J. Michael Gerzso
Independent Researcher
104164.341@compuserve.com

Speculations on a Machine-Understandable CAD Language for Architecture

One of the objectives of research in computer-aided design in architecture has been to make computer tools or instruments for architectural design, not just drafting. There has been work presented at ACADIA and other conferences related to artificial intelligence, data bases, shape grammars, among others. In all of these cases, existence of a computer language in one form or another is implied.

The purpose of this paper is to argue that the progress in the development of intelligent design systems (IDS) is closely linked to the progress of the languages used to implement such systems. In order to make the argument, we will adopt an approach of first specifying the characteristics of an IDS in terms of a conceptual framework of computer languages in a CAD system in general, and what it means to develop a machine-understandable language for architectural CAD in particular. The framework is useful for classifying research projects and for structuring a research agenda in architectural CAD.

Un cadre de langage CAO: spéculations au sujet d'un langage compréhensible-par-machines pour l'Architecture

Un des objectifs des recherches en CAO architectural a été de produire des outils informatiques pour aider la conception, et non seulement pour le dessin. Des travaux de ce genre ont été présentés à ACADIA et d'autres conférences traitant de l'intelligence artificielle, de banques de données, et de grammaires de formes, entre autres. Dans chaque cas on sous entend l'existence d'un langage pour ordinateur, sous une forme ou autre.

Le but de ce travail est de démontrer que le progrès au niveau du développement de systèmes de design intelligents ('Intelligent Design Systems', ou IDS) est directement lié aux progrès au niveau des langages utilisés lors de l'implantation de ces systèmes. À l'appui de notre argumentation, nous spécifions d'abord les caractéristiques d'un IDS en termes d'un cadre conceptuel de langages pour ordinateur dans un système DAO en général, et ce que ça signifie de développer un langage compréhensible par machines pour le DAO architectural en particulier. Ce cadre est utile pour la classification des projets de recherche, et pour l'élaboration d'un agenda de recherche en DAO architectural.

introduction

For CAD systems to be more useful in the process of architectural design than they are now, it has been argued that it is necessary to augment their capability of representing and manipulating information about the design process and built form. Up to now, CAD systems have been very effective in automating drafting activities and have changed the way in which architects and engineers carry out their work. It is safe to say that even though this is an impressive achievement, there is still a big difference between drafting and designing, as well as the strategies for developing software for each.

Therefore, the question becomes: assuming that an intelligent design system (IDS) is the object of research in order to improve the quality of architectural design, what will the strategy for developing such as system look like? How do we know if we are making progress?

The purpose of this paper is to argue that progess in IDS is closely linked to the progress of the languages used to implement such systems. Historically, when there was a great advances in languages, there were great advances in IDS's. However, in the last 10 to 15 years, there have been no great advance in languages to write intelligent systems in comparison to the advance achieved between assemblers (non higher level languages) and the first high level languages, such as Lisp. This is in spite of a major transformation since 1988 of functional languages, such as Lisp, C, and the like, to object oriented languages, such as SMALLTALK, C/C++, ADA, and Java. What has happened is that computer systems, including CAD products, have become increasingly larger and complex, but not smarter.

In order to make the argument, we will adopt an approach of first specifying what an IDS should be. For each aspect of the IDS, we will identify the type of language needed for its implementation. The classification of the languages will be done according to the CAD Language Framework (CLF). This is done by means of a quick survey of the history of computer languages. By comparing what the languages should be like in the CLF and the ones that exist today, we can have some idea where CAD systems stand today, and how difficult it will be to close the gap between what exists and what is desired.

The thesis that the advance of IDS systems are directly linked to the development of languages is the result of the experience gained from designing the TM language and three of its compilers. For those who are aware of the history of computer languages, it comes as no surprise that language development can require an effort of about 10 years in order to understand the grammar of the language as well as its compiler implementation.

the TM language project

The original motivations for developing TM were to provide a language that would facilitate the implementation of "intelligent" architectural design systems as specified in Gerzso (1979). A more complete description of the basic technical characteristics of TM and its relation to data bases can be found in Gerzso (1985). The motivations were to provide a language that would:

1. have extensibility which would permit a programmer to define the required objects which include architectural, graphic, logical, design restrictions as well as any other entities. That is, eliminate the dependency on a predefined and restricted set of data types built into the language.

2. be interactive and be a unified language interface with an entire standard computer system.

In the beginning, TM began as an alternative to SMALLTALK-80 at a time when this language was not available. But as TM evolved during the last five years, several features different from those of SMALLTALK-80 became apparent and worth pursuing.

The first feature had to do with the definition of the objects. In contrast to SMALLTALK-80, TM has as a basic construct called an administrator, which itself is an object and which is responsible for data or pasive objects. Thus, the code is not wrapped up or "encapsulated" with the data object itself, but separate and only associated with it. The association of code to data objects is the way in which encapsulation is done in this language. The concept of the administrator eliminates the need for metaclasses and considers code as a separate object and member of its own class. To those already familiar with object based systems, the idea of an administrator may appear strange, but in reality, it is merely a variation of the traditional idea of code separated from data.

Closely related to the way objects are conceived is the incorporation of data (object) typing (classing) in TM. However, in contrast to PASCAL, typing is strongly recommended but not required. TM allows that objects be classed as "anything", but the burden is on the programmer to make sure that no undesirable effects occur during execution. Typing provides the compiler with information concerning the existence of an administrator which is required for carrying out a response to a message. If the administrator does not exist or if the message is incorrect or non existent, the compiler will warn the programmer. The idea is that many bugs should be detected at compile time, thus reducing the debugging at run time.

The next feature which was greatly influenced by ADA is the explicit separation between public or outside perception of a function and its implementation. Thus the design of TM requires that the compiler support the specification of the definition of the messages that an object (administrator) can receive and answer. The specification can be coded and compiled independently of the implementation of the responses to the messages. Upon the compilation of the response or method, the compiler verifies if the message protocol is satisfied and if all of the message expressions in the method satisfy the specifications previously compiled. In contrast to traditional compilers, TM maintains files where all of the specifications of the objects and linking information required at the time that an administrator is loaded. Other than that, the message passing mechanisms are almost identical to those in SMALLTALK-80.

Upon studying the implementations of SMALLTALK-80 (Krasner 1983), it was concluded that the idea of searching for methods in hierachical inheritance structure during run time is too costly despite its possible programming convenience. In contrast, TM does the method or message response searching during compile time if it is possible to determine the receptor object of the message being sent. It is felt that because of the data typing (classing) feature of the language, the compiler has useful information as to the class of the receptor in many situations. The advantage of such information is to increase the execution of the compiled code. In addition, compile time method searching also makes feasible non hierachical inheritance. This is also closely related to the idea of the compiler checking the message specifications, which has already been mentioned.

All of these features have been incorporated in the various designs and implementations of TM. The design is now in its fifth version (Gerzso 1987). The compiler is now being reimplemented for the third time (Calderon 1992). The virtual machine with its corresponding memory management was completed and running on a VAX-730. It is the second version (Cardenas 1986). A version of the compiler and virtual machine was installed on a Sun workstation and later PC's under DOS (Calderon 1992) and Windows NT.

an intelligent design system for architecture

If we are to attempt to come to grips with what we mean by a CLF, it is necessary to understand first how such a language framework fits in an overall IDS system. In order to achieve this, it is useful to imagine what the functional characteristics of such a system. And since such a system does not exist yet, it is necessary to rely on what we know about present day technology, even though it is evident that such will be surpassed in some aspects in the future.

A CLF by itself does not provide us with an intelligent design system. In the best of cases, it provides us with a means of communicating design ideas to a machine. Therefore, the first general function of a language is a human interface. But once this communication has taken place, the machine must be able to do something with the ideas. This gives the second general function of a language, which is the implementation of the design system. Thus, the first function, we need an INTERFACE language; for the second function, we need an IMPLEMENTATION language. It is not obvious if the two languages should be the same.

The imaginary intelligent design system is is divided into two versions: a full blown version with all of the desired features or capabilities, and a basic version which has only the bare essentials. Both of these versions presuppose technologies which have only been partially developed, if at all. Other technologies have been commercialized for several years. The structure of the system is organized "from the outside in". That is, those subsystems that the designer comes into contact with appear first (human interface programs) and the support systems appear later. For each subsystem, we identify the language that is usually used for either implementation or for the human interface. Later, we attempt to see how these languages are related to the CLF. The list of the following subsystems for the full blown system are:

- sketch and drawing recognition (graphics language, picture grammars)
- voice recognition and speech synthesis
- interactive graphics (3D with everything) (graphical language)
- natural language
- implementation programming languages (Lisp, C, C++, Pascal, Java, TM and others)
- architectural design procedures and rules (production rules)
- building form knowledge representation
- general knowledge representation
- theorem proving (logical language, Prolog)
- relational and / or object oriented data base (query language: SQL)
- operating system (command language or GUI)
- highly parallel non-von Neumann architecture (processor hardware). (machine language)

A sketch and / or drawing recognition module would permit an architect to communicate with a machine graphically, in the same "language" that the architect uses in his daily work (Do 1995; Herot 1976; Negroponte 1973). In order to avoid typing in information or pointing to a menu on a tablet or with a mouse, a voice recognition system would permit the designer to have a dialog with the machine at the same time he or she is sketching or drawing. The interactive graphic system would provide all of the graphic capabilities necesary for digitizing, storing, retrieving, displaying sketches or drawings in 2 or 3D with all of the capabilities that exist on the market today (hidden surface, color, shading, transparency, reflection, dynamic rotation etc).

Natural language module would take care of parsing a sentence, establishing its meaning and translating it into some internal representation or language. The internal representation may reside in a general knowledge representation system or even in a knowledge data base. It would also perform the inverse process by taking the output of some process and generating the sentence. It frees the architect from having to learn a specialized computer or command language.

However, if the natural language module were to be bypassed, then it is possible to communicate with the system through an application specific computer. Such a language may be subset of the CLF.

Up to this point, all of the modules are for graphic or language communication with designer. The rest of the system is what carries out the "intelligent" processing that is specific to architecture.

The first one of these subsystems are the data structures and programs which contain the design rules and methods. It is an embodiment of an architectural design expert, and it possesses the knowledge required for designing. It is not only a record or data base of previous solutions or cases, but also has the capability of storing what a designer specifies of what ought to be done, that is, an issue-based system (Knapp 1996). It should presuppose the existence of a theory of built form generation which is computable.

Closely related to the above system is another system which can represent knowledge about built form. It has data structures or procedures which describe aspects of spaces, doors, windows, etc. and all of the necessary descriptions of the attributes of these objects (Khemlani 1997).

The design rules and procedures along with the knowledge representation systems are built on top of a general purpose knowledge representation system. It may be necessary to have such a module because knowledge for other systems such as sketch, drawing and natural language recognition and synthesis need to have general attributes about the world which can also be stored and available.

Much like the need for general knowledge representation is the need for a logic engine. Such a capability supports the human language interfaces but also systems that are directly related to architecture. A common problem in this area is to decide if a given design alternative that is being generated satisfies a series of restrictions on the use, position, dimension and geometry of spaces. In many cases, these restrictions can best be represented in some logical form and used in conjunction with a theorem prover. Theorem proving is also related to the deontic aspects of a project. That is, statements or rules on what "ought to be", how these rules are processed. Therefore, a logic engine for our design system probably is one that is different from the type of system of PROLOG.

The rest of the system is comprised of modules that are system software that support all of the above. They are required in order to keep track of information in memory or in secondary storage (data base) and to administrate the hardware (operating system). Finally, there is the hardware itself. Conceptually, there is nothing new about the idea of a data base, operating system and hardware The latter has yet to arrive at a level of standardization within the industry but is evolving very rapidly.

In case we want to reduce the scope of the system to a more basic one, what subsystems would be essential if the system were to be considered "intelligent"?

- interactive graphics (3D with rendering and texturizing)
- programming languages
- architectural design procedures and rules
- building form knowledge representation
- theorem proving
- object oriented or relational data base
- operating system
- von Neumann processor (computer such as a PowerPC or Pentium).

There is no doubt that the organization of a design system as described above may be viewed as incomplete, too general, or even erroneously structured. This is subject to debate and further speculation, but the general outline of the system is valid because it is based on experience in design methods, computer design and artificial intelligence. To those familiar with AI, this system looks very similar to a 5th generation machine (Feigenbaum 1983). It is very ambitious, and it not clear if can be implemented in the next 10 to 20 years.

CAD language frameworks

Whether a full blown or reduced system is contemplated, the important thing to remember is the fact that, in several places in the intelligent design system, many subsystems have some relation to languages. Therefore, in attempting to pin down what is meant by a CLF, it is appropriate to take each one of these language subsystems and clarify their characteristics with respect to the overall system. We will see that each one is different, but at the same time, we can speculate that they may constitute a definition of CLF by taking them together. To do this, we classify each language system according to the following categories:

"H" human interface languages;
"A" application, knowledge and logic modules, implementation languages;
"S" system implementation or operator interface languages;

"H" category languages are those which are similar to natural languages. They must be able to deal with some ambiguity like natural languages, but they must also have the capability of being precise like implementation languages. Technologically, this kind of language has yet to be developed. Advances have been made in natural language recognition, but they still fall short of what is required here.

"A" category languages encompasses the application specific modules and their corresponding implementation languages of which many well known third or fourth-generation languages exemplify. “S” category languages correspond to those that are for implementing systems software or function as command languages in operating systems.

human interface languages

The first modules that contains an “H” type language are those related to pictures or architectural sketches and drawings. These appear in two forms: the first one is a metaphor which views architectural drawings as a graphical language as proposed by David (1972). The idea is that architects don’t speak in a natural language (words, sentences and so forth) but primarily via diagrams or drawings. Another well known example of this approach is the pattern language work by Alexander (1977). Such approach has never had a very marked effect on the architectural profession or on the implementation of computer systems. But in design methods, it has been an important influence on a way of thinking about design. In contrast, one approach which has been more pragmatic and related to architectural practice and which has some influence from the pattern language work is the SAR design methods by Habraken and his group (Habraken 1976). However, none of these approaches has developed formalizations on an abstract level as is usual in computer science.

The second form in which the idea of language in design appears is that drawings they can be represented in terms of a grammar which generates sentences that are mapped onto drawings as done by Shaw (1970). In this case, the purpose of the picture grammars has been for pattern recognition or vision in robotics. Because of fact that this is so, it can be argued that it gives some intelligence to the machine in that it must establish what the machine is seeing. But despite the emphasis on pictures, the researchers, such as K. S. Fu and others, in syntactic pattern recognition have never been concerned with architectural design systems (Fu 1974; Gonzalez 1978). It is only those such as the present author, Mitchell, and Stiny in design methods who have been aware of the possible relation (Gerzso 1979; Mitchell 1979, 1990; Stiny 1979).

However, there are other approaches like the one developed by Guzmán which can characterized as the non-syntactic approach in pattern recognition (Guzman 1968). This approach has influenced work of Negroponte, Herot and Taggart, in architecture as applied to sketch recognition (Herot 1976; Taggert 1975; Negroponte 1973). This work was later continued by Y. Do and M. Gross (1995).

In an effort to make application programming languages -as described in the next section- move to a higher level and closer to natural languages, Hewitt, Sussman, and Winograd among others proposed so-called planner languages. The objective was to go beyond LISP and its variants by incorporating the capability of reasoning (logic) and of constructing step by step plans to attain a specified goal, such as stacking a series of boxes in a corner of a room (McCarthy 1966; Steele 190; Hewitt 1971). But they failed, and one of the important reasons was that there was no efficacious way to deal with common knowledge and understanding of the world. However, one language succeeded in merging logic with programming, and that is PROLOG (Kowalski 1977).

The natural language procesors constitute another “H” category language, for recognition and answer synthesis, which is is an area that has received much attention in AI (Barr 1981). Not only that, there is product which has used the results of natural language interface for commercial applications called INTELLECT (Martin 1986). The objective is to eventually provide an interface to permit a “normal” conversation to take place between a person and a machine. The initial and much of the current effort has been to attack this problem on a general level regardless of the application. The success of this type of program has been limited, mostly due to the problem of semantics and how to represent knowledge of the every day world.

application implementation languages

The first examples of an “A” category language reside in interactive computer graphics systems. These contain two types of language systems: command languages and implementation languages. The first group depends on the graphics package implementation and there are no well established standards in this area. The second group of languages mostly entail the modification of a standard computer programming language to include data types such as points, lines, surfaces and the like (Foley 1982; Newman 1969). The issue here is what is the best computer language for this application. Many designs and implementations have been carried out in the beginning with Fortran and now C/C++. In most of these cases, the motivation of the researchers and developers of these languages rarely has to do with providing a system specifically for architectural design. Most of the effort has been in CAD in engineering, flight simulation, physics, chemistry, advertising, business graphics, geology, cartography, special effects for movies, etc. The field grew very rapidly from 1975 onwards and has celebrated the creation of images that look as realistic as possible and processing efficiency. In this context, graphic languages have been very successful, culminating in full length feature animated cartoon such as Toy Story.

Programming languages, such as LISP, FORTRAN, C; and ADA, have been developed primarily for applications in science, business, and engineering. An exception is Glide developed by Eastman (Eastman 1979) which was conceived specifically for architectural applications. The great majority of these languages have very little to do with solving the problem of making a machine understand architectural design issues (Sammet 1969; Wexelblat 1981).

Since 1988, in the area of CAD, the two most prevalent languages for application development are C -and later C++- and a variant of LISP called AutoLisp (Autodesk 1997; Bentley 1995). Again, these languages do not have as an objective to make a machine understand architectural design issues, but rather implement quantification procedures for architectural and engineering applications.

Since 1975, in the area of computer science, there have emerged one or more (depending upon one's definition) so called object based computer languages. The most well known ones are Smalltalk-80 -first conceived by Kay (Goldberg 1983) and Ada (Haberman 1976). It can be argued that they represent the "state of the art" of the most widely distributed object based languages. Many other lesser known object oriented languages have been in development over the last 10 years, such as Oberon (Wirth 1992) or TM (Gerzso 1985, 1987). Even though they have been extensively researched and developed, it is not clear which ones will ultimately survive. The emergence of these languages from research facilities to the general community of computer data processing has made the idea of object based systems very fashionable and the subject of many papers.

To those who may have only recently become aware of object based languages, it may strike them that such systems have been developed fairly recently (circa 1990). The actual historical facts are otherwise. Object based systems and also closely related concepts of data abstraction can be traced back to SIMULA reported in 1958. First versions of Smalltalk and CLU by Liskov date from the early '70's. (Krasner 1983; Liskov 1981). In the case of Simula, the developers attempted to understand the notion of data types, how they can be defined and what are the characteristics of their operations. Subsquent language designs addressed this problem in its own way and each one has contributed concepts to the field. And yet, to date, there is no complete consensus as to what these languages should be like (Wegner 1986), even though C++ (Deitel 1994) is the one that dominates the industry at this time.

The next module that corresponds to the logic engine may or may not have a language. In many systems, Lisp is used as a logical language. This is achieved by its extensibility feature. However, since the introduction of PROLOG, logical clauses have been shown to function as a kind of programming language (Kowalski 1977). The idea is that the programmer states the results or goal he or she wants to achieve and the language attempts to achieve the results. This contrasts with the usual approach to programming in which the programmer must work out how the program will achieve the desired results. Despite the fact that PROLOG is most interesting, it has yet to displace Lisp as the workhorse language in AI.

Most of the effort in developing logic engines including PROLOG has been directed at solving the mechanization of logical reasoning. The important feature of this language is that the system determines that something is either true or false based on a set of assumptions and rules of inference. These two groups of statements make up the program. There has been little work done relating such mechanization to architectural design. As for so called deontic logic machines, at least in design methods, little has been done to mechanize this kind of reasoning.

The last but not least important application type language are those related -or part of- data bases. They are usually referred to as data definition and query languages. The most prevalent one is SQL which as first developed by IBM and has since then become the industry standard in relational data bases. The programmer states the conditions that the information to be retrieved should meet, and the data base attempts to find that information (Feuerstein 1995).

system programming languages

The languages used for "S" type projects are usually asssembler and C/C++. The programs implemented in these languages have to do with the inner workings of operating systems such as UNIX or Windows 3.X or NT. All CAD systems, including the hypothetical CLF presuppose the existence of "S" systems, and any CAD intelligence must developed in addition to them.

shape grammars and DPR's

In the architectural CAD area, there has been many publications on shape grammars (SG) and its relation to languages (Mitchell 1990). But in the context of the CLF discussion, shape grammars are not languages in the same sense as a natural language or programming language. Their linguistic origin is a result of two historical developments: on the one hand, there is the metaphor of architecture as language, and on the other, the application of techniques derived from grammars evolved from the work first done by Noam Chomsky in linguistics (Chomsky 1956). The formalization of grammars is based in part on the idea of production rules, which is a way of modelling languages.

Shape grammars are variations on the use of production rules as applied to architecture, developed by Mitchell and Stiny. In this case, the idea of modelling combinatorial rules of architectural styles is a way of representing design knowledge.

However, a conceptual confusion begins at the time production rules or grammars are applied to pictures and architecture. As long as pictures are mapped onto strings that belong to a language (such as done in PDL by Shaw), the production rules representing the language can still be thought of as a grammar. But when pictures and architecture are no longer mapped onto strings, then the set of production rules can no longer be conceived as a grammar, for neither pictures nor buildings are examples of sentences in a language. For this reason, the present author has proposed (Gerzso 1979, 1985) production rules in the form of diagrams called Diagrammatic Production Rules (DPR's), which represent spatial combinatorial rules that are valid in a given architectural style. They represent aspects of architecture and not languages. Thus, the terminology is architectural. For example, instead of saying that the set of production rules is a grammar, we say that they are an order, as in architectural (Greek) orders. These orders were a kind of intuitive rules of design in a given style. Instead of saying that a design of a building is a sentence, we say that it is a variant, which is an idea proposed by Habraken (1976). Instead of saying that a space is a word, we call it a spatial primitive. And finally, instead of calling a group of spaces, which are part of a design, a phrase, we call it a configuration.

SG's and DPR's have a common goal in attempting to account for the generation of specific architectural solutions within a given style. Because both of them provide (at least in principle) unambiguous mechanisms for generating architectural solutions, it then follows that they can be implemented in a machine. However, SG's and DPR's are not very smart in comparison to a real designer, but they are smarter than improvising rules while hacking computer code.

The motivation for considering DPR's and SG's as knowledge representation is to emphasize the importance of these endeavors. In fact, they are the most important part of an intelligent design system because without it, the rest of the system, such as a CLF, would have little reason for being. Excluding DPR's or SG's or some equivalent method would be analogous to attempting to train an architectural designer, but failing to provide the expertise in the process. This is well understood in AI where it is generally recognized that to develop an expert system, basic problems to overcome are:

- Finding ways to make explicit the knowledge of a given field so that it can be representedin a program. If the field or profession does not have the tradition of making its knowledge explicit, then the effort of programming an expert system may yield poor results.
- Finding an expert who has the ability to organize and present knowledge explicitly to the developers of the expert system.
- Finding computer specialists who can supply additional techniques (called opaque knowledge) which permit representing and using explicit knowledge in a machine (Barr 1981).

the computer language framework

Given existing technology, the purpose of the discussion up to now has been to identify those places in an intelligent design system which require the use of a language system, in one form or another. It has served as a frame of reference for what is to be dealt with next.

As we analyzed the characteristics of the various languages as a group in terms of existing technology, one aspect stands out above all others: there is a special language for each module. Each one is usually not compatible with any of the others. For example, the graphics language may be different than the implementation language, which in turn is different from the query language, the logical language, the operating system command language and so on.

Historically, there have been two approaches to developing languages and systems. The first one assumes that one language should be adequate for all software projects. The most famous examples have been PL/1 and ADA. This can be called the vertical approach to system implementation. The second one assumes that there is a language for each application area or type. For example, for AI, LISP is used, for data bases, SQL is used, for Windows applications, C/C++ is used, and son on. This one can be called the horizontal approach.

According to the first strategy, one language should be developed in order to satisfy the entire CLF. That is, *it should be adequate for each of the three language categories "H", "A" and "S"*. Therefore, the goal would be to eliminate the various languages and substitute for them one language. This one language would function as an end user language and a programming language. An exception may be in the use of a language for implementing a particular module, such as a data base or operating system.

In order to clarify the nature of an integrated language system, it is useful to spell out some of its important characteristics. That is, the language must be:

- similar to a natural language with its corresponding knowledge representation mechanisms;
- able to represent objects in 3D space;
- function as a programming language;
- integrated with a logic and/or constraint engine;
- integrated with data bases which contain the semantics of architectural built form and knowledge about the world in general.

It is recognized that to achieve many of the specifications listed above, *major efforts* in language research may be required. For example, it is not clear how to design and implement a language system which on the one hand is similar to a natural language, but at the same time functions as a programming language and contains a data base and a logic engine. Some features have been achieved in INTELLECT, a fourth-generation language which is integrated with a data base (Martin 1986). Smalltalk-80 has succeeded only in integrating an implementation language with an interactive graphic interface. And in general, history has not been kind to those languages that have tried to satisfy all of the programming requirements.

The second strategy requires a standardization of the way in which CAD sub systems communicate with each other. This is known as the system integration problem. This would require that an IDS system be developed using several different languages. Even though this appears to be messier than the first strategy, it has proved to be the strategy most used, at least in the "A" type AI systems used in industry.

conclusion

In relation to the "H" part of the IDS system, there is still much to be developed in the commercial CAD systems. Functionality such as complete sketch or drawing recognition is still not a reality, although some systems can perform some raster to vector conversion. Natural language recognition for architectural applications, whether typed or spoken, is even farther away into the future than drawing recognition.

Languages for "H" systems advanced rapidly from assembler to LISP in the early 1960's, but have not advanced at the same rate in the last 20 years. These were termed high level languges; the "high level" part was in relation to the assemblers. However, the next generation of even higher level languages have not materialized, even though much effort was invested in systems like PLANNER and CONNIVER. So, it is safe to assume that CAD systems will not help designers do a better job by means of intelligent human interaction software at least in the next 10 years.

As we have seen, for "A" type systems, languages are important and indispensible components of any computer system, whether it be for communication with a user or for programming. In general, the development of a computer language is the result of attempting to facilitate the interaction with a machine or its programming for particular kinds of applications. As a consequence, there exists a situation which is akin to the Tower of Babel in many computer systems. This has been tolerated up until now because of this diversity of applications and the investment in existing systems which cannot be easily discarded. But, it has been found that typical high level languages such as FORTRAN, PL/1, COBOL, C/C++, Lisp, PROLOG, ADA, Java and the like are not satisfactory for a system that requires a machine-understandable design language (CLF) because of the following:

- The syntax and semantics require special training to use and understand–that is, they are very arcane compared to natural languages.
- The expressive power of the languages is extremely limited. A language with the "semantic richness" closer to that of a natural language is what is required.
- Some are not extensible–that is, the programmer must accept the given data types of the language.
- They are usually not integrated with a data base, graphics system or logic engine.
- They usually require that the programmer deal with routines to maintain information in memory and on secondary storage.

In conclusion, the improvement in the quality of architectural design in the near future will not be result of intelligent design systems. As has been argued, the prime reason for this is the lack of a dramatic advance in programming and human interface languages that are closer to the semantic and expressive power of natural languages. The thesis is that CAD software development is closely tied to language development, and as we have seen, most of the development has been in "A" type languages.

How will this affect research in architectural CAD? Assuming that the thesis is correct, research agendas should concentrate on using technology of the "A" type, and only embark on projects addressing "H" type issues with the understanding that it may take decades to complete.

references

Alexander, C., et. al., 1977. *A Pattern Language*. New York, NY: Oxford University Press.

Autodesk Inc., 1997, *AutoCAD R14 Customization Guide*, California.

Barr, A., E. A. Feigenbaum, 1981. *The Handbook of Artificial Intelligence*, Vol. I-III. Los Altos, CA: W. Kaufmann.

Bentley Systems, 1995, *MDL Programmer's Guide*. Exton, PA.

David, R. E., 1972. "Proposal for Diagramatic Language for Design," *Visible Language*, Vlc / o The Cleveland Museum of Art, Cleveland, Ohio.

Do, E. Y., and M. D. Gross, 1995. "Drawing Analogies: Finding Visual References by Sketching," in *Proceedings,* ACADIA'95, Seattle, WA.

Calderon, M., 1992. *Un Compilador Para TM, Un Lenguaje de Programación Orientado a Objectos*, Undergraduate Thesis in Computer Science, Fundación A. Rosenblueth, Mexico D. F., Mexico.

Cardenas, S., 1986. *Una Máquina Virtual para TM*, Master's Thesis in Computer Science, National Autonomous University of Mexico (UNAM), Mexico D.F., Mexico.

Chomsky, N., 1956. "Three Models for the Description of Language," *I.R.E. Transactions of Information Theory IT2.*

Deitel, H. M., and P.J. Deitel, 1994. *C++ How to Program*. Englewood Cliffs, NJ: Prentice Hall.

Eastman, C. M., and M. Henrion, 1979. "GLIDE: A System for Implementing Design Databases," *Proceedings of (Planning, Architecture and the Computer) PArC 79*, Berlin, West Germany.

Feigenbaum, E. A., and P. McCorduck, 1983. *The Fifth Generation, Aritificial Intelligence and Japan´s Computer Challenge to the World*. Reading, MA: Addison-Wesley.

Feuerstein, S., 1995. *Oracle PL/SQL Programming*. Sebastopol, CA: O'Reilly &Assoc.

Foley, J. D., and A. Van Dam, 1982. *Fundamentals of Interactive Computer Graphics*. Reading, MA: Addison-Wesley.

Fu, K. S., 1974. *Syntactic Methods in Pattern Recognition*. New York, NY: Academic Press.

Gerzso, J. M., 1979 *A Descriptive Theory of Architectural Built Form and its Applications*, Ph.D. Dissertation, University Microfilms.

Gerzso, J. M., 1979. "Spacemaker, A Computer Language for Modelling Architectural Physical Form," *Proceedings of (Planning, Architecture and the Computer) PArC 79*, Berlin, West Germany.

Gerzso, J. M., and A.P. Buchmann, 1985. "TM-An Object-Oriented Language for CAD and Required Database Capabilities," *Languages For Automation*, Chang, Shi-Kou (ed). New York, NY: Plenum Press.

Gerzso, J. M., 1987. *Report on the TM Language Design, Version 5*, Internal Working Document at the Institute for Research in Applied Mathematics and Systems (IIMAS), National Autonomous University of Mexico (UNAM), Mexico, D.F., Mexico.

Gonzalez, R. C., and M. C. Thomason, 1978. *Syntactic Pattern Recognition, An Introduction* Reading, MA: Addison-Wesley.

Guzman, A., 1968, *Computer Recognition of Three-Dimensional Objects in a Visual Scene*, Technical Report 228, AI Lab, MIT, Cambridge, Massachusetts.. See also section of pattern recognition in *The Artificial Intelligence Handbook*, Vol. III, A. Barr, et. al., (ed).

Goldberg, A., and Robson, D., 1983. *SMALLTALK-80, The Language and Its Implementation*. Reading, MA: Addison-Wesley.

Haberman, N., and P. E. DeWayne, 1976, *ADA for Experienced Programmers*. Reading, MA: Addison-Wesley.

Habraken,J., et. al., 1976. *Variations, The Systematic Design of Supports*, Laboratory of Architecture and Planning, MIT, Cambridge, MA.

Herot, C., 1976, "Graphical Input Through Machine Recognition of Sketches." *Computer Graphics, SIGGRAPH Quarterly Report*, Vol. 10, No. 2.

Hewitt, C., 1971. *The Description and Theoretic Analysis (using schemas) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot*, Ph.D. Thesis, MIT.

Hewitt, C., 1975. *A PLASMA Primer*, AI Lab Working Paper 92, MIT, Cambridge, MA.

Hopcroft, J. E., and J. D. Ullman, 1969. *Formal Languages and Their Relation to Automata*. Reading, MA: Addison Wesley.

Khemlani, L., and A. Timerman, B. Benne, Y. E.

Kalay, 1997. “Semantically Rich Building Representation,” in *Proceedings,* ACADIA’97, Cincinnati, OH.

Knapp, R. W., and R. McCall, 1996. “PHIDIAS II-In Support of Collaborative Design”, *Proceedings of Acadia 96*, Tucson, Arizona.

Kowalski, R., 1977. *Predicate Logic as a Programming Language*. North Holland, Amsterdam.

Krasner, G. (ed), 1983. *SMALLTALK-80, Bits of History, Words of Advice*, Reading, MA: Addison Wesley.

Liskov, B. H., et. al., 1981. *CLU Reference Manual*, Lecture Notes in Computer Science. New York, NY: Springer-Verlag.

Martin, J., 1986. *Fourth-Generation Languages, Vol. I.* Englewood Cliffs, NJ: Prentice-Hall.

McCarthy, J., et. al., 1966. *LISP 1.5 Programming Manual*. Cambridge, MA: MIT Press.

Mitchell, W. J., 1990 *The Logic of Architecture*. Cambridge, MA: MIT Press.

Mitchell, W. J., 1979. “Synthesis with Style,” *Proceedings of (Planning, Architecture and the Computer) PArC 79*, Berlin, West Germany.

Negroponte, N., 1973. “Recent Advances in Sketch Recognition,*” Proceedings of the National Computer Conference (NCC),* New York, NY.

Newman, W. M., and R. F. Sproull, 1969 *Principles of Interactive Computer Graphics*. New York, NY: McGraw-Hill.

Sammet, J. E., 1969. *Programming Languages: History and Fundamentals*. Englewood Cliffs, NJ: Prentice-Hall.

Shaw, A. C., 1970, “Parsing of Graph-Representable Pictures,” *Journal of the ACM*, Vol. 17, No. 2.

Steele Jr., G. L., 1990. *Common Lisp*. MA: Digital Press..

Stiny, G., 1979. “A Generative Approach to Composition and Style in Architecture,” in *Proceedings of (Planning, Architecture and the Computer) PArC 79*, Berlin, West Germany.

Sussman, G., 1972. *Why Conniving is Better than Planning*, AI Report 255, AI Lab, MIT, Cambridge, Massachusetts. See also section on AI languages in *The Handbook of Artificial Intelligence*, A. Barr (ed), et. al.

Taggert, James, 1975. “Sketching, An Informal Dialogue Between Designer and Computer,” *Reflections on Computer Aids to Design and Architecture*, N. Negroponte (ed). New York: Petrocelli / Charter.

Waltz, D. L., 1972. *Generating Semantic Descriptions From Drawings of Scenes with Shadows*, Technical Report 271, AI Lab, MIT, Cambridge, Massachusetts. See also section on pattern recognition in *The Handbook of Artificial Intelligence*, A. Barr (ed), et. al.

Wegner, P., 1986. “Classification of Object-Oriented Systems,” *SIGPLAN Notices*, ACM, Vol. 21, No. 10.

Wexelblat, R. L., 1981. *History of Programming Languages*. New York, NY: Academic Press.

Winograd, T., 1981. ”SHRDLU,” *The Handbook of Artificial Intelligence*, A. Barr, E.A. Feigenbaum (eds). Los Altos, CA: William Kaufmann.

Wirth N. and M. Reiser,. 1992. *Programming in Oberon: Steps Beyond Pascal and Modula-2.*. Reading, MA: Addison Wesley.