) as its super-class.

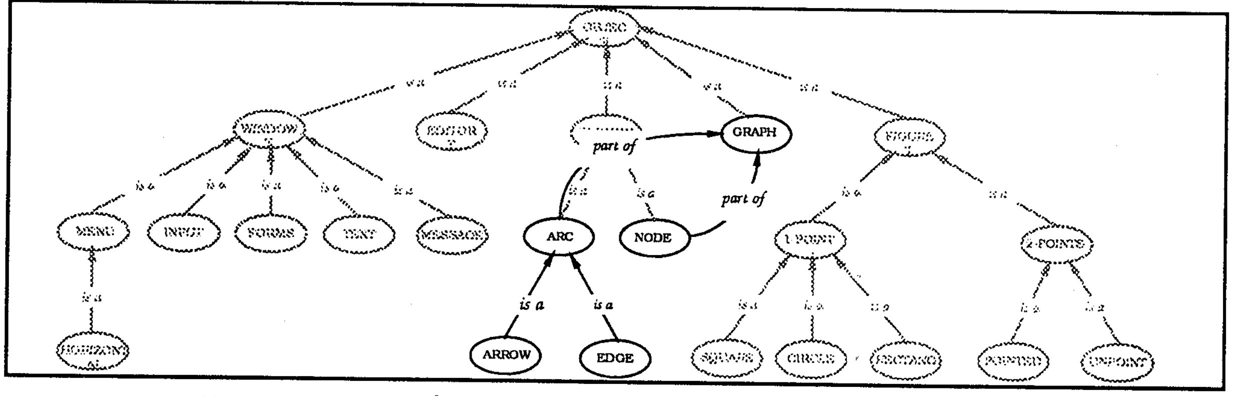

Figure 2.3.1- *Graph*'s semantic network in the *Egg* object hierarchy.

2.4. The abstraction graph *Agraph.* Organized collections of documents, in which some are defined by others, should be related by an abstraction structure which allows them to be traversed, even though each individual instance of a document is an independent object produced by an editor.

In this context, abstraction denotes a process by which a single entity replaces, denotes or names a collection (equivalence class) of entities. Therefore, when applied to graphs we can think of it as a pair of homomorphisms (one for nodes and one for arcs) in which each node in the domain has its corresponding abstraction in the range and each arc in the domain has a corresponding arc in the range iff the nodes that the arc connects in the domain are in different equivalence classes.

Since we allow for multiple arcs with different labels between the same two nodes, the definition must be rephrased as: an abstraction is a composition of two homomorphisms.

The innermost homomorphism (node abstraction) is, therefore, defined as:

- each abstraction equivalence class of nodes in the domain has a node as its image in the range, and
- each arc in the domain has an image in the range iff it connects nodes in different equivalence classes, and the homomorphism preserves arc labels.

The outermost homomorphism (arc abstraction) is defined as:

the nodes in the range are exactly the nodes in the domain, and each abstraction equivalence class of arcs in the domain has an arc as its image; obviously, two arcs can belong to the same abstraction equivalence class only if they join the same pair of nodes and in the same direction.

Notice that this particular definition of abstraction allows us to deduce the innermost homomorphism from the domain and range of the composition and also the outermost homomorphism. In fact, this was used in the development of a minimum implementation for *Agraph*.

Agraph is-a Graph whose nodes are objects of the class *Anode* which *is-a Node* having *Graph* in its property list. *Agraph*'s arcs are abstractions, thus, any two nodes connected by an arc of *Agraph* are the domain and range of an abstraction homomorphism.

By including *Graph* in its property list, each instantiation of *Anode* forces the creation of an element of the class *Graph* (this is a main difference between the property list and the *part-of* relationship). This brings about an interesting effect. Once the *Agraph* environment is operational, the selection of an *Anode* icon creates a node of the class, that accepts EDIT commands, in the sense it accepts an EDIT message from the environment. Upon receiving one message it is propagated to its property list (the extended CommonLISP environment does this). Since *Graph* is in its property list, an instance of it also receives the message. As a result, the environment to edit *Graphs* is called and becomes operational. In this manner the definition of *Agraph* links two editors: when called to edit an *Agraph* the editor defines a graph in whose nodes graphs can be edited.

As shown in figure 2.1.1, an arc is implemented as an object in the hierarchy of objects defined in *Egg*. Its list of properties includes the implementation of the arc abstraction homomorphism (i.e., the outermost homomorphism of the composition), simply by including the labels of the pre-images of the given arc.

Notice that *Agraph* is but an environment defined by means of *SNRE* and the class *Graph*. The μETHOS basic environment provides through *Agraph* the facilities to manipulate abstractions, by allowing the navigation through the different levels by traversing *Agraph* and relating viewpoints by reconstructing the composite homomorphism, as explained above and as shown in figure 2.4.1.

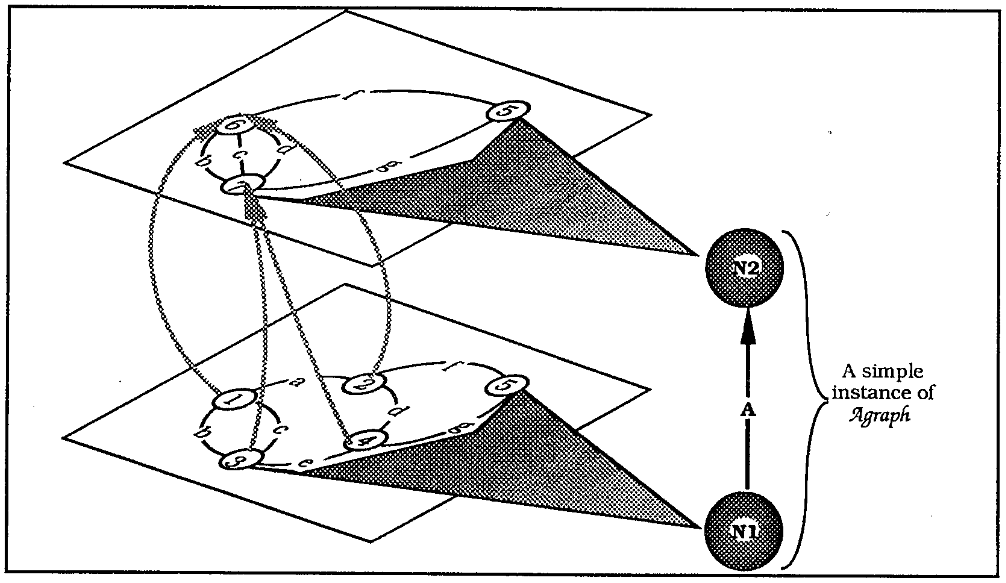

Figure 2.4.1- An instance of *Agraph*.

2.5. *Cliché*. As *Graph*, *Cliché* is a primitive super-class. It is a frame-like structure with four components: a *fixed* part, a *dynamic* part (roles), a *value* part, and a *condition/completion* part [Fro86].

The *fixed* part expresses the cliché's outward appearance.

The *dynamic* part consists of a chain of "slots", each one a terminal value or itself a *Cliché* giving rise to a complex structure, which is richer than plain text[1].

The *value* part contains formulae relating constituents of the different parts. These formulae are evaluated upon insertion of values into referred constituents.

The *condition/completion* part is the set of rules related to a given instance of *Cliché* as an object. In this context some of these rules define the precondition of a particular cliché, i.e., the condition that must hold for the cliché to be executed. Others can be used either to check the condition that must hold after cliché execution (valid termination or post-condition) or to accomplish any required action at completion level. All executions required by clichés are performed by the Prolog engine.

Notice that text editors are easily built from instances of the class *Cliché*, by combining the inherent interactive character of the class with the possibility of linking together many objects in the dynamic part, and the definition of grammars through the rules in the *condition/completion* part.

3. Using the Basic Environment.

Lucena [Luc87] introduced the term "target environment" to denote the environment to be produced by the generator. The basic environment is the development tool of a target environment.

Figure 3.1 depicts the process of generating successive environments to achieve the target environment. Three different roles are recognized in the chain of users. The µETHOS design team, who builds the basic environment from CommonLISP and GKS upwards; the environments designers, who build the target environments; and the "end users", who use them.

As an example we shall develop in the remainder of this section the construction of an environment for editing of DFDs with abstractions, in section 4 will show how such an environment is used.

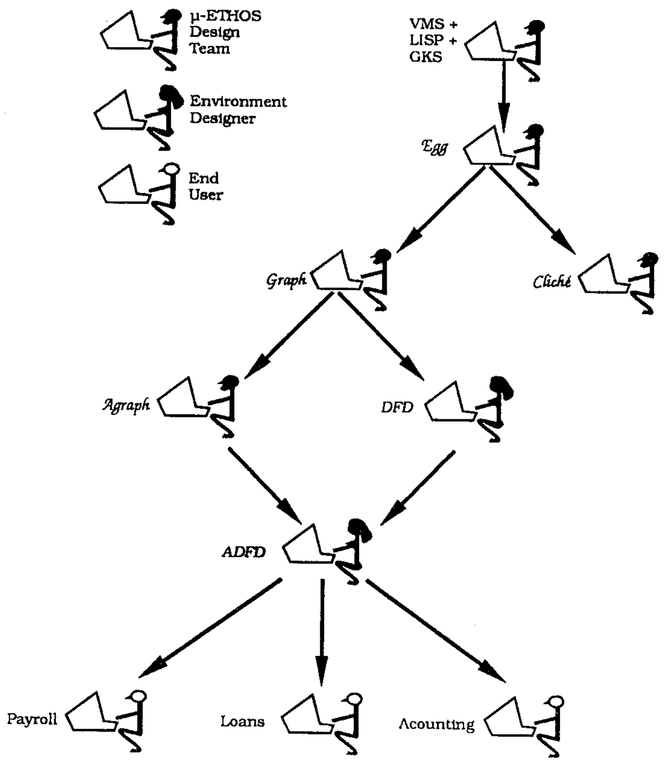

Figure 3.1- Generating environments

The choice of the environment for editing of DFDs with abstractions [DeM78; War84, 85] as an example is justified because of the popularity of the technique and the fact that the meaning of abstraction in the context of DFDs is simple and requires no further explanation. DFDs are extremely popular diagrams that model the behavior of systems through external entities that act as sources/sinks of data, processes that transform data, storages that hold data for indefinite periods, and flows that join all of the preceding elements pairwise, showing how data circulate in the system.

As seen in figure 2.2.1, only flows are arcs, whereas external entities, processes and storages are nodes. Figure 3.2 shows how the basic environment looks when used to define the semantic network that describes a DFD with abstractions.

The basic environment provides the semantic network that defines *Agraph* as mentioned in section 2,4. The user (who at this level is the environment designer) brings the network to the *SNRE* window and adds the nodes DFD, ENTITY, and FLOW.

The designer then further refines ENTITY and FLOW to show that ENTITies are either PROCESSes, STORAGEs or EXTERNAL entities, and that FLOWs are DATAFLOWs. (This apparently useless refinement is justified because some other tools use control flows, also known

[1] A future version of µETHOS should implement this feature using Hypertext.

as event flows, and our designer is aware of that fact). When the new objects are introduced, the environment allows for the designer to change (override) the definitions inherited from the super-class. For example, the class *Node* has the object CIRCLE as *Figure* in its pro[illegible]y list. Unless otherwise stated, any descendent class will inherit a CIRCLE as its *Figure*. Th[illegible]signer therefore redefines the property list for STORAGEs and EXTERNAL entities, since the[illegible] outside appearance is not circular, but leaves PROCESSes' property list untouched.The des[illegible]r now links PROCESS, STORAGE, EXTERNAL (entity) and DATAFLOW to DFD with [illegible] *part-of* relationship, thus completing the semantic network definition.

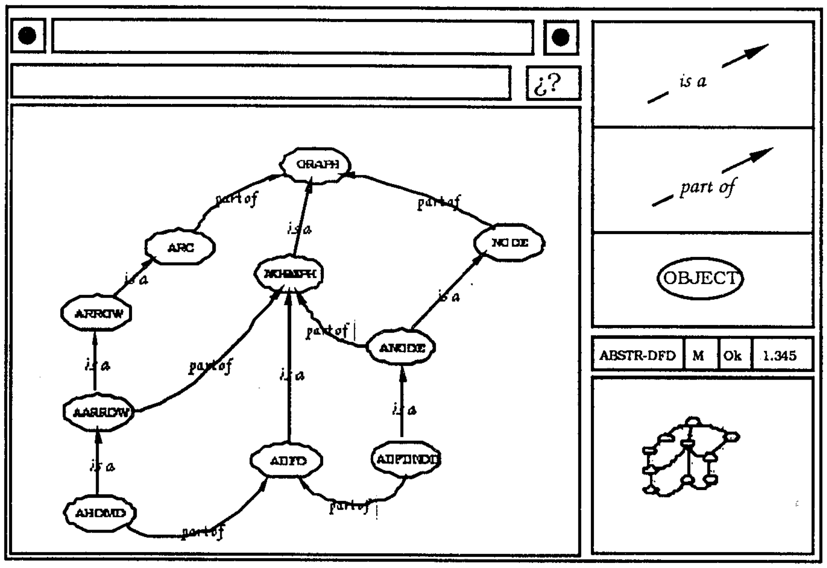

Figure 3.2- The *SNRE* screen when defining *ADFD*.

As far as the external interface goes, that's all it takes. Selecting from a list the recently defined environment will now instantiate an editor that will generate graphical documents with the adequate disposition. Moreover, since the editor inherits capabilities from the *Agraph* editor, abstraction-handling is automatic, and the environment's user can create individual DFDs on nodes of the abstraction graph, visit each DFD by selecting a node of the abstraction graph, or see the way the abstraction holds pairwise between two DFDs by selecting an arc of the abstraction graph. The following section will deal with the way the environment behaves.
However, certain properties of DFDs are not captured by the semantic network as we have so far explained it. The designer can refine the behavior of the environment by adding rules to the RULE-LIST selecting it from the Properties Menu. An example set of rules for DFDs is shown in figure 3.3.

The environment produces natural DFDs, with no reference to Data Dictionaries, a common partner to them. To introduce such links, the designer can use the definition of clichés to implement windows. The procedure is as usual: the *Cliché* object must be included in the property list of the objects that are «*part-of DFD*». In fact, this is the way the environment provides for symbolic execution of DFDs. Each component of a DFD has an associated cliché (and, therefore, possibly a tree of associated clichés). The behavior of the DFD is described Through the associated clichés and, when invoked, executed[1].

4. Using a Target Environment.

Once the target environment has been created, it is ready to produce the desired documents. As an example, an application of the environment for DFDs with abstractions will be discussed.

In figure 3.1 the sequence of environments leading to a target environment was shown. The last line of that figure showed three end users developing documents for Payroll, Loans and Accounting applications on top of the DFD with the abstractions environment, described in section 3.

[1] The reader can infer the implementation of symbolic executors from the description made of the object *Cliché* in section 2.5.

The user sees in the screen an interface that is similar to the ones used by all previous levels of users. In this case, the icons shown are two: one stands for the nodes of the abstraction graph representing the tree of DFDs and the other represents an arc that stands for the abstraction homomorphism between pairs of DFDs.

Upon entering the environment, the user generates a node. This will stand for its first DFD. Immediately, the user can EDIT the node and construct the DFD with all the tools from the previously defined DFD environment. Returning to the abstraction environment, the user can create another node and EDIT it, representing a different abstraction level of the previous DFD. To complete this portion of the abstraction graph, the user must define an abstraction arc that will hold the set of correspondences between arcs in the two DFDs. As it was explained in section 2.4, this information together with the labeling on both levels is sufficient to reconstruct the complete abstraction homomorphism. If EDIT-ing an abstraction node provokes changes to a DFD that renders the homomorphism no longer valid, the environment signals the error to the user and explains its reason.

```
RULE LIST (DFDs)

 LANGUAGE DEFINITIONS:
An entity X receives data from an entity Y if
        entity Y is the source of some flow and
        the same flow has destination in entity X
An entity receives data if
        there is a flow that has the entity as destination
An entity X produces data for an entity Y if
        entity X is the source of some flow and
        the same flow has destination in entity Y
An entity produces data if
        there is a flow that has the entity as source
A process consults a storage if
        the process receives data from the storage
A storage is consulted by a process if
        the process consults the storage
A storage is consulted if
        there is a process that consults the storage
A process updates the storage if
        the process produces data for the storage
A storage is updated by a process if
        the process updates the storage
A storage is updated if
        there is a process that updates the storage

 ERROR DEFINITIONS:

To create a flow is an error when
        there is an error (2) of importance in the flow
To create a flow with source entity X and destination entity Y is an error when
        there is an error (3) of importance in the flow with source entity X and
        destination entity Y
There is an error (2) of importance in the flow if
        there is no process
There is an error (3) of importance in the flow with source entity X and
        destination entity Y if
        none of the entities is a process (X Y)
```

Figure 3.3. Sample of Rules from the Rule List of DFDs.

It should be noticed that all environments constructed with the μ-ETHOS Basic Environment, behave in a cooperative manner, i.e. the inference engine handles the rules both forward and backwards, so that it can explain the reasons for an error in a language that is accessible to the user.

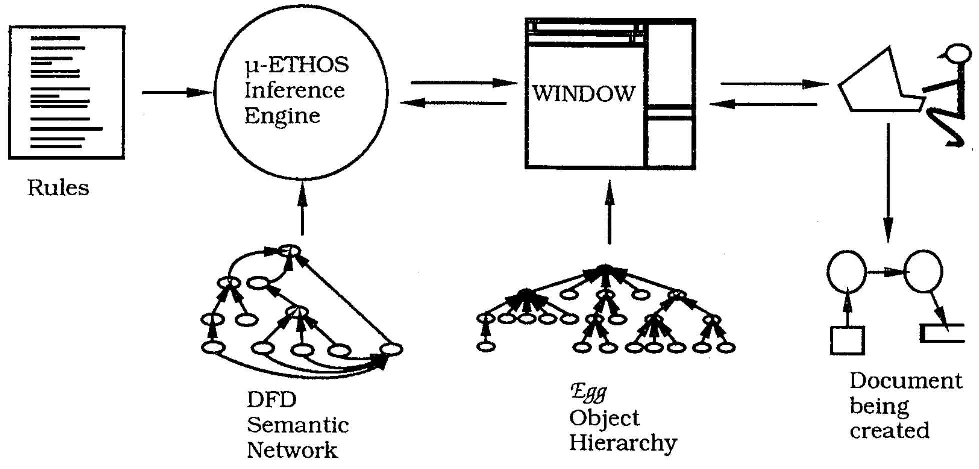

Figure 4.1

To achieve this, the disposition of components acting in the μ-ETHOS environment is as shown in figure 4.1. The Graphics Graph Editor, depicted there as an empty screen (it will always hold the adequate icons and set of commands) , is governed in its behavior by the objects already defined in *Egg* (section 2.1). Since this is but the external interface, the inference engine is invoked to support every interaction with the user. The inference engine "feeds" the semantic network and the rules defined by the environment designer, and precludes any action that could end in an ill-defined object. The same rules are used to explain the actions of the engine to the user (see figure 3.3).

5. Conclusions.

The μ-ETHOS Environment Generator is a fully operational experimental prototype. It is concerned with method-based environments. Any method that can be modelled by a graph (in the graph-theoretic sense)[1] or clichés, or a combination of both, can be easily represented in μ-ETHOS. It covers as many methods as the designer can express through graphs and rules. We have experience building environments for many simple tools (DFDs, ERDs, STDs, Petri nets [DeM78; Che76; Pet81]) and some methods (ASML, Gane and Sarson's Structured Analysis, Jackson's SDM [DeM78; Jac83]). Some other methods can also be handled by the environment generator, such methods include Elmendorff's Cause-Effect Graphs [Elm73] or Nassi-Schneidermann Diagrams [Nas73]. The ease and speed that characterize the use of the environment generator can be illustrated by the fact that the Petri Nets environment was designed and built in a day.

Some features of the Environment Generator are interesting extensions of traditional Software Engineering Workstations. In particular, the symbolic processing capabilities supplied allow the users to execute, albeit primitively, documents generated by those environments. For example, this is a distinguished feature for Petri Nets.

μ-ETHOS provides a consistent user interface from one environment to another. Moreover, it provides a consistent interface throughout all levels of users, so that the basic interface is used by the μ-ETHOS designers, the environment designers and the end-users.

Underlying μ-ETHOS there is a powerful multiparadigmatic set of tools. They are embroidered into a coherent, uniform interface in which each component is easily identified and set in context.

The design team for μ-ETHOS expects its users to interact with μ-ETHOS as a closed environment. However, there is a trapdoor left open that allows the interested user to enter the CommonLISP environment without leaving μ-ETHOS. This makes μ-ETHOS an open environment, if one is willing to program in it.

Upgrades of the current version should begin by migrating to more friendly operational environment.

[1] It should be stressed that the environment is independent of any particular graphical representation of graphs, nevertheless, the canonical representation of many graph-like objects are graph-like diagrams.

6. Acknowledgements.

The authors are thankful to the authorities of the PABI that throughout the period of development consistently helped them in so many ways that makes it hard to mention them all. In meetings organized by that institution in both countries, Viviana Rubinstein, Leo Pini Magalhaes, Maria Laura, Olga, Paulina and Emilio provided transportation, accommodation and, in general, all kind of support to make each occasion profitable.

Professor Pereira de Lucena was a perpetual source of inspiration and enthusiasm. Gabriel Baum helped throughout the work with support in formal aspects. The ESLAI housed the project during part of the work, and its members and Professors should be also thanked. Last, but not least, this paper would not have been possible without the funding provided by PUCRJ in a hectic week in March.

References

[Che76] P. P. S. Chen, "The Entity-Relationship Model - Towards a Unified View of Data", *ACMTrans. on Database Systems*, **1**, **1**, (1976).

[DeM78] T. De Marco, "Structured Analysis and System Specification", Englewood Cliffs, (New Jersey, 1978).

[Elm73] W. R. Elmendorff, "Cause Effect Graphs In Functional Testing", IBM Technical Report TR-00.2487, (Poughkeepsie, 1973).

[Fro86] R. Frost, "Introduction to Knowledge Base Systems", Macmillan, (New York, 1986).

[Leh84] M. Lehman, "A Further Model of Coherent Programming Process", *IEEE Proceedings of Software Process Workshop, UK Feb 1984*, (IEEE C. S.,1984).

[Loo86] *IEEE Software* , Special Issue, January 1986.

[Jac83] M. Jackson, "System Development", Englewood Cliffs, (New Jersey, 1983).

[Luc87] C. J. Pereira de Lucena, "Inteligência Artificial e Engenharia de Software", Zahar Editores, (Brazil, 1987).

[Ste85] L. Nassi, B. Shneidermann, "Flowchart Techniques for Structured Programming" *SIGPLAN Notices, ACM,* August 1973.

[Pen88] M. H. Penedo, W. E. Riddle, "Software Engineering Environment Architectures" *IEEE Trans. on Software Engineering*, **14**, **6**, (1988).

[Pet81] L. J. Peters, "Software Design: Methods and Techniques", Yourdon Press, (New York, 1981).

[Sow84] J. J. Sowa, "Conceptual Structures: Information Processing in Mind and Machine", Addison-Wesley, (Wokingham, 1984).

[Ste85] M. Stejik, D. G. Bobrow, "Object Oriented Programming, Themes and Variations" *Artificial Intelligence Magazine*, Fall 1985.

[TLH89] Takahashi, E. T., Lucena, C. J., Haeberer, A. M., "The ETHOS Project: An Introductory View". *Information Technology for Development* **4**, **1**, Oxford University Press, (Oxford, 1989).

[War84] P. T. Ward, "Systems Development Without Pain", Prentice-Hall-Yourdon-Press, (New York, 1984).

[War84] P. T. Ward, "Structured System Development for Real-Time Systems", Prentice-Hall-Yourdon-Press, (New York, 1985).