9/18

TECH MEMO

a working paper

System Development Corporation / 2500 Colorado Ave. / Santa Monica, California

TM-2456/007/00

AUTHOR E. Franks

TECHNICAL

RELEASE S. M. Cooney

for D. L. Drukey

DATE 9/6/63 PAGE 1 OF 12 PAGES

(1)

DEVELOPMENT AND MANAGEMENT OF A COMPUTER-CENTERED DATA BASE:

PROCEEDINGS OF THE SYMPOSIUM (10-11 June, 1963)*

Part 7: LUCID

E. Franks

*This symposium was conducted by the Command Systems Department's ARPA Command Research Project, in cooperation with the Advanced Research Projects Agency. This talk, one of the presentations at the symposium, is to be incorporated in a Technical Memorandum containing the complete proceedings.

LUCID

E. Franks*

We in SDC's Command Systems Department feel that in research directed toward solving military command and control systems, one of the big problem areas is the management of data. We feel, in fact, that any computer-based system is essentially a data-management system, at least in the area of command and control, and that certain problems have never been solved in the past. We feel, in the first place, that the data themselves are an integral part of a system concept. Commonly, in the past, designers have begun system designs without any clear idea of what the data involvement was; often, in fact, people suddenly asked, "What do these files we talk about actually look like?" Part of our effort at SDC has been to provide a tool that will help to solve the problem of system data content and structure earlier in the design process. This is part of the function of the system we call LUCID.

Another thing that designers have often experienced is that, by the time a system is formulated and has gone to the coders and been coded, it's already obsolete. The result is that most programming systems have had a very short life and many of them have been stillborn. The problem that LUCID attempts to attack is the enormous time lag between conceiving a system, and getting something running on the computer. Concomitant with this time lag is the expense; to get a system going and then not use it seems to me to be a luxury we can ill afford.

The LUCID project is based on a concept of a system design, namely that computer-based command-control systems are data-management systems and that the problems of data base are central to the problems of design. LUCID provides both a language in which design of data management systems can be expressed, and a programming system to convert this expression of design automatically into something that can be run on a computer. The project, ~~at present~~, is engaged in formulating the design concept of data-oriented systems; in the establishment of a language that carries design from the man to the machine, and in the construction of a preliminary model of such an automatic programming system, which can be used, not only to validate the general concept itself, but also to provide a useful tool for our other research efforts requiring a data-management facility.

One large problem is that systems change--what you start out with is very seldom what you are going to be satisfied with, even in a very short time. We are trying to provide a method for getting something on the computer fast, experimenting with it, modifying it, so that the evolutionary process can be continuous. We

*A member of the SDC programming staff in the ARPA Command Research Project, Mr. Franks is working on programs for handling a data base. His programs, called the LUCID (Language Used to Communicate Information System Design) System, are being developed as a tool for system-design research.

have envisaged two methods of operating this system. One would be an interpretive method in which inputs are given to the system; outputs are obtained, examined and evaluated; and changes are made through a feedback loop that continues over a period of time. Then, when the system appears to perform according to the current expectations, a generation process would be undertaken to produce automatically a set of noninterpretive programs that perform the same functions that the interpretive system just exercised had performed. This second mode would operate more rapidly on the computer, to meet some real-time constraints that an interpretive system might not be able to meet However, we think that a user who is doing this should not abandon the interpretive system from which he made this generation, but should continue to use it for development purposes. In other words, he squeezes off a specific version of his system whenever he feels a significant improvement has been made. Once the generation process works perfectly, the user never modifies the generated version of his system, but he does continue to experiment with the interpretive version, and, from time to time, produces new models through the generation process. Through this means we hope to have all the flexibility inherent in interpretive systems, plus access to noninterpretive versions of systems that would give us the computer efficiency associated with them--the best of both worlds, in fact.

A further possibility along these lines is that generators could be developed for several different machines, so that if the user wanted to operate the same system on different hardware, or parts of a system on different pieces of hardware, he would have a method of simulating the whole system on the computer that he used in the interpretive mode, and producing the generated versions for whatever machines were desirable.

Another aspect of the interpreter-versus-generator concept is that some functions of systems are probably not improved by being generated to a specific program--in particular, such things as what is known as dynamic retrieval, that is, the retrieval of data in sets that are not predictable. So, we are thinking in terms of partial generation so that certain portions of a system could be produced as a set of direct programs, while other portions remain in an interpretive form.

The way that we have gone about our design and development is, I think, the opposite of the case of MITRE Corporations's ADAM. We started with the idea of a language in which the kind of things you would like to do on a computer can be expressed. We found that, even though there is an infinite number of ways of doing things on the computer, the number of things that you do is quite finite and surprisingly small.

The idea of our vocabulary, then, is that it contains terms for these basic operations and for the basic relationships that are tested on a computer or are used to express algorithms on a computer. A certain grammatical arrangement of these terms is understood by our program. However, we do not wish to tie users to our particular versions of these terms. We tried to canvass opinion

and find out what was a good set of terms and we found that people were not really prepared to answer, so we arbitrarily selected a set of our own; but we do provide the facility for the user to substitute his own term for these to any extent that he wants. He can describe his system in French, if he wishes, if he first explains to the system the French terms that correspond to the terms understood by the system. Then, with this set of basic operators, we decided that the user should be able to talk about the system--his particular system--in his own terms, terms that are meaningful to the analysts working in his area.

The way a system is described in LUCID is that the user takes some terms, which can be (although I hate to use grammatical analogies) nouns--that is, the names of data; the names of files, or processes. He uses the word to which he wants to assign this particular meaning in terms of his own system and he defines it in the basic terms that are known in advance--the a priori terms known to the LUCID system. Then, once he has defined one of these terms, he may combine it with others, or with others of the basic vocabulary, and build up more complex sets of data or more complex sets of operations. In this way, he ultimately defines his whole system. Finally if he wishes, he may define his whole system with a single word, such as OPERATE. This word OPERATE will have been defined in terms of a whole hierarchy of things previously defined. Or, if he wishes, he may leave the system in pieces so that he can operate parts of it by calling on the names he has assigned to the various pieces. I think this is something that goes along with the evolutionary approach that we propose in developing the system in LUCID. You start simple and, as you become satisfied with the series of operations that may recur under several different circumstances, you name them and give them the full set of definitions; and from then on, whenever you need to use the series of operations, the single term will trigger the operations that you defined.

Since we are not, fortunately, bound by any production contract that requires that things must be done by such-and-such a time, or that certain things already exist that must be accommodated, we're able to operate rather freely. Our aim is maximum freedom for the user, so we have set no arbitrary limits on the length or the complexity of the names that the user may give to his data or his processes. He may choose to call something "THE NUMBER OF TIMES SITUATION X HAS OCCURRED"--all that--if he is willing to repeat it all every time he wants to refer to it. He may designate some value by such a name. We feel that people will naturally tend to use short terms simply because of the labor of writing them.

We are setting up two modes for the use of LUCID. We think that, in most systems, there is a certain amount of processing that is constant--that is, the requirements don't change very much in the short run. So, for one mode we can establish file building, file maintenance, and standard reports in advance, as part of the system, with some confidence that they will remain unchanged for long enough to make generating this portion worthwhile.

The other mode is an on-line mode, in which the user may want to try something just once, or it may be an abnormal request that he doesn't expect to be repeated. He doesn't wish to add the series of processes involved in this request to the permanent system that he has described. In the on-line mode, the request would be translated and performed but the translation would not be added to the system as permanently described. On the other hand, he may, at any time, change the permanent description of the system.

LUCID, in fact, is a programming system--one that avoids the detailed level of specification that results in the time-consuming process with which we're familiar We have tried to provide operators that are simple to use and that reduce the amount of coding--the amount of description required to produce the results that the user wants. It's very hard to strike a balance between having too detailed a language that would not be essentially different, say, from FORTRAN or JOVIAL, and setting up a language that has so many special terms in it that it would be very difficult to learn and use We decided that we would provide special facilities with the vocabulary principally in the area of data base manipulation. In the description of data base organization, the description of sets of the data for retrieval or for making a smaller data base that is a logical subset of another data base, we have tried to provide the most powerful operators, so that as much of the process as possible is automated, and the specification required by the user is minimised.

For storage and retrieval of data, we assume that usually there will be a normal use or maximum use of a data base and that the organization of the data base should be optimized for this particular use.

The LUCID language provides for the description of such an organization, and the system automatically sorts the data into this specified organization when they are loaded or updated, at the same time providing retrieval indices based on the organization criteria. For this maximum use of a set of files the retrieval path follows the storage path for maximum efficiency. Unfortunately, in real-life systems, we always find different uses of data, for each of which the optimal organization is incompatible. With several high-level uses of a data base in which the optimal organisation for each use is different, we provide a subsetting process, which may be a one-to-one subset, or may involve a merging operation as well as a subset. For the different uses, you have the data stored in different ways. This, of course, introduces problems of duplicate updating and duplicate file maintenance. We have no magical solution to these problems.

In the extreme case, in which the use of the data base is completely unpredictable--in which it is a master file that may be accessed on the basis of any of the values in it (I think this is probably quite a rare case, but in experimental work, it certainly may well arise)--we provide the capability for a complete cross-indexing, which we call a "concordance " What this involves, in effect, is producing the data base twice: once in some logical order; the other time, as a list of logically connected cross references. We feel that,

in the case of a data base too large to be contained in a random access or high-speed access memory (and particularly a data base recorded on more tapes than can be mounted at one time on the computer equipment), "concordance" retrieval is always better than random search, and will pay off, although it is not optimum for any one particular retrieval.

To describe a system in LUCID, hierarchies may be established that we call executive systems and subsystems--you may name your control portion when you describe it as an executive system; you may then describe other portions of the system, and state that they are subsystems of the name to which you have assigned the appellation "executive system." The subsystem then has access to the data of the executive system. Any common data, then, would be described as belonging to the executive system, and special data that relate only to a certain set of processes in a subsystem would be described as belonging to that subsystem. In connection with this, we provide a system of programmed interrupts and priorities so that different users may have simultaneous (at least, in human terms) access to the common data in the system. We are fortunate in that we have a machine for which this kind of program interrupt is fairly easy. We haven't really faced the problem of a machine without a clock, or something like it for which some artificial means would have to be provided.

I would like to say something about LUCID as a programming system. We tried to experiment with some of the things that programmers have always said you should do, but that they seldom do. We have modularized to an extreme. We have a "rule of thumb" that no procedure would exceed 60 JOVIAL statements (and, I think the average is well below this). We found out, in actual practice, that we have greatly shortened the total amount of code at the expense of increasing the number of procedures. We found very many cases--some in quite unexpected places--where duplicate lines of code occurred. These were taken out as separate procedures.

The system is in four gross parts. We have our own executive system control. It is quite short, as executive systems go, and what it is going to do next may be controlled by some operating portion of the system. In other words, LUCID is pretty well data-sequenced, and the operating systems that have been described by the user may direct the sequence of events by setting data flags, which have also been identified by the user in his description. This control portion is the link between the user who is to operate the system, and the LUCID system. It processes control functions, such as describing the computer environment. We operate with other systems that sometimes have a higher priority and may have to have their tapes left on. Sometimes tapes are down. This is the kind of communication that the control processes. You can tell the control that the inputs are on-line, on the on-line card reader; that they're by teletype; that they're on a pre-stored tape; etc. These, we assume, are not primarily the concern of the user, but of the computer-operating staff, and these people are the ones who will communicate with our system control.

One of the operators recognized by the Control is the word TEXT. This means that what follows is a system description in the LUCID language. When this operator is recognized, the control is passed to another section of the system called the Translator, which processes the description of the system. We have made it so that you may refer to data that are not yet defined. It's a two-pass system in the sense that, after all the inputs are read, there is another phase in which they are collated and any discrepancies--anything that finally is undefined, anything that is defined in duplicate, etc.--are flushed out. This second pass is a drum pass in the case of the Q-32.

The Translator output is a series of tables; some are data-description tables and some are process-description tables; another level we call a "mode table." A mode consists of one or more tasks, a task being the environment and sequence necessary to perform a process as described by the user. Every task is also a mode. When one process defined by the user employs other processes that he has defined or will define, a multi-task mode results. A mode describes the sequence of tasks that can be predicted, that is, the sequence of tasks that do not include a decision as to the next process. At the end of each mode, then, is the specification of a switch--a data switch--that determines which mode table will be processed next. The operating program, by setting the values for the data switch, controls the sequence of processes in the system.

When LUCID is assembled it is empty--that is, it knows nothing about its user's wishes. Once a system has been described and translated, the new master resulting from the translator still has the full capability of LUCID to understand new descriptions, modifications and operational commands. It also understands the data and processes that the user has described. The master tape is now the user's own LUCID system, and will, in fact, carry a label that is the name the user has applied to his system. The presence on this master tape of the LUCID system itself makes it possible for the user to change this system or add to it; to refer to data already defined; and to define new data. These operations result in a repetition of the second pass of the Translator to collate the new information with the old. Various system-description updating functions are possible, such as eliminating processes or data altogether, changing them in part or in whole, or adding to them.

The interpretive mode of operation of the system is performed by what we call the Interpreter, which, again, is a series of generalized processing routines, some of which are controls for others. These are driven by the tables that are output by the Translator. The controls are driven by the mode tables and the rest of the Interpreter is driven by the data process-description tables.

The Generator is the fourth part of the system. We have several ideas on how to construct this although we haven't started doing it yet. The simplest way we see, and the way that we will probably start with, especially for the partial-generation scheme in which certain parts of the system would run interpretively and certain parts directly, would be to take some of our substantive general-purpose processing routines and store them in skeleton forms. These skeleton

forms would be filled in, that is, specific addresses, index values and constants would be supplied in the code by a generator program. The appropriate set of these skeletons would be assembled to perform the functions of the particular task, and the whole would be stored on the master tape as a closed procedure to be called, together with the necessary data environment, whenever the particular task was needed. The system control would be the same for both interpretive and generated operations. The mode table would still control the sequence of processes.

Another method of generation, which we think shows much more promise, is to treat the parameter tables output by the Translator as a data base logically equivalent to a set of machine instructions. Through use of the power of LUCID itself to specify data transformations and to subset one form of data base from another that is its logical equivalent, the parameter tables would be converted to another form of data base that would be, in reality, a string of instructions for a particular machine, such string being the one required to perform the process described in the parameter table from which it was derived.

What we're working on currently is a version that does not have all the features that we have thought of and feel are desirable. It operates only in the interpretive mode and does not have any generation capability. It will have a certain limited on-line capability but, initially, probably not with anything but cards and possibly teletype. It is essentially a data base managing system that will allow the people in our department who have data problems with which they wish to experiment to build data bases in various forms, and to perform these experiments. One of the things that we are developing is planned for this initial version: a data generator that will make up artificial data following specifications that allow various degrees of randomness or control, and will allow the ingenious user to build a data base that contains values with the kinds of statistical distribution that he wants. With this preliminary system a user can perform the following operations on a data base: he can load it, either from real inputs or from artificially generated ones; he can abstract from an existing data base others with, perhaps, a different organization and a lesser content; he can form a third data base by merging two existing data bases, selecting from each a part of its content; and he can retrieve from a data base, through a querying process, those entries that meet the criteria expressed in the query. In the process of transferring items of data from one data base to another, the user may specify arithmetic operations to be performed on them--for example, the value for an item of data may be converted from miles-per-hour to feet-per-second as it is transferred. Certain summary values may be derived from a selected body of data in the forms of totals, averages or other common statistical measures. Finally, the user may specify formats in which the output of an operation will appear on the printed page. (So far, we are not thinking of displays because our display equipment isn't available yet, or it wasn't when we started this, so the capability is confined to printed formats.)

To allow the user freedom to use his own vocabulary, we are trying an automatic tagging system, whereby we take the user's term and give it a unique tag within

its data base; (if it is a process name it has to have a completely unique tag). We've often felt that for data that are essentially the same, i.e., mean the same thing but are in different files, it is very onerous to have to think of new names, maybe a dozen times, for the same piece of data. So we are experimenting with providing a localizing capability in naming the data so that, for example, you may say that you want the aircraft number in the generation file, or the aircraft number in the SORTIE file, and even though the information itself has the same name in both cases, you localize it by naming its file.

We are experimenting with this because we feel it a necessary concomitant to the free use of user terms in naming data. Once the data are in the system, LUCID refers to them by numeric tags so that the references in the tables that describe processes would be the numbers, and not the original user designation. In the generated version, of course, they would be in the form of addresses and indices. When the user wishes outputs, we have to translate back to his terms to make his outputs meaningful to him. We wish to give these data back to him, to let him use them in input and receive them again in output. If all the data names are in a single large file--and in a large system, this can be a very large file--the process of retranslation on output (it seems to us) would consume excessive time. We limit the validity of a name to a particular file, and the environment that must be accessed to reproduce the user term is much smaller and shouldn't seriously affect the running time of the system.

I've tried to discuss some of the unique features of the system. In doing this, I may have left out a few essential features, so I would like to invite any questions to try to fill some of the gaps.

QUESTION: When do you expect to have your first LUCID running?

ANSWER: We have an impartial arbiter on that called the Q-32 The first version is about 85% coded, and a much smaller amount is checked out; because of the great degree of modularity and the fact that it is divided up into very small packages, it is difficult to know how much you have actually checked out. You check out one thing, and this aids greatly with the next and there is a snowballing effect, which we are already appreciating, and this makes us fairly optimistic. We're planning on having something significant running on the computer sometime in August.

QUESTION: You made mention of the fact that when a user compiles his system and describes his data base on a master tape, this includes a version of LUCID. How big is LUCID itself and how many instructions end up on a tape as a result of that?

ANSWER: LUCID (taking the first version, which we have a much better handle on), including the control tables that are necessary for its operation, is about 30,000 Q-32 48-bit words. Of course, the size of the tape that is produced as a result of the user system description is dependent on how much the user has described. The tables that represent his system in a parametric form are packed and as compact as we can make them. I really don't have any feel, until we've done some translation, of how you'd measure this.