Acknowledgement is made to K. Reichert et al for the use of Figure 9 on p. 108, which appears on the front cover of this book.

Advances in Electrical Engineering Software

Proceedings of the First International Conference on Electrical Engineering Analysis and Design, Lowell, Massachusetts, USA, 21-23 August 1990

Editor: P.P. Silvester

Computational Mechanics Publications
Southampton Boston

Co-published with

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo

VI- Topology of the Software Projects

In the traditionnal approach, the skilled programming consists in structuring the software according to logical layers, in order to get application safety, and then achieving a typology on the modules of each layer, in accordance with the technologies used, in order to get a margin of flexibility. Communication between layers is achieved by programming exchange channels. Access to modules is taken into account in the same way by programming a routing module exploiting object description tables. The special cases and exceptions are taken into account only with difficulties and sometimes in a not very lisible way. In a word, all is based on the programmer cleverness.

OO Design, on the contrary leads to systems inclined to be naturally structured in abstraction levels. The project typology is automatically taken into account by the object programming system. The encapsulation mechanism imposes safety, while instanciation, overriding et dynamic ligature support routing. Exceptions and particular cases are taken into account by the mean of special sub-types relative to the common type. Programming becomes simpler, the quality is much less programmer dependent, what is important in large applications.

VII- Conclusion

The OOD technology is the most natural in the industrial area, because it matches with the human concept of the real world. So it is easier to implement the large scale softwares. OOD improves the software production.

OOD represents also a fundamental restructuration, with improvement of the safety, reliability and flexiblity. We have tried to point out this method in the manufacturing field.

References

[1] G. Alrebdawi, J.J. Skubich, Y. Martinez : Supporting Tool For Object Oriented Design Of Real Time Applications, Proc. of the 15th IFAC/IFIP Workshop, Valencia,May 25-27, 1988
[2] G. Bel, G. Bensana, D. Dubois : OPAL : A Multi-Knowledge-Based System For Industrial Job-Shop Scheduling, Intern. J. of Production Research, vol. 26, no.5, May 1988
[3] G. Bel, J.B. Cavaille, D. Leveque : Simulation And Object Oriented Languages For Manufacturing System : Description And Control, EURO XI-TIMS XXVII Joint Intern. Conf. AFCET, Paris July 6-8, 1988
[4] CIM Open System Architecture, Reference Architecture Specification, ESPRIT Project no. 688, May 1989
[5] G. BOOCH : Object-Oriented Development, IEEE Trans. On Soft. Eng. Vol SE-12, No. 2, pp. 211-221, Fe
[6] S. El Baba, A.Troncy, M.T.Martinez : Simulation and Teaching Techniques For Interactive Robot Programming : ROPSE, CG International'88, Geneva, May 24-27, 1988
[7] P. Gopinath, K. Schwan : CHAOS : Why Cannot Have Only An Operating System For Real-Time Applications, ACM Operating Systems Review, Vol. 23, no. 3 Jul. 1989
[8] M. Gravel, W.L. Price : Using the Kanban in a Job Shop Environment, Int. J. Prod. Res. Vol. 26, No.6, 1065-1072, 1988
[9] A. Haurat, M.C. Thomas : LMAC a language of Industrial Robots, 13th Intern. Symp. On Industr. Robots, pp. 1269-1278, Chicago, Apr.1983
[10] T. Lozano-Perez : Robot Programming Proc. of the IEEEE, vol. 7&, No. 7, July 1983
[11] MMS Manufacturing Message Specification ISO Project ISO/DP9506 May 1987
[12] P.L. Primrose, R. Leonard : Predicting Future Developments In Flexible Manufacturing Technology, Int. J. Prod. Res. Vol. 26, No.6, 1065-1072, 1988
[13] H. Tokuda, C.W. Mercer : ARTS : A Distributed Real-Time Kernel, ACM Operating Systems Review, Vol. 23, no. 3 Jul. 1989
[14] A. Troncy, M.T. Martinez, S. El Baba, C. Hugues : Modular Robots - Graphical Interactive Programming, 1988 Int. Conf. On Robotics and Autom. Apr 24-28, 1988
[15] M. Stefik, D. Bobrow : Object Oriented Programming : Themes and Variations. The AI Magazine, Winter 1985

Object-Orientated Programming: An Original Application to Substations Design

J-L. Lilien, M. Pallage
Université de Liège, Institut d'Electricité
Montefiore B28, Sart-Tilman, 4000 Liège, Belgium

INTRODUCTION

The whole layout design of an open-air high-voltage (from 72.5 kV to 765 kV) substation has always been a real challenge to systems designers, so true is it that it requires them to be acquainted with numerous fields such as electrical and mechanical engineering.

Adding the strongly parametrizable and country-dependant type of expert appraisement to the fact that very few people can control the entire design strengthens the difficulty in dealing with it.

In connection with industrial partners, we felt the need to build a software whose aim would be to mix this world-spread expertise together with very up-to-date computation codes, database systems and artificial intelligence techniques.

This ambitious work began in September 1988 with the collaboration of an electric devices manufacturer and a network operating staff, namely Merlin Gérin (France) and EDP (Portugal). We present here SAPHO. the preliminary version of our system (SAPHO stands for Système d'Aide à la conception de Postes à Haute tension Ouverts).

OPEN-AIR SUBSTATIONS DESIGN

The complexity of the whole substation design is of common knowledge for the following reasons:

- system designers have to deal with numerous fields going from electrical engineering to mechanics, some of those fields requiring them to become acquainted with very up-to-date research results
- experts for the entire design are very seldom
- like any other design, the high-voltage open-air substation design is not a sequential work and you may have to re-begin a one week engineering office work when noticing that for example the planned structure won't support the electrodynamic stresses due to short-circuits

It appeared while collecting the expertise that many of the values to consider, such as security distances, many computing methods and the types of electric devices to be placed in the bays often differ from one country or company to another. In fact, you can give the same specifications for a substation to different experts and will rarely obtain two similar results. This lack of uniformity is mostly due to what we call "historical use", and clearly constitutes a big problem when willing to computerize the design. Historical use is: *as long as we do not run into troubles because of the methods, values or materials we use, why should we change our habits and adapt ourselves to up-to-date computation methods or research results?*. This establishment motivated us to introduce adaptability or ease of modification in our system.

The whole design is commonly accepted as being a well identified sequence of seven huge tasks to fulfill. the way to achieve those tasks being subject to changes, as said above. Possible backtracks from a task to a previous one may happen when the results of that task are unsatisfactory. Those seven tasks are the following, given in the correct sequence:

1. Single-phase diagram choice among 15 possibilities
2. Electric devices (circuit breaker, surge arrester, current and potential transformer, isolator, wave trap, anchoring insulators and insulator supports) choice
3. Busbar type (rigid or flexible) choice. General busbar and switch-bays (feeder, transformer, coupling) disposition including overall space requirements (clearances, security, span length) computation
4. Busbar sizing (diameter, cross-section and thickness or number of subconductors) in order for it to support the rated and short-circuit currents and to lessen the corona effect. This task also includes the sag verification, the frequencies analysis and the aeolian vibrations study
5. Computation of both static and dynamic overloads like wind, ice, short-circuit and combinations of them. Final acceptation of the overall design with devices choice updating if needed and definition of the load to be supported by the anchoring structures
6. Computation of the ground network for step and touch potentials
7. Lightning protection

THE CONSTRAINTS

1. As the target user audience is mainly composed of engineers and students, supposed not to be acquainted with computer science, a first constraint is obviously a **convivial man-machine interface**.

2. If the sequence of tasks to be performed is commonly accepted, the way to achieve those tasks objectives often differ from one country or company to another. Furthermore, the expert appraisement is quite parametrizable, thus providing the following constraint: **providing an easy way to access and modify both expertise and parameters.**
3. The last constraint is to provide the user **explanations** about what is going on.

Our main will is to come to a system where **everything** will be available to the user in a very simple and convivial way. This means that all **formulas, rules, parameters** and **pieces of expertise** should be **accessed** and possibly **modified** by the user (entitled to it) willing to **fashion** the system his way. As will be presented later, this constraint motivates the adoption of an original control structure.

BUILDING SAPHO

We will in this section present the preliminary version of the system. For reasons explained later we chose to build our system using an object-oriented architecture and of course an object-oriented language. You will thus first find a very brief approach of what is object-oriented programming. Further details will be found in [1] and [2].

Object-oriented programming

Object-oriented programming is a programming style based on the **encapsulation** of both data (the information to handle) and procedure (the way to handle this information) concepts. It is in opposition with the "classical" programming style which maintains a clear gap between data and procedures. The task of program writing thus consists of the definition of a world of independant **objects**, communicating with one another through **messages**, each object being made of a certain quantity of information and procedures to process that information. For instance if we want to compute the permissible current rating of a given busbar:

- in "classical" programming, we will write a procedure computing this value for any busbar, procedure that will receive the given busbar characteristics as arguments
- in object-oriented programming, we will create a busbar object with the characteristics of the given one and ask that object to compute its permissible current rating and to return the result

Objects are the unique type of entities that can be handled by an **object-oriented system**. It means that **everything**, an integer as well as a modeled disconnector, has to be represented as an object. An object is made up of three parts:

consisting of a collection of **attributes** whose

can distinguish between two types of objects: **complex objects** whose attributes are means of referencing other objects and **primitive objects** that do not have attributes but only a value which is the object itself (an integer for instance)

- it has a collection of **methods** that capture its behaviour. The methods of an object are the only procedures able to manipulate the object private memory and to return its state

- objects communicate with one another by sending messages requiring the receiver object to execute one of its own methods. An object can thus be considered as an independant entity, except for a collection of messages, coming from other objects or itself, it is supposed able to interprete. This collection of messages is called the **communication interface**. Notice that the structure of a message is the following:

```
[receiver  method  arguments]
```

Each object is defined as being part of a **class**. A class describes the structure of a collection of objects having the same attributes and methods, those objects, called **instances**, only differing by the value associated with their attributes. When a message is sent to an instance, the method implementing the response to it is found in the class definition.

Each created class is considered to be subclass of an already existing one called its **superclass** and **inherits** the methods and attributes of its superclasses. There is a special class, superclass of all others, serving as root for this hierarchical structure. A new class can bring new attributes and methods adding them to the inherited ones.

To end this quite abstract section, we illustrate that theory with an example. Suppose you want to modelize busbars. You first have to take into account the fact that there are two kinds of busbars, rigid and flexible ones with specific characteristics for each. There are thus two specializations of the busbar concept, in fact two subclasses of the *busbars* class. Thanks to the inheritance mechanism, you will have to introduce in the *rigid-busbars* and *flexible-busbars* subclasses, only those characteristics strictly specific to them. Among others, the *rigid-busbars* subclass will contain a *wall-thickness* attribute and a method to compute the permissible current rating, while the *flexible-busbars* subclass will have a *number-of-sub-conductors* attribute and, since the computation is different from the one for rigid busbars, a method to compute the permissible current rating (it may be given the same name as the other one). The *busbars* class will be structured with all attributes and methods common to both types of busbars, such as *chosen-material*, *cross-section* and *external-diameter*.

SAPHO, a preliminary version for our system

build an expert system in the computer science meaning of the word. In fact we were facing a very important algorithm making use of lots of techniques, with a lot of unavoidable tasks to perform, the way to achieve them only being subject to changes. Furthermore, this algorithm was quite sequential although there may occur a backtrack to a previous stage of the design from time to time. SAPHO was nevertheless called "expert system" because it was based on expertise and even if it didn't make use of decision trees or inferences, its aim was still to try behaving like an expert in the field.

We use the object-oriented language Spoke as software support for developing SAPHO. The reasons for this choice are:

- object-oriented programming is very well suited to model the many complex physical entities that are involved in the design, such as breakers for instance

- those languages provide easy ways to design convivial man-machine interfaces because of the window toolkit they include and the object concept itself which makes that a window is treated as a single entity only responding to a given list of activations

- the constraint of letting the user access and modify the expertise, in fact the formulas, rules or parameters used, can be satisfied by using object-oriented programming. In a system where everything should be adaptable it is interesting to consider independant entities such as named formula objects that can be modified without affecting the rest of the system

The hardware support for our work is a SUN 3/60 workstation with 12 Mbytes RAM.

SAPHO is composed of four modules which we will next briefly develop.

The knowledge base This module is to contain knowledge about the physical entities involved in the design procedure. All informations about entities such as busbars or string insulators are there to be found. In fact, this knowledge base consists of numerous SPOKE class definitions, each modelling a real-world component, as presented in the example above.

Fortran codes interface As said above, the design requires numerous complex computation methods, especially methods using finite elements. We decided to integrate into our system a software called SAMCEF-CABLE developped at the University of Liège. By integration, we mean that we avoided the user the troubles of using such a complex code by automatically generating the input data files (with automatic predetermination of most critical short-circuit conditions), managing the result files and by allowing visualization of those results. The integration was obviously made easier by a very good knowledge of SAMCEF-CABLE.

A lot of small Fortran functions were also added to the system, their integration being facilitated by the Sun Common Lisp Fortran interface.

Electric devices database This module is to permit storage of electric devices catalogs in order for the system to select among them the best fit device, that is the one satisfying a number of criteria.

After having eliminated the commercial database softwares, the relational ones because the objects to store were everything but flat tuples and the object-oriented ones because none of the existing few ones was compatible with either LISP or SPOKE, we decided to proceed the following way. For each device type, a file was created, containing the many instances of the class corresponding to that device. That is we managed files of SPOKE objects. Selections were made by loading the concerned file into the SPOKE environment, selecting the best device and then killing all non-selected objects. Although this is a naive approach it provides good results and seems to be sufficient for our preliminary version whose aim is among others to show the feasibility.

A fully adaptable control structure The identified constraints egged us to build a fully adaptable control structure which we will now introduce.

We are confronted with the problem of implementing huge conception tasks, each of them requiring different techniques and expertises. As said above, the corresponding algorithm will be a task chaining one with possible backtracks to previous stages.

What we first thought was to implement each huge task separately using classes including all methods needed to achieve a task. The sequencing of operations inside a task being scheduled by a *leader* method. This unelegant type of implementation surely would have worked, even if making roar object-oriented programming purists, and was motivated by the following remark: Object-oriented programming authors all tell you to model the behaviour of each real world entity in order to be as close as possible to reality, but what if you have to deal with numerous methods such as *select among breakers the one satisfying some criteria*? Do you attach this method to the *breakers class* knowing that it won't ever model one of its behaviours or do you create a *task object* whose only behaviour will be to perform the selection?

We then adopted an architecture, based on the following ideas:

- why not consider *task objects* corresponding to elementary operations instead of a large number of complex operations? We would then have to consider independant entities which we will call actors, each of them having the responsibility of an elementary task such as initialization, computation, help providing or results displaying

direct reference to the next actor it has to activate

- there will be a class for each type of actor, and each particular actor of that type will be one of its instances. For example, we can have a class for all actors whose operation is to let the user enter a value for a system variable (not such an easy stuff as it seems, because, this type of actor will have to generate a menu with all possible values, allow the user to ask for help about the variable and allow him to enter his value in a convivial way whenever he is not satisfied with the proposed ones) and instances of that class for each variable to be given a value

- the control structure will then be made of a static network of actors of which the leftmost one is to be activated to start running the system

- one such approach satisfies one of our constraints which is **explanations generating**. Indeed, as each actor is devoted to a type of operation, it is easy to generate rudimentary but sufficient explanations at the activation of any actor and after the result of its operation is obtained. To manage this we can have a standard *trace* for each type of operation and adapt this trace, basing ourselves on the context of the activated actor. For example, if we have an actor devoted to the initialization of the highest-voltage, the (silly in that case) trace will take into account the variable it has to initialize, that is the highest-voltage

To illustrate this, we can consider the simplest type of those actors, the initialization ones. The following pieces of code briefly describe the very simple way of managing this kind of actors and all actors in general. First of all, we have to define a root class *init_actors* for all those actors:

```
[init_actors isa class superset system_actors
   with
     (slot the_variable_to_init
              range system_variables)
     (function actor_operation ...)
           ...]
```

An actor, instance of this class will have the following main characteristics: it will be assigned a reference to the variable he is to initialize and he owns a method allowing it to perform its operation.

We do actually consider two sub-types of those actors: the first one, the simplest, will be for actors clearly knowing the next actor they have to activate and the second one for actors in which a test will be performed on the value assigned to the variable in order to determine one of two actors to activate next. Here is the code for the first sub-type:

```
[next_known_init_actors isa class
          superset init_actors
   with
```

```
    (function display_trace ...)
    (function display_trace_when_over ...)
    (function activate_next_actor ...)
    (function activate ...)]
```

All those characteristics are self explanatory, but let us precise that the *activate* function is the only function through which any actor can be accessed. It is responsible for the sequencing of actions inside that actor.

The second sub-type is defined by:

```
[next_unknown_init_actors isa class
        superset init_actors
  with
    (slot it_is_ok range boolean)
    (slot next_actor_if_ok ...)
    (slot next_actor_if_not_ok ...)
    (slot the_value_to_compare_to ...)
    (slot superior_to range boolean)
          ...
    (function display_trace ...)
    (function display_trace_when_over ...)
    (function activate_next_actor ...)
    (function activate ...)]
```

The attribute *it_is_ok* is to reflect the result of the test to perform, the *the_value_to_compare_to* slot is for the value that will be compared to the entered value for the variable to initialize and the slots *superior_to*, *inferior_to*, etc. are to tell what kind of comparison operator will be applied. One may wonder why the other functions for that class look similar to the ones of the above class: in fact their names are the same (this is a feature of object-oriented languages) but the content differs (for instance, the trace to generate clearly differs from one case to another).

This structure cannot convivially satisfy the remaining constraint that is **access and modification of expertise** since the user would have to deal with generating new actors and modifying bindings. Thus we decided to consider three abstraction levels:

1. **the basic level**, which is made of the already introduced actors network
2. **the middle level**, which is made of a collection of instructions to generate actors and establish bindings between them, and which generates the *basic level* by simply executing those instructions
3. **the electric strategy level**, which allows the user to read and possibly modify the expertise by adding, suppressing, interchanging or modifying high level instructions, expressed in a very simple formalism. This level

Our main will was to allow **everything** in our system to be accessed by the user. All system entities such as formulas will thus be available and the user will be able to define his own entities in order to fashion the system his way.

How will he do that? Those entities will be stored as named objects and those names will be used to compose the high-level instructions. The user defining his own entities will give them a name. Suppose for instance that we have a high-level instruction to compute the busbar external diameter to support the rated current using a formula entity named *formula_1*. If the user whishes to use another formula, he will only have to enter it in the system, in a way described later, to give it a name and to replace *formula_1* by that name in the instruction.

All entities and expertise (the high level instructions) will be editable in a suited editor, allowing the user to modify them or to compose his own ones by simply selecting on menus everything he needs. If we consider formulas, those menus will present mathematical operators and all system variables (their name). A menu for all system entities (their name) will always be user available in order for him to access these entities or to select their name for insertion into a high-level instruction. The user willing to compose a new high level instruction will be provided a menu with the simple syntax of those instructions and the menus for all system entities.

The system entities are:

- **formulas**: a formula will be written in a lisp-like formalism for the simplest of them. For complex computations, FORTRAN codes will be used as formulas, assuming the user is familiar to this language
- **parameters**: a collection of parameters will be available for value modification, that is the user will be able to give a value corresponding to his country to parameters such as *the percentage of the span to consider in order to proceed to the busbar sag analysis*. He will also be allowed to add new parameters which he will insert in his formulas
- **system variables**: in order to be used by high-level instructions and formulas, the system variables such as the highest voltage will be available by menu. Each variable is assigned a collection or interval of possible values and the user will be abilited to modify them
- **selection criteria**: the attributes of all classes of the knowledge base will always be available to compose the selection high-level instructions
- **rules**: a collection of rules will also be user available and definable
- **checkings**: a checking is a collection of tests to validate the value assigned to a system variable with regard to already initialized ones. They are to ensure a consistency of the system. For instance, a checking based on the highest voltage can be done when initializing the rated current

It is now time to introduce the high level instructions. There will only be three types of them because our experience with the expertise told us it would be sufficient (for all tasks but the *disposition* one that will need the integration of a CAD tool in the system and for which other instructions will be introduced). Each of those instructions will correspond to at least one actor, selection operations for instance requiring many of them. We will only give one example of the most complex instructions for each type:

1. **initializations** : For initialization of the planified highest voltage and basic insulation level, we use the following instruction:

```
[sapho init_and_test highest_voltage using (< 300)
  (if_ok [init lightning_impulse])
  (if_not_ok [init switching_impulse])]
```

   That particular instruction means that once a value has been given to the highest voltage, a test on this value is made in order to determine the next instruction to execute. That is, if the highest voltage is less than 300 kV then proceed to the lightning impulse initialization, else give a value to the switching impulse. As can be seen on figure 3, this will generate a three actors sub-network. Notice that many instructions can be inserted in the if_ok and if_not_ok branches.

2. **computations** :

```
[sapho test_then_compute external_diameter
  using (= busbar_type rigid)
    (if_ok with formula_1)
    (if_not_ok with formula_2)]
```

   In this case, the formula to be applied in order to compute the busbar external diameter depends on the busbar type.

3. **selections** :

```
[sapho select_single disconnector
  using criteria
  ((= disc_type chosen_disc_type 1)
  (= manufacturer (siemens merlin_gerin) 1)
  (>= disc_highest_voltage highest_voltage
          1 25%)
  (>= disc_bil bil 1)
  (>= disc_rated_current rated_current 0.5)
  (>= disc_sh_circuit_current
                  sh_circuit_current 0.5))]
```


device for each of the mentioned manufacturers, device that will be of the chosen type and that will satisfy all specified criteria. It is clear that a solution will always have to be provided, so, if a criterion cannot be satisfied by the stored devices, the system will have to proceed to an approximation on that criterion, that is, if there are devices satisfying all criteria but the one involving bil, the selection will return among those devices, the one whose disc_bil attribute is the closest to the system bil. In that rare case, the system will go on working with that approximation if told to, telling you anyway to contact a manufacturer to build a special device for your needs. If you prefer waiting for the dimensions of that special device, the session will be stopped until you enter the new device in the system.

The numbers at the end of each criterion express a relative importance among those criteria. This will be useful to determine a single device. In effect, suppose we have a collection of possible devices, if we wish to select the most adequate one, we have to sort those devices on the criteria, in order to get an ordering of devices. If we consider the example, it can easily be seen that the device satisfying all equality criteria and whose attributes involved in inequality criteria are the smallest ones to be found in the ordering, will be selected. What if we are to decide between two devices having the following characteristics:

- A1: supported highest voltage 245, supported bil 1050, supported rated current 1600 and supported short-circuit current 40
- A2: supported highest voltage 300, supported bil 1175, supported rated current 1200 and supported short-circuit current 31

It must be possible to tell the system to begin sorting for instance on the highest voltage, then on the bil and so on. An hierarchy for criteria is thus to be established for the final sorting.

Some criteria are to be considered carefully. If we consider for example the highest voltage, it is not acceptable to select devices supporting 765kV if the planified highest voltage is 420kV even if obliged to go that up because of other criteria to satisfy. That is the purpose of the percentage found at the end of those criteria. In our example it tells the system not to go beyond 25% of the planified highest voltage.

We hope that the syntax of those instructions will not be too discouraging so that the control structure will effectly be adaptable. The instructions are of so high level of abstraction that we believe the entire expertise can be expressed using few of them thus insuring an easy reading and understanding of it.

CONCLUSIONS

We are currently developing a fully adaptable expert system called SAPHO. For this system to be good, we are confronted with a substation design combining high levels of both security and reliability at the lowest price. These requirements necessitate a good level of expertise and some complex computation methods in order to be as close as possible to reality thus insuring reliability and money saving (no oversizing).

Those complex methods are of difficult use and often discourage the user because of teduous manipulation. SAPHO fully integrates those methods and the expertise needed for their manipulation. It also provides a very convivial man-machine interface and a high level of expertise easily adaptable by a user who is not the system conceiver.

We hope SAPHO will satisfy both students, in a didactic point of view and the industrial world, by the time saving it brings and the facts that it synthesizes the procedure to be followed and that it helps not to forget any detail.

ACKNOWLEDGEMENTS

We would like to thank Mr. J-C Leroy from Merlin Gérin and Mr. F. Mira from EDP for the very valuable help in the expertise collection and synthesis. We would also like to emphasize the initiating procedure of this project made by Mr. M. China from Merlin-Gérin and the help of his A.I. team. Finally, it is worth pointing out the stimulating contacts with the Montefiore A.I. team, especially with Mr. P. Yans.

TRADEMARKS

SPOKE is a trademark of the Laboratoires de Marcoussis and is distributed by Alcatel ISR. Sun and Sun Common Lisp are trademarks of Sun Microsystems.

BIBLIOGRAPHY

[1] Goldberg A., Robson D., *Smalltalk-80: The language and its implementation*, Addison-Wesley, 1983

[2] Stefik M., Bobrow D.G., *Object-oriented programming: Themes and variations*, The AI magazine, January 1986

Translators: Towards Open Ended Finite Element Software and a Common Standard

S.R.H. Hoole, S. Ellsworth

Department of Engineering, Harvey Mudd College, Claremont, CA 91711, USA

Abstract

Many finite element programs are now available for the solution of electromagnetic field problems. They use different formulations and even when they rely on the same formulation, they employ different data formats/structures for their input and output. This would be fine so long as one always uses the same program.

Different programs have different strengths and the practising engineer usually likes to switch from one program to another depending on what he or she wishes to do. For example, in three dimensional field analysis, it is often desirable, in view of the paucity of good preprocessors, to use solid modelers or mesh generators for mechanical problems to define the geometry and then couple its output to a finite element program.

The realization of this ideal requires a common, neutral data standard for data exchange so that all programs are interfaceable with other programs. In this paper, the authors propose a neutral data standard based on minimal hierarchical standards for describing finite element data and demonstrate translations between packages and the common standard. Specifically, translation from the graphics PICT format to a form suitable for finite element analysis is demonstrated.

Further, the experience in developing such neutral tools for interfacing finite element programs with drafting packages on the IBM PC and the Macintosh machines is described and the paper draws upon the experience of the electronics industry in enforcing the common standard known as EDIF.

Introduction

Under many circumstances, the ease of using finite element analysis [1] depends on the strengths and weaknesses of the package one uses to create, solve, and analyze problems. For example, in our own work, we were hampered by the lack of convenient scaling and rotation in the mesh definition stage. These are tasks which are built in to most CAD packages. To cover this weakness, we added the capability to import pictures drawn in one specific format (Macintosh PICT) into one of our implementations. This is obviously not the preferred solution; while Macintosh CAD programs can generate this format, IBM CAD programs generate other formats, and VMS programs generate still another. Also, once we import the pictures into our program, reading a new mesh in from the CAD programs would involve the loss of data.

Any given vendor of analysis software is likely to be unwilling to adopt a standard at first, as they would prefer that their customers continue to use their package in preference to someone else's. On closer examination, however, this reasoning fails. Having the capability to export data to other programs will enhance the capabilities of all packages [2,3]. At the present time, a package must be chosen based on its overall