Cena zł 24,-

omputation Centre of the Polish Academy of Sciences

Efficient Production of Large Programs

PAŃSTWOWE WYDAWNICTWO NAUKOWE

[illegible]

COMPUTATION CENTRE
OF THE POLISH ACADEMY OF SCIENCES

EFFICIENT PRODUCTION OF LARGE PROGRAMS

Proceedings of International Workshop
Jabłonna, August 10–14, 1970

This book contains transcripts of round-table discussions held at the Jablonna International Workshop on problems of large program making and design. The Workshop participants analyze sources of difficulties in the task of large program production, suggest some remedies and methodologies. The book is intended for professional programmers and research workers in the software field.

TEXT EDITOR
Barbara Osuchowska

COVER DESIGN
Alicja Szubert-Olszewska

TECHNICAL EDITOR
Jan Lipski

C O N T E N T S

P R E F A C E

This volume contains an almost verbatim transcript of the tape recordings made during the sessions of the International Workshop on Efficient Production of Large Programs held in the meeting centre of the Polish Academy of Sciences in Jabłonna near Warsaw, August 10 - 14, 1970.

The Workshop was organized by the Theory of Programming Division of the Computation Centre, Polish Academy of Sciences, in order to facilitate an informal discussion of problems in the broad area of large program design and implementation. As could have been expected, the participants in the Workshop were preoccupied mostly with the present-day inability to express pertinent subjects in a sufficiently rigorous manner and with the problems of selection of tools that could assist in large program making.

The similarity of views and problems indicated by the participants from several countries was a most encouraging experience for many of those who attended the Workshop Sessions; it has been decided to publish the discussion records so that a wider group could share this experience.

The only significant deviations from the chronological order of discussion were introduced in Session I which consists of records made during two physically distinct sessions; while editing it became obvious that the subjects discussed at these sessions were so close that a consolidation was necessary.

In all other instances, the editing process was restricted to the elimination of irrelevant repetitions and divagations straying off the Workshop's scope. Some of the more cumbersome phrases were straightened and the syntax was improved as far as possible without changing the semantics.

x

x x

The not easy task of transcribing the tapes was performed by Mrs. L. Żyła and Mr. A. Więckowski. Miss B. Osuchowska and Miss M. Rochowicz did their best to edit and type, respectively, the final version.

W.M. Turski

February 1971

P A R T I C I P A N T S

MIROSŁAW DĄBROWSKI Computation Centre of the PAS, Warsaw
HELENA DRYZEK, Computation Centre of the PAS, Warsaw
KONRAD FIAŁKOWSKI, Electronic Computer Laboratory, Technical University, Warsaw
MIROSŁAW GRZYMKOWSKI, Institute of Computing Science, University of Warsaw
C.A.R. HOARE, Dept. of Computer Science, the Queen's University of Belfast, United Kingdom
WALTER ISSEL, Institute of Applied Mathematics and Mechanics, Berlin, GDR
V.M. KUROCHKIN, Computation Centre of the USSR Academy of Sciences, Moscow, USSR
S.S. LAVROV, Computation Centre of the USSR Academy of Sciences, Moscow, USSR
EUGENIUSZ ŁUCZYWEK, Computation Centre of the PAS, Warsaw
LEON ŁUKASZEWICZ, Mathematical Institute of the PAS, Warsaw
JAN MADEY, Institute of Computing Science, University of Warsaw
GLENN K. MANACHER, Institute of Computer Research, University of Chicago, USA
ROMUALD MARCZYŃSKI, Computation Centre of the PAS, Warsaw
JÓZEF MAROŃSKI, Computation Centre of the PAS, Warsaw
JACEK OLSZEWSKI, Computation Centre of the PAS, Warsaw
BARBARA OSTROWSKA, Computation Centre of the PAS, Warsaw
WIESŁAWA PULCZYN, Computation Centre of the PAS, Warsaw
BRIAN RANDELL, Computing Laboratory, University of Newcastle u. Tyne, United Kingdom
(Workshop Co-Chairman)
ALINA RUSZKOWSKA, Computation Centre of the PAS, Warsaw
ANDRZEJ SADOWSKI, Computation Centre of the PAS, Warsaw
G.K. STOLIAROV, Mathematical Institute, Academy of Sciences of the BSSR, Minsk, USSR

W.M. TURSKI, Computation Centre of the PAS, Warsaw
(Workshop Chairman)
ANDRZEJ WIĘCKOWSKI, Computation Centre of the PAS, Warsaw
ELŻBIETA WYSMUŁEK, Computation Centre of the PAS, Warsaw

WORKING SESSION I : DEFINING LARGE PROGRAMS

TURSKI: The choice of the name of this Conference, and I will concentrate on one word only, that is on the word "workshop", is not incidental. We wanted to have the discussion of topics involved in production of large programs as informal as possible. Perhaps the word "efficient", added in the title, is somewhat premature. It does not seem that there are any formal recipes for efficient production of large programs. Many people are complaining that they produce lots of programs in a very inefficient way. Well, let us see what we can say about it.

The question which pertains most to the subject of frames of reference in which to speak about large program is, of course, the following one: What is a "large program" ?

When I think about large programs, it is almost inevitable that I think about two definitions.

One definition, which may seem a little facetious, is that the large program is a program which is never finished, program which is always being developed, and, indeed, there are many of those programs which are continuously being elaborated and improved and you have thirteenth or sixteenth release of a program, and people already use it for years and years, and yet you are getting new versions of essentially the same thing, perhaps with new facilities added to it, some blunders taken away, not all of them. This might be a definition of a large program, because with other programs you do get to a state when it becomes frozen, and no subsequent releases of such programs are made. I cannot give you many examples of such non-large programs which are really fixed, but at least some algorithms made into programs, i.e. written in some programming language, which are included in libraries of our computers, really do not change so often.

The second definition of a "large program" describes a large program as a program made by a large number of people, certainly by more than one programmer. That is, I would say that none of programs done by one programmer is a large program. This is, of course, only the necessary, but not the sufficient condition; you may have things written by more than one programmer, and yet you will be reluctant to call them "large". If we accept this as a criterion of the "largeness" of a program, then we will come up to a really interesting conclusion that it is human communication problems involved in producing common outcome such as large program which make it so difficult and so awkward to make.

When speaking about large programs, we are a bit confused about their design criteria. If, for a given task, we produce something which you may say is a non-large program, then the cost involved in such enterprise will be relatively small, so that you may attempt to make different programs for the same task, the programs different in the preponderant attitude taken in making them. You may try to produce for the same task two different small programs: one, which will be, say, user-oriented, and another - which will be hardware-oriented. That is, if the program is non-large, you may be really tempted to make two different programs to do the same task: one to be used in the development stage and another one for production runs. With large program this is not normally the case; we have to decide whether we are going to have it user-oriented (as close to the demands of the user as possible), or we make it as close as possible to the optimal usage of the hardware, or perhaps of the computing system which includes hardware and the software in which such a program is embedded. Certainly, we may decide, if we wish, on certain compromise in this field, in that case we shall get programs bad for hardware and not too good for users, but we certainly will not tend to make large programs in two versions: one, user-oriented and another hardware-oriented. Then, there are the problems of estimating the cost of large program production. Of course, we shall mean perhaps in different way the word "cost", but we all agree that software is not free, that software costs something and therefore we do agree that there should be a measure imposed on it. I do not think there are any satisfactory suggestions on how to measure the effort involved in producing a large program. Even more interesting problem is how to estimate the expected cost of a large program when you start making it. Connected with that is the problem of how to direct the actual work on production of large programs.

According to one of my definitions, a large program is a program in production of which we bring together efforts of many people, so you have a team at work. Now, how do we organize the team work on a single software product ? The cases are known when

the costs of communication between various divisions and groups, and subgroups working on a huge program were considerably higher than actual cost of productive work. Now is it incidental, or bad management, or necessity, that the cost of human communication in producing large programs, perhaps because of highly intricate nature of documentation involved, is so high.

As the production of large programs is going to be so expensive, is it possible to have it automated ? Is the automatic programming really possible when the programs to be produced are the ones which one would consider to be large ? We will see that both my definitions of large programs preclude such a possibility. A program produced automatically will be produced by the same mechanism always in the same way, so you will not have any next releases.

As of the second definition, which calls for many people working on the same program, of course it does not say anything about an automaton producing the same program.

In almost every paper on the subject of large programs you will find plenty of words like "modularity" and "stratified design", and "multilayer programming", "structured programming" and so on. At the same time, it is quite obvious at least from a superficial inspection, that if you have a program which is really modular and which consists of many little building blocks, then a lot of time and effort in execution of such program will go into transferring the information form one of these blocks to another. If the program is not modular, but a "monolithic" one, the necessity for this information transfer within the program will be relieved to a great extent, because it is just because of the modularity that you have to transfer information in and out of the module, perhaps in a very rigid form of intersystem messages. That means that since the modularity introduces this overhead of information transfer within the program, between parts of the program, you loose efficiency in a sense. You loose it in a sense because you gain it in another sense. An the other sense is that modularity means adaptability, that is, if the modules that you have programmed are designed clearly enough, when you have to change all program, you will not have to change the monolith, but you just change a little brick here and there and replace it by a little building block of similar dimensions with the same communication interface, but with different inside functions. So you gain efficiency because your program is better adaptable to changing requirements. Now the problem: Is it worthwhile, is the loss of efficiency caused by the modularity greater or less than the gain in efficiency due to the adaptability of the resulting product ?

And the problem of adaptability brings me to the next batch of problems, for which I hope we will find time to discuss this week, namely whether the large programs are such products which should be easily changeable by the user. I am inclined to say "no", and consider large programs as finished products, adaptablity of which is inbuilt into the product, and any change, which cannot be obtained by means of prescribed tuning of this program, should be out of question.

Another large set of problems is that of checking the correctness of large program. I think that the basic question here is not how to check the correctness but what the correctness of large program is. One could venture to say, even though I would not subscribe to it fully, let us agree that programs should work in, say, 95 % of cases. It is exactly like we set the acceptance standards for any other industry products. This may be not 95 %, may be 99.9 %, but it is always statistics. But as soon as you start requiring the full proof, you are running into serious troubles. And the question is following: Is it worthwhile ? Why should I prove that this program will work always, if never in my lifetime, or in computer lifetime, I will try all cases.

As I have said before, I am not expressing my own views, I am just setting the frames of reference for the discussion. And this is one of the possible points of view which you may find very inelegant to begin with, because it is much more elegant to require a mathematical proof of correctness.

And the last point I would like to bring to your attention is as follows: probably none of us has started his programming career by making large programs. Some of us have already graduated to the large program class, some of us are still in small or perhaps medium program class. Now there is a question that I would like to ask: As a good pupil in the class of small programs or medium-size programs, am I to succeed as a good student in the large program class ? Is an experience which we gain in making small programs of any assistance when we start making large programs ? Is it going to help us, or not ? And even though an automatic answer is: "yes, of course", I am not so sure after thinking

about this problem for a little while. This is because small programs and medium-size programs have nothing in common with large programs. This is, if I am good at making programs which are not changeable, then I am not gaining any experience in making programs, an essential feature of which is that they are going to be changed continuously. In making small programs I try to make something rounded, finished, closed, neat, no matter how much time does it take, whereas it seems that in making large programs, I should try to make something workable as soon as possible and then let it change, and change, and change. So all my working habits will be different: rather than striving for perfection, I should aim at adaptability.

By my second definition, all my working habits which I pick as a single programmer probably will be no good when I work in a team, because when I work as a single programmer I am my own boss and I have the overall view of the whole endeavour, whereas if we are working in a team, then I will have only a little piece to do and I may not even know where this piece fits into the large program.

MANACHER: You have talked about the large program and you stressed that even if their design modules are very exact it is a dangerous or uncomfortable policy to build them in such a way that the user can expect to adapt them beyond their natural expectation of tuning. On the other hand, you mentioned that big programs are constantly changing, and I should assume that your definition includes the case in which they are divided into modules, so that the modules also are changed. This means that the modules have the property that they are expected to be changed rather frequently by the person who build them, but very infrequently, or never, by the person that uses them, and this would seem to imply a certain bias that it is safe for their creator to change them, but very unsafe for the user to change them. I wonder whether perhaps there was not some apparent contradiction.

TURSKI: I do not know if there was a contradiction in it. What I really wanted to say is that no matter whether we change individual modules or a whole thing, it is only the author who has the special tools for changing not included into the large program itself. That is, you give out the product without a service kit, because in order to use a service kit you require a special training which you do not get as a user of this product. It has nothing to do with the modularity of programs.

RANDELL: Did you make technical distinctions, or purely political ?

TURSKI: Let us say it is technical. You make large programs to be used by almost anyone nowadays, and their users certainly do not have the expertise to be able to change them. We might try to make the programs in such a way that they would be easy for the user to change, but the real problem is: should we ? If we do, the user will not be able to resist the temptation to change because we gave it to him. This is politics of course, but there is also some purely technical reason. I am sure that it is easier and neater to make large programs in such a way that they do not contain tools for easy changing of them. So to your question, I would say it is both technical and political decision.

RANDELL: You see the technical reasons in not wishing the tools, which might help to change the program, to be included with the program, but you assume that they exist, because you assume that the maker is in a position to change the program. It seems to me that you have to design the program so that it would be possible to associate with it tools with which we hope to change it. And it is a political decision when you let those tools out of the front door with the program. There may be technical differences in the amount of the documentation, that you send out, there may be political distinctions depending on the amount of problems that you get. But I do not see that you made technical, real technical decision yet.

TURSKI: First of all I am not certain that it is proper to say that "there are tools for changing large programs". These may be only very primitive things and they may not be so closely connected with any particular large program, but they may be of much more universal nature. In the latter case I daresay that the user of a large program may not have the expertise to use those tools; whereas people who make large programs, well, they have to.

RANDELL: There is the assumption that your program is very clever and that your users are very stupid. I know cases when of necessity it would appear that the user becomes very clever because the program has been very stupid.

TURSKI: The point is taken.

MANACHER: I think there may be a scale, perhaps two senses of the modularity. In the first one you construct modules which are only supposed to interact with some reasonably small main program. There is enormous number of modules but it resembles a situation I think of very strong patriarch and hundred grandchildren, so that if one grandchild misbehaves it can be caught. In such a situation it is wise to construct the module which obeys certain discipline and if it did not obey that discipline it would be caught. In the other discipline, one starts with a certain assumption about the way in which modules fit together and all is based on the faith that each of them works according to this principle. I would think that the question of whether a module can be easily varied is closely connected to which one of these philosophies are the modules constructed according to. Do you agree that there is some scale of that sort ?

TURSKI: I am not sure if I quite got your point. You said that there are two types of modularity: one is essentially hierarchical and another one is just two-layer, viz. something which is very strong, your patriarch, and then not a children-grandchildren system, but just many little equally strong, equally valid modules.

MANACHER: Yes, and in one case the misbehaving module would be detected, so that one can write with relative freedom because whatever environment the module fits into will detect (hopefully) most or all mistakes in the module; and in the other, one is given a set of operating principles which the module had better obeyed, and when one programs, there is an article of faith that the module will obey this informally given restraint. If it does not obey that sort of restraint then one is in serious trouble. And I would maintain that there are different styles in programming if one happens to work under one case or another.

TURSKI: I still do not think I follow your argument. I think I can agree with these two styles of programming, but somehow I do not visualize the picture of circumstances catching the errors of modules.

MANACHER: There is a very simple illustration: If the module is an ordinary subroutine, it may be so that the call saves for you set of registers and does not care what you do with them. In the other case it may come to a convention that you are not to use certain registers without checking. That is a very simple example, but I think that conventions of that sort can be composed in high level and can be referred to; the demand, for instance, that certain common data structures maintain their integrity in the module, that if you tamper with it and the supervising program does not detect the change, then something bad may happen much later.

TURSKI: Thus it is not so much the hierarchical structure of programming versus more parallel one, but it is a problem of whether individual modules may harm each other. There is probably only one argument against the full-proof modules and that is they are very expensive to produce. Otherwise we should be using only the full-proof modules.

MANACHER: Yes, I think that there is a trade-off and I would propose that there is also a scale of possible trade-off.

TURSKI: This is an interesting point. It seems to be very highly interlinked with the problem of checking the correctness of the whole thing, because if the module is full-proof I can state that this module has only prescribed effects and has nothing except them. And if I can be sure that such statements hold true about each module in my system, then by application of almost mechanical reasoning I can prove statements about my whole system, if I know how modules are connected - which I should know.

LAVROV: May I say something about the definition of a large program ? It seems to me that there exist such things as programming laws; perhaps I cannot formulate them, but I think that they exist. And when we make a small program, then any deviation, any violation of these laws leads only to time-loss and - we can make small programs with some violations of these laws. But if we have a large program, then any violations of these laws make it impossible to get such a program going.

Therefore I propose to consider these programming laws and try to formulate them. In my opinion, they must be common for small and large programs.

Another point is the similarity between large programs and any large project, large undertaking. Of course there exist some other laws common for every large project, as well as for programs.

I think that there exists no controversy between modularity and effectiveness. The sound basis for dividing programs into modules is such that between modules there must be minimum data transfer. Only the global information should be transferred from one module to another. If this division is made well then we shall not have any big loss in efficiency. Modularity is a principle which is the basis of any large system, technical, logical or social system, and I think that this principle must be applied to programming, too.

I do not think that the user would want to change anything in a well elaborated large program. If he does want to, then it means that this program was badly designed, badly constructed. May I use some example. There exists a toy for children, the MECHANO set. Building various models from different parts does not mean that we change this toy, since it always consists of the same parts. We only try various combinations of these parts, and if the user needs some modification then it must be only superficial changes, not essential ones in the program itself. The program should be designed so as to allow these modifications or changes, but there must be no changes in the program itself.

TURSKI: It seems to me though, that there is a slight gap in your reasoning. In one instance you are saying that there should not be much loss of efficiency because we should limit the transfer of information between the modules of a program. That means - and I agree with you entirely - that we have to transmit only the very global information. But if we do so, it will not mean that there might not be quite a lot of it to be transmitted; you may transmit relatively small amount of information only if the modules are large. Then they contain enough internal, local information to make processing possible, and then perhaps only the status words have to be transmitted between the modules. But now I am coming to the second point of yours, namely that you do not want to change the modules in order to change the program, that you should like to have such a set of modules out of which you would like to be able to build any program, within certain class of programs, of course. But if you really wish to construct programs from modules and be reasonably flexible, you will almost certainly find that the size of modules is pretty small; if the average size of the system modules is large, the system is not flexible. You see, these little bits in the MECHANO toys are really very small bits, and you have to assemble a lot of them to get a workable toy; so the total amount of information which flows between the modules is rather large. I do think that there should be a trade-off between modularity and effectiveness.

RANDELL: It depends. Sometimes the modularity itself may be a component of the effectiveness. To sell a program to somebody - you have to meet certain of his desires. He might be somebody who just wishes to program only by putting numbers into the front end and get answers at the back end, or he might wish to use the system almost as a MECHANO set to go to build his own programs. To him, therefore, modularity will be something that he needs, something that he will pay for. You have the different types of customers and because of that you cannot think of modularity as being completely orthogonal to the effectiveness.

TURSKI: I think I mentioned two types of effectiveness, and I agree that one sense of effectiveness of programs is just modularity, adaptability of programs. But now I am talking about the other aspect of effectiveness, that is to limit as much as possible the circulation of information within the system which does no good to the user. A silly and simple example - all the storing of information which takes place after interrupts is useless, it does not do any good to the user directly, it is necessary only to run the system. A user could not care less whether there is any storing of information on the interrupt or not, it is there only because a system is built in such a way that you have to interrupt and you have to store information.

RANDELL: Let us just worry about the storing of registers on interrupt. You can imagine two people offering their systems to the user. One has just one set of registers, and this brings us to the business of storing and restoring. The other has two sets of registers which he uses very inefficiently because in any period of time only one of them will actually be used. The user should not care which of those inside solutions is in fact chosen, what should matter to him is what the overall result is, and I do not think you can describe either of those pieces of mechanism as being of no importance. They have not been put there so that they do absolutely nothing, they take part in the total function. If they have been designed incorrectly, in the relation to all

the other things that are going on, then you cannot really say "that is useless to the user", and "that is the only thing he wishes to pay for". It is the same with an operating system: very often you will say "that much is overhead and that much is useful work". They try to make a logical division between useful work and overhead, which is virtually impossible. Some is more clearly useless than the other.

I have to agree completely with what Prof. Lavrov said originally about modularity, if I understood him correctly. This was that if you have modularity which fits the actual structure of your program, then this is not something that has a definite cost. It certainly has a value. If however you impose an arbitrary, a a ridiculous modularity so that in fact you are transferring vast amounts of data across an interface, when in fact there should not have been any interface, then this is clearly bad and it is not that modularity is bad, but stupid use of modularity is bad.

TURSKI: If you think about building large program as a sort of industrial production, you will imagine that you just have warehouse full of modules, you take them from the shelves, glue them together and you get your system. Roughly speaking, you may say that these modules come in different size. The bigger the size of modules (I do not mean really in terms of code-lines in it, but - in the same vague sense as I defined large programs), clearly the smaller the amount of unnecessary transfer of information. If they were of the size of full large programs you will have no stupid data transfer at all. In general - the larger are the modules - the less stupid information transfer you have in your system. But, by the same token, the larger they are, the more difficult they are to build and combine in useful programs. It is only when they are really small you may take staple modules and make out of them a useful program, because they fit neatly into any configuration.

RANDELL: I think I would better comment on this: you said "the more difficult they are to fit in the useful combinations". I think that more often it is: "the more limited is the range of things you can program within a given degree of facility".

TURSKI: Sure, quite true.

RANDELL: And as long as the range that you get from the set of modules is one which is of sufficient value to you compared to the cost of the modules, you have restricted area of interest. Suppose that all you wish to do is matrix-algebra, then as long as we have got the various standard matrix operations who cares about the differential equations ?

MANACHER: I would like to suggest a possible definition for good modularity. The separation of a program into modules is good when either the transfer of control from one module to another is trivial, or otherwise the transfer of data is trivial. But if it is not the case, then that is bad modularity. The other case is, for instance, an allocator which might be a part of a bigger program. The allocator is maintained an enormous pool of data, but it does so privately, and its job is to peel of a little bit of data and to send it on. On the other hand, control is taken from the allocator to the main program very often.

TURSKI: Coming back to Mr. Randell's remarks, I agree that if you stick to a prescribed size of modules, then it is a range of useful things you can build out of them which really suffers when the modules are big. Now, I am looking at production of large programs not from the user's, but from the producer's point of view. I am interested in making as many large programs as possible, so in having as universal set of modules as possible. This forces me to think rather in terms of small modules because I would have to have an enormous number of large modules to be able to satisfy the needs of different applications fields. But if I go for small modules then I have the information transfer problem in my lap. And I cannot do anything about it, so there has to be a trade-off.

RANDELL: Is there anything like a limiting case on small modules ?

TURSKI: I would say - yes, a single instruction and then I am quite universal at this level, except that I do not transfer any information from one instruction to another one. So an instruction-sized module is some really ugly thing which has one instruction and a memory dump with it. It is like imagining the supervisor running this way: it picks up an instruction, restores the memory, performs this instruction and performs another dump of memory and goes to the next instruction to restore the memory, and dump it

again. This is a limiting case of a small module with full transformation transfer, which is obviously rather inefficient.

RANDELL: I can give you another possible definition of a good module. A good module is when it is much easier to describe what it does than how it does it.

TURSKI: You go for the modularity imposed by the modularity of the problem to be solved, so that the structure of program mirrors the structure of the problem. This is a very nice approach, except that it does no good whatsoever to the software-maker, because when the modules reflect the grains of real world problems they are just unique for a given problem and you cannot re-use them to make other large programs.

MANACHER: I think that problems with large programs are difficult enough just in terms of getting out one program to do one job; if you want to use modularity as a tool that is wonderful, but perhaps it is pushing is too far to ask that the modules should be re-usable products.

RANDELL: I think it gets us to a slightly different point which might be appropriate now. One reason that the program can be regarded as large, where you are using the word "large" almost like "complex", or "difficult" or "horrible", is the question of the program being not meant for one particular user, in one particular environment, but being meant to satisfy lots of different users in lots of different environments. That can make programming much more difficult. The operating systems produced by a large computer manufacturer provide facilities for a very big range of users, working on lots of different types of machine or lots of different configurations of a machine. That is a much more difficult thing to produce than the specific operating system, working for a known single user on a known single computer installation.

KUROCHKIN: Some years ago, ALGOL 60 compiler was considered as a large program, today it is not so. It may be a little exaggerated, it may be a little rude, but I think that there is no "large" program at all. I think that some people make large programs, they make the programs large, because they are not experienced enough, they try to do the job that is not for them, they are not trained enough to fulfill condition of their job, they are still learning at doing something. The programs for general use should not be made by people who do not have experience in making such programs.

TURSKI: If I may object to your proof of non-existence of large programs. I would say that according to both my definitions ALGOL 60 compiler was not a large program, even five years ago, because it could be made single-handed, it has been done so, and after being debugged, it never changed again. So it lacks, and always lacked, those two essential characteristics which I attribute to the notion of large program.

FIALKOWSKI: I think that a large program is the one which we do not expect to be error-free and this is the main point. In all our classical experience of programmers we are expecting of our programs to be error-free. But this is some distinct reminiscence of Laplace conventions and mechanistic meanings. Now, having very complicated problems we do not expect our programs to be completely error-free. But I wish to point out that I do not think that the best way is to apply simple statistical methods for all those non-hundred percent error-free programs.

MANACHER: I agree in general distinction between small and large programs, but I think I would put it in a little less mechanistic way: for small programs, if one looks at a number of bugs discovered, let us say every month, and one plots it, one sees a time-constant and one can say that there is a period in which presumably almost all the bugs will disappear because there is an apparent convergence. With the large programs, one expects to happy use them during the period in which the rate of convergence of the debugging process is not good.

TURSKI: Am I to understand your comment as follows: small programs have finite decay time whereas large programs have infinite decay time ?

MANACHER: No, not infinite but just very long. In other words, with a small program, one expects to have a certain bearable period which is perhaps even long compared to the decay time, whereas one cannot tolerate in a large program waiting until several decay time have ocurred because it is too long. You have to tolerate the use of such a program in a period of time which is short compared to several decay times.

TURSKI: We are coming to definition of large programs in a very circular fashion. All the definitions we have heard so far are not concerned with programs as such, but with their properties, or properties of their making. Are we not talking about so badly defined concept that no direct definition can be given ?

MANACHER: Perhaps it is possible to define a large program in terms of its environment. If I take a compiler and feed to it some task, that task really contains an exact prescription of what is to happen to it. It is a message, an input tape to an algorithm. I suspect that one of the things that characterize most of the large programs that people have in mind is that they are connected with the maintenance, or control, or interaction with some environment which is not an exact encoding of a form acceptable to some automaton. I agree that some sort of statisitcal measure is a good way to measure such programs; the reason it sounds good for big programs and bad for compilers is, I think, that 95 % may represent the degree of approximation to the interaction that we wish a large program to have, but if we expect the program (like a compiler) to react exactly to some input, then 95 % is not tolerable.

RANDELL: This is to do with robustness of programs, and since you mentioned the ALGOL compiler there is some robustness in that, because it uses a very simple, yet very effective programming trick technique. It always takes positive decisions. If somewhere in program you are prepared to deal with possibility that the value of x is zero or one, you check whether it is zero, if so, you do what is necessary, alternatively you check this one and then do what is necessary. And if it is neither zero nor one, something which probably you will convince yourself could never happen, you print an error message; there you have the case of deliberate redundance by very simple programming-trick, which can easily be taught to any programmer.

LAVROV: One more definition: A program is large if one cannot say without an error how large it is.

STOLIAROV: The difficulty in defining large programs seems to be caused by the fact that at the moment we are in the border zone between small programs and large programs, we have difficulty in deciding whether a compiler is a large program or a small program, and perhaps when we cross this border zone and enter deeply into the territory of large programs, that is programs which cannot be made not only by one or two, or even by ten people and require larger teams to work, it will become much more clear what is a large program but then we shall need not so much a definition of a large program as a recipe how to make them. That is an important part; that is to decide how to produce large programs when we know for sure that the product we are making is a large program. When we enter deep in large program making we shall discover that there is no question of modularity versus monolithic approach because large programs will be necessary modnlar since if they are made by large teams, there is no use waiting until someone consolidates the work of so many people and we shall be forced to use the capsules, the modules made by individual people as a single system of modules, that is a large program.

MANACHER: I would like to suggest that there is one experience with a large program and that is OS 360. Every two months there is a special bulletin published by IBM. One section of the bulletin is called "unresolved errors"; every error of that kind is known to exist and it is known how to avoid it, but it is not known where in system the mistake's whereabout is, we have only a prescription to get around it. And the number of such errors in each edition is about two thousand and very often they carry over from one issue to another, and, as far as I know, in the last four years the number has been about two thousand and that is the reason why I said the time constant is very large.

RANDELL: First of all, I would like to agree with STOLIAROV. At present, there exists a class of problems which can only be programmed by very huge number of people; such very large teams are eventually capable of producing programs for problems, which at the moment, could not be possibly programmed by one or two or three or four people. Some people will argue with that. I happen to agree with what I think you imply.

Now I come to the OS 360; there are many discussions as to why the known error rate of the OS 360 stays at the two thousand. One suggestion is that since there is a very large number of facilities which are due to be added to the system, they are added at a rate which keeps the level of errors at two thousand; people have learned to tolerate two thousand errors. When OS 360 was first released, it had four hundred thousand instructions, it has now,

some four-five years later, two million instructions. Therefore it is not just the question of minor changes, it has expanded by factor of five.

TURSKI: If it keeps all the time two thousand errors, it means that when it was released first, one instruction every two hundred were wrong and at present it means that only one instruction every two thousand is wrong. It is quite an improvement. May I ask you what are the reasons for which you think that there might be problems for which, in principle, you do not see at the moment any other solution except the army-of-ants approach.

RANDELL: Because a number of different facets to the problem is too large to be comprehended by one or small number of people. As later we learn more about how to put programs together then I think we will not try to build huge programs all at once. I think we will gradually go to adding more and more programs together, where, hopefully, different people understand different parts of the program and where the total effect seems to be tolerable.

STOLIAROV: We do not have to worry about the definition of large programs, we can sit and wait until they die of internal causes.

RANDELL: One is trying to get something tolerable ! The best thing one can say of OS 360 is that a large number of people have learned to tolerate it.

MADEY: In the class paper prepared by Dr. Turski the term "large program" is used. It is used almost in every part, but, at the same time, the definition of the large program is - well, I would not say it, it is not a definition at all, but nevertheless, this is a different understanding of the definition. We have been given two definitions by Dr. Turski, and we have heard some other proposals, like "there are no large programs at all". I would like to suggest that we should not concentrate on the programs and we should not try to define a large program, but we should try to define a class of problems which we want to solve. I think that the features pointed at in definition number one (that the program is never finished) are not program features, but features of the problem which we want to solve. Thus we should distinguish, first, a class of problems we want to discuss; secondly, a language, and thirdly, a program written in the language for a problem from the class. It means to discuss these things not mixed together, but quite separately. This is the first suggestion.

The second one: the word "large" is very misleading; using it I always think about the huge amount of paper with something written on it. We could simply forget the word "large" talking about a program written in the language to solve a problem from a certain class of problems. This class has the following feature: it includes the problems defined by sets of attributes which change in the time. It means that in the time, let us call it t_o, we know some attributes of a problem and so at that instant we write in a fixed language a program to solve this problem. But while we are writing this program, the time is passing and at the time t_1 we get some new information about this problem, partially because we have results from our program, partially because we learn something more about the subject. And so we go on, producing in fact new programs which solve partially the problem which is still changing.

With this understanding we can now answer the question about ALGOL compiler which was written in 1960. In 1960 ALGOL 60 was not yet well defined. So the description of the language still changed a little bit. And secondly, if we write a compiler, we will learn something about language itself. From this point of view, in 1960 to 1963-65 the problem of writing ALGOL 60 compiler would be a problem of our class of problems.

I would suggest to call this class of problems: non-deterministic problems. Whether this class is empty or not, depends on various things. It might be not empty at a certain time, some problem may belong to this class and later on it may not. The final conclusion is: instead of trying to define a large program, let us stop using the word "large". It is a program which in fact tries to solve certain type of problems.

TURSKI: Is it not true of any sort of problem that it always belongs to a class of problems ?

MADEY: It always happens so that we have a class of problems, a language and a program, but this what you would like to call a "large program" is a program written for special problems which we should call somehow.

TURSKI: Are you saying that it makes no sense to speak about large programs, and we should speak about large problems instead?

MADEY: I do not like the word "large problem" either. It would be better to talk about the special class of problems and how to solve them, how to attack them, how to start this and then to come to "program" at a certain time.

RANDELL: In one sense I agree, and in another I disagree. Certainly, it seems more attractive to worry about the problem that you have to solve, because you might say that one person thinks this is difficult, and another knows it is simple. One should not therefore treat this as being a case of a large - in Turski's sense - program. Rather you treated it as being a case of an ignorant programmer. On the other hand, in the real world, people who have problems to solve, have human programmers, and the method that they use to solve the problems must take into account the skills and knowledge of the programmers that they have. And although, ideally, everybody will be absolutely intelligent and experienced, the same problem, not only in the different times, but also posed in different circumstances to different people, will give rise to quite different programs.

MADEY: Yes, I do agree. We try to define something called a "large program", and the definition says that this program is never finished, changes in time. What does it mean "it changes" ? It means that in the next time-step we get some extra information about the problem. If you know some more about the problem and so you write another program, or change your program, you get another program but at every time you have a program for the problem. It can, of course, contain errors.

RANDELL: There are some people who worked for many years in the same problem area. And they have produced many different versions of a program, which essentially is trying to do the same thing; each time it does it a little bit better. They find out more about the problem as they do some more mathematical analysis, they get a better computer and so gradually they make whole seqeence of programs. But in fact the programs are small, they can throw them away and start again. An important thing that Madey did not say is the fact that sometimes it is impractical to start entirely again from the beginning. And, instead, you have to make use of what you have got and improve on that,as a program, and there the largeness comes in. It is really a point of practicability.

MADEY: But it depends on the problem which we want to solve, not on a program, the feature in question is that of the size of the problem.

RANDELL: You cannot step around the fact that the size, relative to our present understanding, is important here. Changeability is certainly something, but not all.

MADEY: Yes, I agree.

MANACHER: In the example given, ALGOL 60 compiler, one should observe that the specifications are changing, but they are changing because one wishes them to change in that case for the most part. Perhaps there were mistakes in the ALGOL 60 Report, but I think for the most part the changes that one has in mind are definite improvements. So I think, one can say that there is a specification in 1960 and another specification in 1965 and there is simply a series of non-large programs, each one for a new set of specifications.

I should like to know whether there is anyone here prepared to give an example of a program he believes to be a large one, which in any sense at all does not contain the real-time interaction.

LUKASZEWICZ: For instance a PL/1 compiler.

MANACHER: In my inner, intuitive understanding this would not be a large program.

ŁUKASZEWICZ: So even complicated program with, let us say, hundred thousand instructions could not be a large program ?

TURSKI: Is it necessary for PL/1 compiler to contain as much as hundred thousand instructions, I do not think so.

KUROCHKIN: It is. It just contains so much.

TURSKI: It does not prove that it is necessary. I have seen ALGOL compilers containing several thousand instructions which are obviously not necessary.

LUKASZEWICZ: I have no figures about PL/1 compiler, but for the FACT language, one of the first data processing languages it was

claimed that it had more than two hundred thousand instructions. I would say that even today it is a very large program. If you deny that, you go very far from the meaning of the word "large". Perhaps, if you like to set the meaning of "large", it is the number of instructions, and degree of complication, interrelation between parts of programs that we must somehow consider.

MANACHER: We have to consider what was the rate of convergence. This goes back to the point we made before about the convergence of the large programs. What was the rate of convergence,number of bugs found in FACT system. My guess is that it would have been moderately rapid, just purely guessing.

TURSKI: If a language is changing every so often, you do have the real-time interaction. It is not necessary for real-time that we have measuring gadgets or things like that, there may be real intellectual input.

RANDELL: I think my comment to MANACHER's is: a program does not have to have a real-time constraints in order to be "large" but it helps.

TURSKI: Would you say it as strongly as that: if there is a real-time interaction then program is necessarily large ?

MANACHER: No. For instance, there are machine tool control programs which are surely not "large".

KUROCHKIN: May I ask you to write down on the blackboard some problems that can result in large programs. I am interested how much of them can be actually written now.

After some discussion, following list has been drawn up on the blackboard. The last item caused considerable controversy but was eventually included because some participants insisted that there might be many versions of sine subroutine developed over prolonged periods of time.

1. Space flight support.
2. Military.
3. Seat reservation system.
4. Air traffic control.
5. Inventory control.
6. Centralized on-line banking system.

7. OS 360.
8. PL/1 compiler (optimizing).
9. Sine subroutine !!!

KUROCHKIN: Almost all these examples are big because of the big amount of information and programs themselves are not large; for example, a seat reservation system or centralized on-line banking system.

RANDELL: A seat reservation system usually is not just one which keeps a list: how many tickets have been sold for each airplane. Normally, it contains a very large number of programs which would do many different things. It so happens that they are all based on the fact that there exists a file indicating who is going to fly, on what plane, and when. Once you have that information, then you can, as airlines do, write a large number of extra programs and all these fit together, sort of, and become a very huge software system.

KUROCHKIN: But these separate programs are connected only with the help of this file and not by themselves.

RANDELL: But the degree of connection, even if it is nearly only through the file, can still be very strong, because the extent to which one program is dependent on what another program has done to the file.

MANACHER: I agree with Prof. Kurochkin. Suppose you have a banking net system which, in principle, is very easy to describe. It only does very few things, and even though it maintains a very large file it came into our list. If we make a library maintenance system, for instance, and you only require that it does few very simple things, but for many people, I think it should belong to the list.

TURSKI: The library maintenance program becomes large not because of the large volume of data, but because of the large traffic in data, and that is quite different thing.

MANACHER: May be there are many dimensions. I am not presuming much traffic or much data. I am only saying we should discuss whether programs with with very much data but not much traffic, with very much traffic but not much data and so on belong to our list. These are the dimensions, possibilities, and it is worth dis-

cussing each one of these to see whether one dimension alone will allow a problem to be classified as large.

RANDELL: If a file becomes very large indeed, say about several orders of magnitude greater than some base file, then this size itself brings in a lot of extra problems. Somebody mentioned a library. A program which would enable us to keep a catalogue of the books in our computing laboratory library would really be very simple. We are trying to write one of those for the whole university library, with perhaps some million books or something. The difference in scale means that it is now essential to deal with problems that you can ignore in the small case. With one million books the problem of getting all catalogue entries correct becomes immense, while when you only got a thousand or two thousand then you punch some cards and you put them in a right place. In this way a change of scale becomes almost a change of type.

TURSKI: In a small library many decisions can be taken statically, a priori, and do not have to be evaluated; a decision making process does not have to be repeated during the run-time of that program. In another case, the situation changes. When we go on to a large library, many decisions have to be evaluated, made on the basis of dynamically changing circumstances, therefore, you have to put separate pieces of program to do those things which otherwise you do just by hand.

KUROCHKIN: I do not think that big amount of homogeneous information makes the program big. There must be a measure of complexity of information and when it exceeds some degree, when the complexity is too high, then the problem can be called large. For example, the problem of linear programming, programming that deals with some million of integers. It is very difficult to solve such a problem. There are many iterations, many difficulties, but this problem can not be termed large. The information is too homogeneous. This cannot be said about, for example, OS 360; it consists of many kinds of programs and types of information, the connection between them is quite complex. I do not think one could list many problems that would require large program. In space flight support there are many different programs very slightly connected, so I would not call this a large program. From our list I think only OS 360 is really large.

MANACHER: I had the opportunity to talk to one of the software designers on the Apollo project, so I can speak from that point of view. I think Prof. Kurochkin is absolutely right. The design philosophy of all the modules of the space flight Apollo program, perhaps with the exception of launch control, I have to admit that, wants to stay away from the difficulties that we associate with large programs, and it was possible to do this through many different means, first of all by having some partial controls. I do not want to take the time to explain it, but it is certainly true that the project, except the launch control, has been divided into many modules, and the reason for the size of programs in space flight support is not because it is doing something complicated to control but mainly because it is the program which translates the telemeter data into a highly human-readable form and which does so quite flexibly. I think it is designed with a degree of simplicity in mind which in some way keeps almost all of it from belonging to "large program". But not quite, because to present so much data so quickly, you keep the system fluid. But the lack of control and direct real-time interaction keeps it away from being large.

RANDELL: I think it comes to that: at the moment there are certain problem areas which people have tried to produce programs for, using large number of people, and with great difficulty they have sometimes succeeded, sometimes failed. These are problem areas which as yet nobody has succeeded, or, sometimes even attempted, to program with just two or three people. As the problem areas get more understood, perhaps they will be, in the future, done with by two or three people. But there do exist programs which have very big teams of people working on, and it is not very believable to say: "Ah, but that could have been done by just two people"; one might, however, say: "It should have never been done at all".

LAVROV: The main goal of our discussion is to understand why programming at all is a difficult thing, what is the nature of the difficulties, and how one can solve them.

HOARE: Can I ask you whether any member of this meeting is currently engaged, or has successfully or unsuccessfully been in the past engaged in design or implementation of a large program according to any definition ?

I should start with myself. I have not been engaged in any large programs, but if Madey's definition is accepted, then I did: I was engaged in an ALGOL compiler in 1960. But really, I think this is the case where it turned out not to be a large program, because we cheated, we avoided little problems that would have lead it to be a large program. And I think perhaps this conference would be more interested in programs that failed to be large, due to some skill or good fortune on the part of their designer and implementers more than in the large problems, which suceeded in being large programs.

RANDELL: Normally when people talk about large programs, one of the problems they believe exists is that of saying how long it would take until the program is usable, or of how to produce the program in time for some deadline. When the program is very large this must surely be an almost impossible task.

Most people would think of one space-flight support program, the Apollo project, as a very large program. The number of people involved is about three hundred programmers, and the number of instructions that they produced - certainly many hundreds of thousands; by these measures people would say "that is a large program" and "that would be a program very difficult to complete in time for the schedule of the launch of the rocket". Apparently, in project Apollo they did complete the programming in time for its launch. One of the reasons you find behind this is that, as HOARE said, they sort of partly avoided the problem because you would find that they did not have a fixed set of things that had to be working by the time of the launch. If something was working - good, they put it in; if it was not - they left it out. Only a very small central core of the program had to be working. All of the rest was such that if absolutely necessary they would discard it.There is a central part concerned with obtaining the basic telemetery data, but nearly all of the amount of programming that they do, involves processing-up data and translating it, and presenting it in such forms that people can use it, and can sit at different desks and inspect it. A lot of that data, if necessary, can be ignored. The flight would then not be so valuable, they would not get so much information from it, but they would at least get the men back.

HOARE: I think that this echoes a theoretical conclusion which I came to in thinking about the designing large systems, particularly those which are intended to be robust and withstand certain types of failure. It seems to me that it is essential to program and get working the reversionary mode of operation before you program the main application. The reversionary mode of operation to which you revert when your equipment (or your software) fails, that is to say ...

TURSKI: help procedures -

HOARE: I do not think so. An air traffic control system must continue to work even when a great deal of its equipment is down; obviously it will work at reduced efficiency, and probably require a certain amount of operator's intervention. This is the place where I must begin, get this working first for two reasons - one, the main one, is that unless you get your reversionary mode working solidly, it will never work, you do not use it frequently enough to debugg it from practice; secondly, unless you know exactly what your reversionary mode of operation is, when you are developing the extensions, you will never get successful transition from full working to the reversionary mode. In the Apollo project they have a small core of working programs to which they can revert when all other programs are not working, and that is how they succeeded.

RANDELL: Another point there is that if you start off worrying about - what you called - reversionary mode, and you are thinking then in very pessimistic terms about all things that possibly can go wrong, then you sometimes can be quite realistic. For example, some of the earliest airline seat-reservation systems have one rather amusing little reversionary mode: if the girl sitting at the seat reservation console, who will normally give you your ticket, wants to find out whether there are any spare seats on the flight, she will type the number of the flight and then will get a message back, saying "yes, there is space", or "no, there is no space". The reversionary mode for first SABRE airline reservation system was one which when the system was not working always sent the answer: "yes", on the principle that, on the average, there would be in fact a seat. When the system was put back to work, say, half an hour or some time later, then it would have

enough data to know to whom it should send messages of apology to. It was effective. In fact, you will find that this method of selling tickets had very much in common with the technique which had been used before they had the computer system, so it was really quite realistic. Up to a certain limit they would assume that there would be enough space, even if later at the book-keeping, every night or every week, they might find that they were overbooked.

MANACHER: A bank discovers the necessity of abstracting its files in quite a number of ways, this is not done every day, but it wants to have the facility. Therefore it gets a statistician to do the work, to discover - let us say - twenty rather complicated reductions of the data in the file, each of these being independent. The bank wants the job to be done in one year, so it gives the job to some programming house, they make the observation that programming each way of reduction takes five people, a wonderful-sized team, as we all know. There are twenty jobs of roughly equal magnitude, it therefore gets one hundred people to work for one year to meet the deadline of having all of these twenty modules done in one year. This is not a large program. Reason: The interaction between the modules is too weak. Conclusion: Having a large team of people working on a project to meet the deadline is not sufficient condition to qualify the effort as a large program.

RANDELL: HOARE mentioned that some years ago he did an ALGOL compiler which at least started off as a large program. I similarly did an ALGOL compiler which I now do not think of as a large program. But one point about this compiler was that almost all of it had to be working before you could even run the program which just said: <u>begin</u> - <u>end</u>.

HOARE: My experience is different. I adopted a completely different structure of the compiler with the result that the simple parts of it could work before the rest. The concept of reversionary mode does not seem to have much relevance to a compiler, but on the other hand, perhaps, the routine that recovers from a syntax error is one that you should get working early. This we failed to do in our compiler, and had some difficulties afterwards, superimposing on the system a method of detecting more than one error in a given program. If we had given the amount of thought that the subject demands at an earlier stage, we would,no doubt, had written the syntactic error recovery process as one of the earliest parts of the compiler. Just think, how lovely would that be: you would be able to translate any program what-so-ever, even with very partial compiler; on any construction which is not recognized, we would just call "reversionary mode skip to semicolon".

KUROCHKIN: I do not think that to make an ALGOL compiler is a very simple thing, even now when we have much experience in it. In the course of running the ALGOL compiler there arise various types of errors, and you find out that you did not foresee all the conveniences required by the users. For example, we can ask you to add semantic recovery, or string manipulations, and to insert these new features in an already existing ALGOL compiler may be very complicated task, especially if you think of your ALGOL compiler as of a single program, not consisting of modules. Any program that is not made for one use, that is which runs continuously during large period of time, always will require some changes. According to TURSKI it will be a large program.So is ALGOL compiler too.

HOARE: A funny thing about programming languages: if you take ALGOL W for example, this is coming to be quite widely used in various scattered places, and there is very little demand for further extensions and as I know, very few errors have been discovered. For a language like PL/1 which is really quite large the demands for extension are very strong indeed.

MANACHER: I think that if you have great many demands for change, then in a sense you do have this funny real-time interaction: demand for evolution. Yet, it seems to me at the same time rather synthetic to use that as criteria for largeness, because it ought to be inherent in a program as an effort. In other words, if you describe the process of programming something, you should not be able to say that it goes from small to large by suddenly discovering something that you do not like about it. It seems queer somehow. You are using an external criterion to try to specify what ought to be an internal one.

RANDELL: A lot of us will agree that ALGOL compiler five years ago was large, or complex, or difficult, and that now it is not regarded as a large program. That is because a lot of people have written ALGOL compilers and a lot of experience is available. Different techniques have been tried and some of them were found bet-

ter than others. What has been learnt about ALGOL compilers is not very useful for much else, apart from some other compilers with features in common with ALGOL. We have made a progress in one particular area, which was a large problem area, in understanding that particular problem.

HOARE: It reminds me of the Rome conference in which many people tried to explain how successfully they could manage programming project. But the real problem which makes the programming of this sort things so difficult is simply that the people engaged on it - do not know what they want to do, and if they did, they would not know how to do it.If you start a project in such a state of ignorance, you might have problems, and there is no management, logical, and programming technique which will ever, in general, tell what it is you want to do and how to do it. It can only be discovered in the hard way, by tough investigations and gaining an insight into the problems concerned. So, we do not have large programming problems, we have large air traffic control problems. The problem, and the solution to it, is a matter of better understanding of the application, and not of the problems of programming.

LAVROV: I want to say why I consider the sine subroutine as a large program, or it would be better to say, as a large problem. This example makes it clear that we should not think about a program but actually about the problem. May I recall the main approaches taken to this problem. First of all, the pure mathematical approach, that the sine function is an infinite power series, is not going to help you in writing the subroutine. The first practical approach was to take a section of this infinite series as the tool for preparation of sine function. After that, it was recognized that one can use Chebyshev polynomials. Yet another approach was used later, for example: the rational approximation to sine function. Then comes such point as various subroutines: some fast and not very precise and others more precise but slow. That is another aspect of a big program. One more point is that in an assembly language we may call this subroutine by its number,later it was recognized that it would be better to write "sine" instead of a number. Then, one can mention such question as whether this subroutine should be re-entrant or not in a software system. I think that this subroutine still is and will be a test for determining whether some software system is well-designed or not. All this I consider as supporting the idea that one should not consider the problem of writing specific program, but one of solving the specific problem. Any one of the problems, which may look as a very small and very simple, in fact interacts with many other problems, and should be considered every time when we try to solve some new associated problem.

HOARE: Is the sine subroutine the largest program which you have been concerned with the implementation of ?

LAVROV: No, it is not. In fact, I have had not much to do with this problem myself, but I know that many improvements were made in the approach to this problem.

STOLIAROV: I have little experience in large programs. I took part in making a 1 MB size system. Even though we clearly understood what we were supposed to do, we were supposed to make a software system for second generation computers, including ALGOL, FORTRAN and COBOL compilers and small batch processing operating system, nevertheless we were faced with technical problems of putting this software together, because the system was big. As an example, I may mention the difficulties in changing parts of software to mirror the changes in hardware specification. Obviously, when we start to work on a software system, even without assuming at the beginning that it is going to be a large system, as the application of the computing system progresses successfully, we are gradually passing from a piece-wise processing to a complex processing. On the top of our initial core of the programming system we are getting more and more capsules, more and more programs, each of which may be quite simple. For instance, if in the complex problems of industrial control, which were first conceived as individual problems, but which possessed, to begin with, a common data file, we change the basic data structure, then we are forced to introduce changes into many dozens of individual programs, Therefore, I think that there are very few cases in which we can talk about the homogeneity of systems consisting of individual parts. As a general rule, when the total volume increases, in order to preserve the efficiency we have to complicate the structure. Every good software system is a large

program, with possible exception of some well-known examples. When we start designing a large software system, first of all we have to make the hard-core software, using which we can build whole system, or we produce special software for software making. But we should not pay too much attention to the division between the software tools used in the production of software and the software itself; in every fairly-sized software system there should be a small control part, the purpose of which is exactly the same as the purpose of the whole software.

HOARE: I think that the essential feature of programming, which makes it so difficult, is the fact that it consists almost wholly of making decisions. Decisions right at the top about the overall, global features, about the structure and modularity, then the detailed decisions on which instructions to use, which bits to use in representing data, what action to take in every possible circumstance that could ever arise in practice. And these decisions must be taken, perhaps ten, or fifteen times a day, by each person engaged in the project; it is difficult to count them, but in a large program there must be many millions of decisions to do something one way or to do it another way. And every one of them must contribute towards the overall objectives of the system, which, of course, are unclear throughout most of the time when the decisions have to be taken. In any organization where so many decisions have to be taken, you would have problems very similar to programming problems.

ŁUKASZEWICZ: I think that no definition of a large program is possible, because we have many different approaches to what a large program is. Personally, I like the definition that a large program is a complicated program which must be written by a team of people, one person is not sufficient, and then there are problems of communication between these people. I think that big problems are: how to divide the work of writing the whole program into small parts, and how to communicate. And I expected some answers to these problems from this conference.

SADOWSKI: I worked in a team which made an ALGOL 60 compiler for URAL-2 computer. It was a four people team, therefore the work was split into four parts. When we were trying to join these pieces together, we had considerable problems because the way of communication between individual parts was not designed properly. That is why I think that the really important big problem in the large programs development, parts of which are being developed by different people, is getting those things into a single workable program.

HOARE: After the ALGOL 60 project was reasonably successful, I took part in the design of an operating system for a fast, fairly small computer which had many of the features of what was later known as third generation software. This project grew in size, and towards the end included some thirty people. At that time I was the manager of the project. After a while it became apparent that the project is not going to succeed, certainly not in any foreseeable time-scale, so I was concerned with trying to rescue something out of the wreckage of it. The project had been going for two years, there seemed to be no end to it, so I tried to cut down specifications so that we would have something to deliver to the customers within a period of six months or so. It was a difficult time, the people were not very experienced, there was no proper management tree, no experience in estimation of the time that the programming would take and, virtually speaking, we had built up from the ground an organization, a framework with reasonably proper delegation of responsibility. After six months, we actually did begin again to deliver programs, and everybody worked very hard, and the first of the suite of new programs was ready for the delivery. It was very slow. It was, in fact, the second version of the ALGOL compiler, and whereas the first version was capable of compiling ALGOL at the speed of between five hundred and a thousand characters per second, new version was compiling at the speed of about two characters per second. So although it was possible to deliver this as a working program, it was too difficult to persuade the customers to accept it. As the result, a very considerable amount of additional work was expanded to see why it was so slow, and what measures should be done to improve it. It was very difficult, but after few months we managed to speed up the compiler by a factor of four, to eight characters per second. After that, we decided to cancel the project as a whole, and the grand-design into which we have put an amount of effort which perhaps is small by present standards but which at that time we considered to be very large (per-

haps between thirty and forty man-years), was written off completely; there was nothing that could be salvaged from this project, and so we started again. Gradually, using some of the management skills we had to develop during the difficult phase of the previous project, we managed to set ourselves modest targets which were achieved, mostly, on very short time scales, delivering each improvement as it was ready to our angry customers. In the end I think the experience led to very much improved design of software. The company decided to concentrate on the scientific computer market, small computers in universities and scientific establishments, and for this market, by limiting its objectives, it could produce quite good working compilers and operating systems. From this disasterous episode the reputation of the software team did recover and the products are now in successful use and, I think, are as well accepted by customers as any software product can be.

MANACHER: Could Mr. Hoare give us some difference between the first and second effort's working method ?

HOARE: It was very difficult to find, to see why we had such a spectacular failure. I think that the main reason is that the things that we were trying to do were not possible. The machine had a very limited amount of core storage, though it seemed a lot to us at that time. It was not expansible, so it was not possible to retrieve the situation simply by more core which is the standard answer. In the first ALGOL effort the reason why we have not succeed was that we decided to do only those things which we knew how to do and we might do reasonably well. In other words, we started to do a rather small sub-set of ALGOL 60 which I designed in the expectation that I would know how to implement it. We decided to do a single pass compiler which, given a guiding structure in principle, makes compiler writing of a suitable language a very simple task. The size of the subset that we were implementing grew continuously during the project, as we realized that we could satisfactorily implement more and more of the features of the language. We had the freedom to choose what we would do. We had no fixed time scale, the project was one on which we embarked as more or less as an experiment. I think that the important thing was the freedom that we had to decide on certain overall principles, I think we had one stroke of luck, and that was that we hit upon the compiling technique – a method of structuring a compiler, that was extremely suitable for the language and very simple, that is the top-down method of syntax directed compiling. And we happened to hit on it as on a method of compilation right at the beginning.

STOLIAROV: We are far from being able to formulate the sufficient condition for the large program but we can formulate a necessary condition; and that is the presence of angry customers.

WORKING SESSION II : LARGE PROGRAMS DESIGN METHODOLOGY

RANDELL: "I think that this outline works well for small systems, where it is essentially unnecessary, but does not work, other than in exceptional cases, for large systems".

This comment was made at the recent Rome Conference on Software Engineering Techniques, following a carefully thought out presentation of the basic steps that should be taken in developing the design of a large software system. In fact I think that this can be said of any design methodology that I know of. Large systems do get built, sometimes more or less satisfactorily, and various techniques, which perhaps could even be called methodologies, have been evolved to help with the horrendous managerial problems that ensue when one has to organize a very large number of people in pursuit of a common goal. These, however, are hardly design methodologies.

The process of design is largely one of taking decisions. Ideally at each stage in the design process, one is conscious of which decisions have to be taken, what alternatives exist, and what rationale should be used in choosing between alternatives. The most important resources needed for all this are intelligence and experience – both are vitally important, and they are interchangeable to only a limited extent. (Certainly they cannot be made up for by numbers).

When a design problem is very small the decisions are few enough in number for them to be all considered in parallel – i.e. in taking one decision, one can bear in mind the implications of this on each of the other questions to be decided. For a design problem of any complexity, and certainly for those which are of relevance to this workshop, this is not possible, and one must choose (hopefully consciously) some sequence in which to attempt to make design decisions. Because there will be only limited opportunities to rethink design decisions, this sequencing is likely to have a considerable effect on the final outcome of the design. Much of what has been written about methodologies for the design of large software systems has concerned this problem of design sequencing. Here for example, I am thinking of discussions of "top-down" versus "bottom-up" approaches to design. However to my mind, the most careful and extensive thought on this problem comes from a different subject area, that of architecture and town planning, and is contained in the book "Notes on the Synthesis of Form" by Christopher Alexander. Alexander proposes a mathematical technique for dividing a large set of design problems into groups, and for choosing an ordering of these groups. First of all, what he suggests is that one should try to identify and write out the various negative features that you would like your town to avoid having. You would not like the town to suffer from air pollution,you would not like houses to be very noisy;and he stresses the advantage of trying to design an object so as to minimize its bad features. He said it was easier to list the bad features that you would worry about, than to list all the good features that you would like. Because when you start listing good features of a town or any other object, it is very difficult to distinguish between those which are good features and those which made the object a town. A good feature of this town is that it should contain houses; now if it did not have houses, would you call it a town ? So there is quite lengthy and rather well-written argument about the idea of stressing avoidance of bad features rather than listing all the good features.

Alexander's technique involves identifying all the design decisions that are of importance at some common conceptual level, and then assessing the degree to which each pair of problems are related, because a good solution to one will either aid or hinder the solution of the other. You fill a matrix indicating essentially unrelated and related problems, and then you have mathematical techniques for treating this matrix for choosing and ordering of the partitions which you would then consider as giving you a grouping of desing problems and ordering of groups. What you have done really is just choose, according to some mathematical criteria of partitioning, an appropriate order in which to consider sets of design problems which are closely related. This, by the time you read the book once, is extremely seductive. The problem of course is, how on earth do you get this list of design problems. And even if you have got it, how do you assess the degree to which problems are related. The only way in which you can state that one design problem is anti-related or closely related to another design problem, is because of the knowledge of the assumptions that you have made about the underlying reality. It

clearly stands of falls depending on the accuracy with which the matrix of pairs of correlations has been set up (even through the division of pairs of problems into those which are related, and those which are not, is probably all that is necessary). Alexander points out that this involves forming some conceptual model of underlying principles governing the object that one is designing. It seems to me absolutely clear that this approach cannot conceivably be of any use in design problems which are not essentially very similar to ones which have been done many times in the past. If you start out with a new design problem, and you have no idea of underlying principles, nearly all the questions are ones that you have not even seen before, then you can in no sense make a meaningful assessment of relationships. In fact you cannot even make a meaningful list of problems at the same conceptual level. You only know that there is the same conceptual level because of your experience, or the experience that you have obtained from somebody else. It would be naive to assume that we are in a position to do this with large software systems, or in fact even to identify all the design problems whose interrelationships should be assessed. When we have made many airline reservation systems, and if it is not the case that each of them has far more new facilities than its predecessor, then perhaps we will see enough consequences of different design decisions, different trade-offs that we might eventually be able to make a good job of it. At the moment, however, often all we are seeing is inadequate airline reservation systems. We do not know, for example, how big a machine, how much core storage, how many processors you should need for an airline reservation system. Computer manufacturers probably are quite happy with the machine power which apparently is necessary. We may, in fact, be a long way from knowing the best way in which to do, for example, the airline reservation systems. But it would be a very long time indeed before we have obtained all the necessary experience, and before we accumulate enough experienced and intelligent programmers. Therefore I am surely not being too pessimistic or cynical in saying that it will be many years before any such approach will be of any relevance to the design of large software systems. (And in fact, I now gather that Alexander has abandoned this approach itself - however I still recommend his book highly to all of you).

So, although the various writings on the problem of sequencing design decisions, and in particular those of Dijkstra, are definitely worth studying, there is not yet any magic solution. I therefore believe that it is important for a designer or designer team to be aware of the fact that they will not be able to get the basic sequencing of design decisions right, let alone make each decision correctly. They should, therefore, use as one of their guidelines the idea that the object they are designing, since it will be badly designed (though, of course, some are far more badly designed than others), should where possible be constructed so that it can be modified when redesign becomes necessary. An exceedingly obvious case of this, in software systems, is the use of high level languages. Other ideas which are of value for this reasons, as well as other, are tools which aid, for example, the tasks of documentation and communication of the design.

Somewhat belatedly, let me say a few words on the role of design, in the production of a large software system. Some people might claim that design is that which comes between specification and implementation. In the case of small programs, such as have been used as examples in papers about how to write good programs, how to prove that programs are correct, etc., one does indeed have a fixed, accurate, complete specification. However, with current large software systems the specification has been none of that. Rather, it has been little more than an initial bargaining offer, subject to continuous negotiation as the design proceeds, until it is discarded as further discussion proceeds solely in terms of the current version of the design. At the other end, I have yet to see an adequate distinction drawn between design and implementation. Instead, I would repeat Kinslow's classic comment that "you design until you think you understand the problem, and then implement until you know you do not".

Now I have expressed just one view about the design process, and have discussed just one way in which a large design task differs from a small task - i.e. that decisions have to be taken in some sequence, with only very limited possibilities of going back and changing a decision. I would, of course, like to have comments on these points, but I would also like to hear your view on the whole area of methodological approaches to the task of designing large programs.

HOARE: What you were saying is that it is no good hoping to be the first to do something in a new field.

RANDELL: What I am saying is that there is little hope of getting a methodological approach to something where there is no experience, on which to base this methodology. It is really rather sad that in the area which I would have thought we had - or somebody had - a lot more experience, such as the design of buildings, apparently, such a methodological process is not regarded as suitable.

TURSKI: First of all I should like to try to shake a bit the very pessimistic statement that once you have made a wrong decision somewhere in the beginning it turns out to be fatal for the whole undertaking, and there is no way around it. Of course, if you have such a hierarchy of decision making as a tree structure, it is quite true. But on the other hand, the hierarchical splitting of decision making is not the only possible way. You may try to think about, sort of, flat separation in decision making, so if errors are made in the individual decisions, they pertain just to the little part in which they are made. Of course, natural objection is that we know how to split the problem hierarchically: we start with the crucial decisions first, and then you will go further down to details and so on. But from all that we were saying before it follows that the large programs as we know them today, are not highly hierarchical in their structure, they grow up from clearly defined things, which are meant to do separate and relatively isolated tasks and then they are glued together. So perhaps the process of building this structure that you have is slightly different in time-order, namely you start with programs which do something hopefully useful, and then you decide: well, let us try to combine them into a system. So instead of going from top to down, we have those programs, which work individually, and then combine them into a system. The decisions you are making now are really on combination of things into larger pieces and each of them is again an independent entity which works as a black box, it has input and output, and its transition matrix. Then it only matters which of those boxes you combine into a larger box. So you are building like a house from bricks. Perhaps that is what you said about the intuitive approach. Lacking the experience as we are at the moment, perhaps indeed we cannot hope to build this nice and beautiful, logical tree of decisions, but rather we shall go from the things which we know upwards. There is none of this error fatality in this system, because every time you join something together, that is every time you make a decision to join things, you are getting something which is relatively independent. And you test its functionality, its usefulness at the moment you construct it. It may appear to be very wrong and very bad approach, because the resulting program structure may turn out to be quite ugly. So on this thing you should superimpose certain constructional rules, certain rules of building, exactly like in the house building you have certain rules which permit you to put one sort of bricks together but not mixing different sort of bricks because you know that the wall falls down. You are not interested in the purpose of this wall, they are just rules of thumb. So I would split the decision making into two parts: one is the decision making concerned with making bigger functional units out of existing functional units of proven usefulness. You start with something which is useful albeit very small.

RANDELL: First of all, I do not regard what you said as being opposed to what I said. One of decisions that one might take early on in a design period would be what modules shall we use of the set that already exists, what will we try to obtain from elsewhere, what will we program ourselves from the beginning. You might take that decision right at the beginning. Alternatively you might say: well, let us go some way into the design and then find whether any existing modules are going to be of some use for us; there are two sequences of design decisions. All I am talking about with that apparent tree, is the idea that you take a set of decisions, one decision is whether to use an existing module, or whether to program it yourself. The order in which you should take these decisions can sometimes be just important as each individual decision. But clearly you do not take decisions one at the time. What I am really talking about here is the idea of considering a certain problem area, taking a comparatively small group of decisions, and then going on. One way of structuring the system is into sort of levels, you might define set of interfaces and then look inside them, and check whether you can build what is necessary in order to support

such an interface. Clearly you must do that in two separate groups working in parallel. However, if later you find that the interfaces were wrong, that you designed them before you knew enough about how to choose the right decisions, then you have great troubles.

STOLIAROV: I have two comments, first: a minor one, that there should be a name attached to this methodology. This method should be known by the name of Descartes-Randell method. The second is that when we have the correlation matrix it does not necessarily determine a tree, it may be a multigraph and the trouble with which we are concerned in general is the general decomposition problem, that is a problem of finding the most suitable partitioning of the problem, into subproblems having the least possible number of interconnections.

RANDELL: In his book, Alexander describes one proposal for doing this and I gather that he has written programs which will do such decomposition for quite large matrices. But I think that we are very far from being able to use such a method in practice.

KUROCHKIN: May I ask a question: I did not understand your restriction, how do you study negative and positive politics, because not having negative qualities is equivalent to having positive ones. Not being noisy is equivalent to being not noisy.

RANDELL: Certainly it would be possible to take any quality, put the negation on it, and say: that is something we would like to avoid. In fact what you do, is again based on your own and everybody else's experience of towns that have been built, and had agreed faults. It all depends on the vast amount of experience, that you had in the past. In programming, there are very few trade-offs that we have much knowledge about. We know, in general, that we can trade space and time, but for a given algorithm we often do not have sufficient analyses of the behaviour of the algorithm to know that we can trade-off one quality against another quality, performance in one aspect against performance in another aspect.

It is very difficult to schedule inventions - "next year I will invent so and so". And often that is what is involved in programming, particularly in programming something which is a difficult problem, in which you have little experience.

LAVROV: To avoid undesirable, negative features seems to be easy enough, but essentially it is not too good, because avoiding some known negative features we never know which new negative features we shall get.

RANDELL: Hopefully, as we shall build more and more villages and towns, we gradually exhaust a list of possible negative features. However, to deliberately include one negative feature in a fear that if you remove it, another one will appear, would not be appropriate.

HOARE: I have a possible point here, I think of a place where a programmer has very great difficulty in taking major design decisions. He can be quite experienced in taking small decisions at the low level: how things are to be coded, and how information is to be represented locally in a part of the program. And he is used to expressing these decisions in terms of very strictly deterministic code of his program. Whenever he comes to have to make global design decisions at very early stage in the design of a large program, he is completely lost, because he does not know, first of all, how to take a decision on technical grounds, and secondly, how to express this decision in a way which separates one decision from other decisions which would perhaps be irrelevant. The code of a small piece of the program expresses all the decisions that he has taken during its construction, practically each command, expresses a decision he has taken. And therefore his practice on small programs does not give him any guidance as to how to separate decisions from each other, how to avoid taking unnecessary decisions, merely as a result of the fact that he wishes to describe a particular decision in detail or with great precision. One helpful suggestion is the attempt to teach designers to express their indecisions in a rigorous and bounded fashion, and to be satisfied with imprecise definitions which are yet sufficiently precise for that purpose. It is very difficult for programmer to feel satisfied with such things.

RANDELL: I think that very often the decisions taken in the
f a large program are in fact taken unconsciously.
o not realize that they have taken a decision, and certainly do not realize the other quite obvious alternatives which they have discarded. Going back and trying to justify all the design decisions that you have taken often can have a very good effect on the actual program itself. If you really made somebody to document his program, and he has the freedom, on realizing that particular piece of documentation is very difficult, to change the program - so as to make the documentation easier, then he probably will improve the program and any of the usually relevant measures of program quality; but it involves training people to be self-questioning all the time: What where the reasons behind the decision to do this, what did I know when I took this decision, what did I assume ought to be the case even though I have got no good reason to justify it. Somebody who, ideally, documents this for the benefit of other people, and certainly thinks this way, is designing well.

OLSZEWSKI: Do you think that a correct result of programmer's work would be represented on the decision tree by one single leaf ?

RANDELL: I think that the final product of the programmer's work should not be just a program, but, with the program, enough information to show the set of decisions which were taken in making the program and the alternatives which were considered and discarded. So, to a certain extent, though certainly not completely, I would expect rather more of tree that just one path through it to be in existence after the programmer has finished his work. That, of course, is assuming that his program is going to be treated as more than just a blackbox, never to be opened, and looked inside by anybody else, including himself later. To document a program and to indicate the decisions which lay behind what you have done, makes a program a lot more understandable, and makes it a lot more useful. If you want to change program in some way, then statements of design decisions or alternatives will often point the way to the implications of change in some parts of the program.

TURSKI: Take a full decision tree, and select one of its branches. You want to include in the documentation the discussion of possible design variations. What I am inquiring is this: how deep should we go in the discussion, if you do not want to give a full tree, in its entire complexity - which would be very nice to have, but so impractical that we can not suggest it as a good methodological approach to anyone.

RANDELL: So what I am suggesting - is that you document the arguments that you used in actual design.

TURSKI: So you would like the people to admit their ignorance ! I claim that until you develop all branches to the end points, you do not have the evidence to show that your decision was correct. If you are going to obtain the evidence of the correctness of your choice by looking at the discarded branches, you have combinatorially many problems to solve, rather than one problem.

RANDELL: I am not trying to do that. I am saying that if you do document decisions, then your attention will be drawn to obvious inconsistencies, and the program that you produce, will be more understandable, and even perhaps somewhat more easily modified.

MANACHER: So, this thing, if I understand correctly, is being recommended largely as an alternative, perhaps a very powerful one, to putting in comments into large programs, macro comments in a way.

RANDELL: If you control a long design process, you often forget where you took decisions, and sometimes a decision you took turns out later to be clearly based on misconceptions. And if you have tried to impose this discipline on yourself and to document your decision making processes, then there is more than a chance of your making a better job of it; not proving correctness, not exploring the combinatorial freedom of possible designs, nothing to do with that.

MANACHER: Let me say something in support of this point of view: I think the thing that you forget the most easily is not why you took the particular decision out of many, but what was the appearance of the nature of the decisions that you decided not to take at the moment that you took the decision. In general, when you construct a complex program, the most agonizing thing in the world that I have discovered i. not just the program, but a language design. Right now I am doing the language design of a quite complex language, perhaps of the complexity of ALGOL 68. And I have discovered that the most maddening thing is going back over some decision which looked right at the time,and look shaky now.

LAVROV: We touched upon very interesting point: which sort of solutions should be taken, optimal or acceptable ones. Nowadays it is very fashionable to talk about optimal solutions, optimal controls and optimal everything. If we look what is happening outside of the realm of human life, or even inside human activity, we shall see that many decisions taken by the nature are just acceptable ones, not necessarily optimal; perhaps they could be improved. For instance, if we take the life, and its variety of forms , we see that we cannot say that any single one of them is optimal, all of them survive because all of them are acceptable solutions to the problem of life forms on the Earth. Of course, the evolutionary development has one serious drawback - it is very slow. The development of human culture and human knowledge over the last few centuries is much faster than normal evolutionary development, it has a higher derivative, perhaps even higher second derivative. But even in the human society we can see many solutions that are far from being optimal. I said previously that the tendency to avoid the negative features is not conducive to fast progress, but in general I think that this way seems safer because if we try to achieve certain goals, positive goals, which were not tested yet in any place, at any time, it may turn out that when we have achieved these goals, we get by the same token some other features which we did not opt for at all.

Now, coming back to the problem of selecting versions going down the decision tree, we have to consider all paths on the tree only if we are looking for the optimal solution. If we can be satisfied with an acceptable solution, then at each step we should be concerned only with selecting an acceptable choice, not necessarily all possible choices. Of course, it may happen that following the set of acceptable choices we may arrive at an unacceptable outcome. If we consider an analogy with playing chess; every player knows that after a sequence of moves, each of which can be considered as a reasonable move, he loses a game. But in the design we should always have an opportunity to replay game; when we discover that we lost the game we should be in a position to come back to the certain point and play again, trying another solution.

STOLIAROV: Therefore, an acceptable solution is the one which we accepted.

MANACHER: I would like to disagree with one of the main points made by Prof. Lavrov, namely that it is not such a wise strategy to proceed negatively. In programming, very often the negative decisions are extremely subtle and extremely specialized or marginal, but nevertheless they are logical and if they are negative, they are a sufficient reason to rule out a path. In real life if you have a path which is negative but just for a very small reason, sometimes you do not pay attention to it, but I think in programming you have to, and that many decisions are made because of strange subtle negative reasons that prevent your taking a decision.

RANDELL: When you have got enough experience in building towns and villages there is a general agreement as to what town or village is. People faced with an object will be in a general agreement: this is a town or a village, or this is not. The problem is to try and make as effective a town or a village as you can. When you state the criteria in positive form, it is sometimes difficult to avoid going down and listing criteria which really are an essential part of something being an object of that class. You will find then easier to avoid doing this if you just concentrate on known or feared design faults. You would not criticize a railway-engine for not having houses in it, it is irrelevant. One point Prof. Lavrov made: if you take a set of adequate decisions, will the end result be adequate ? There are certain aspects of the problem of designing large software which I would call system aspects. For example: reliability and size; the reliability and size of a large program depend on almost all decisions together. You cannot isolate, find some small group of decisions, giving the reasons why the reliability or the size of the system are inadequate. Or, shall I say, it is not usually so easy to isolate the faulty decisions given something too big or too unreliable.

HOARE: I should like to turn your own remarks back on yourself. I know that you were engaged in the design of a very large and complicated object with no experience of your own or anybody on your team in the design of such an object. I am going to ask you whether there are not some ground rules, methods that you believe are conductive to the success in breaking new grounds in this way.

RANDELL: The business of documenting the reasons behind decisions, and listing the alternatives that you considered and discarded, was something that we have used very extensively in the ALGOL compiler. And in some cases it made us go back and change, and in fact improve the compiler, so I am talking from my own experience, I am quite convinced about it.

The current programming habits are such that most people are satisfied the moment they have a finished program which gives the right results.

There is that famous, or notorious, study intended to be an investigation of whether conversational or batch processing was better for program testing. The study involved getting different programmers to do the same program. They soon found out that even within a group of programmers, all of whom would be regarded as adequate (they could earn their living, get about the same salary and at least by that measure they would be regarded as roughly equivalent programmers), tremendous variations were found, factors of ten or fifteen in the time taken to produce the program, factors of four and five in the size of the program. Now those sorts of variations are not the sort that one really should feel happy with. When you design a program, you very often construct the whole set of decisions, then you are forgetting about them and accept the whole program; you do not realize that there is an alternative which would be two or three or more times faster. When we get to that stage in programming, then this sort of technique may be even more valuable. What are your reactions to this sort of idea of trying to make the design process a very conscious process, making it very introspective ?

MANACHER: As to the value of massive introspective and conscious documentation I think it extremely valuable even if there are some unconscious assumptions; and I would suggest to anyone who doubts this that he obtains from Illinois Institute of Technology a copy of a new implementation of SNOBOL 4, and he will find in that documentation an ample justification for what I was saying.

RE: It has been example of good documentation or bad ?

NACHER: Good.

HOARE: Do you think that program documentation should be done before coding or after it ? This is very practical question. There are many programmers who actually do it afterwards, and I cannot help feeling that it is a disastrous mistake.

TURSKI: I doubt if it is possible to do it before; it is much better to do it afterwards then never at all.

KUROCHKIN: Do you think it is possible to do it after ? Nobody has to do it.

HOARE: The point of documentation is that it should be readable, and unfortunately there is no way of forcing unwilling people to write readable prose.

TURSKI: The ideal solution, of course, is that documentation should be done simultaneously with the coding, so the coding itself is the documentation. If you program in such a way that the program is a readable piece of prose (or poetry, depending on your taste) then you cannot avoid making documentation while you do programming. Coming back to the problem of documenting your decisions, with respect to the branches you did not take, I still cannot force myself to like this idea; when I think of things which I did not do, though I could conceivably have done them, I do not know how to use this information. I am much more interested on documenting the decisions made, as soon as they are made, but in the positive sense; so that I could remember why I decided to do so and so, rather than for recording why I did not do so and so.

RANDELL: What I am saying is that you should indicate which possibilities you considered. Clearly, you do not list down all the conceivable techniques which clearly are inappropriate. If you make a list of the decision and the alternatives that you considered and why you chose one, and therefore discarded the others, that is all I am asking for.

TURSKI: "Others" in general, without specifying which ones were discarded ones.

RANDELL: I am assuming that you chose "A" over "B", "C" and "D". Presumably you do not say: I did it this way because it was good, I did it this way because I think it was better than that and that.

TURSKI: Surely, not always; you work in this way only when you consider several clear-cut alternatives or variants. Surely you arrive at the decision, sometimes at least, by a constructive

process. When you construct just a single good solution, good decision, the discarded things are not the alternatives of this solution, of this decision. They are little tiny bits which you reject when you are constructing the decision at a given point.

RANDELL: Sometimes such action is fine.

MANACHER: I think it probably happens, that there is a paradox: "A" is more plausible than "B", but when carefully analysed, "B" is correct nowadays. I think those types of paradoxes occur constantly, and not having a record of one day affair is really to loose the grip of your thinking at that point when you made decisions.

RANDELL: If either you impose on yourself a discipline, or if, alternatively, your boss demands the discipline that you write down "this choice was arbitrary", then your willingness to admit that the decision was arbitrary indicates that you put certain amount of thought into trying to make the decision, before admitting that you had not enough knowledge to make it.

LAVROV: If we were able to describe the sequence of decisions made in developing a program, and moreover add to this description the whole underlying reasoning, then, indeed, we would have had a marvellous documentation. I think, unfortunately, it is quite unrealistic to hope for that; first of all, in reality we are taking such tremendous amounts of decisions that it is physically impossible to preserve the records of all of them. It will become more impossible if we try to attach to them the underlying reasoning. We will make fools of ourselves because, of course, each of us makes a tremendous amount of ridiculous decisions every day. That is, the maximum one can hope for is that when the full set of decisions has been taken to record why and how the most important ones were taken. Generally, I think that proposal for the documentation is so complicated that, in order to do the documentation of many sizable problems, you have to spend not less effort on the documentation than on the problem itself. It is not customary for scientific publications to provide such documentation. For instance, mathematicians generally publish proofs of their theorems, but they never say why did they prove this in this fashion. Similarly, in natural sciences, there are often published the results of experiments, and very seldom we ask why they did such and such experiments, and why other experiments were not performed.

Now few words about the documentation of programs. I think it is reasonable to distinguish three levels of documentation: (1) internal documentation for the designers, (2) maintenance documentation, (3) user's documentation. The documentation of the first level, i.e. for the designers, should be as detailed as possible, because the designers are going to change their product anyhow and they need detail documentation. For those who are concerned with the maintenance of the system the documentation may be considerably less voluminous. The users have no need to know anything about the program they are using; the only thing which is really required for them is description of the input/output language of this program. I do not know if any good documentation of programs exists in the world, but in all examples of programming establishments I ever saw, the documentation was always the weakest part.

RANDELL: First of all, I agree that one cannot hold complete listing of all design decisions, but it does not mean that you should not have any. Secondly, I would say that it is exactly that sort of information which is very valuable for the internal documentation for the designers.

OLSZEWSKI: I would say something about the documentation of the operating system SODA. Quite large amount of the documentation has been made for SODA, but it is now a documentation for not exactly the system which has been made. It was the internal documentation for SODA makers. I would like to try the best methods to improve the documentation, but I do not feel that the internal documentation of SODA needs to be improved.

RANDELL: I mentioned much earlier the fact that very often in design of a large program system, one could make use of very simple tools. Many times the task of providing the documentation, for keeping it up-to-date and so on, involves clerical tasks. For example, in Communications of the ACM (April 1970) there was a paper by H. Mills describing a very simple technique of aiding the programmer in producing the documentation and attaching it to his program. He called this paper "Syntax-directed documentation for PL 360". It was really very simple, but you could see that if intelligently done, it would be unnoticeable clerical

assistance. Very often that sort of impediment is one which could be easily removed by an appropriate action on the tools. Sometimes software designers are not very good as software users. They tend to think of the computer as being something for other people.

HOARE: Surely, the operating system SODA is intended to be adapted to various environments. The adaptation of a program is terribly risky business with respect to taking right decisions in the very complicated context and avoiding the introduction of the software errors. Surely, this is a very strong motivation for producing good implementors' documentation; particularly documenting decisions of the paths that have not been taken - because it could be that in different environments the rejected paths should be taken. I do not see how you can say there is not much to be gained by writing a complete and correct documentation even at this stage.

OLSZEWSKI: I am not quite sure that SODA should be implemented in new environments, but if it will be, then of course the documentation should be made as Prof. Hoare said.

RANDELL: It is rather tempting to say that any programmer should be taught to assume that his program would last for long time, will be modified and used as the source for ideas for further programs. But just as there are some programmers who take take a very narrow outlook and program exactly to the specifications, there are also programmers, at the other extreme, who will always try to generalize everything; when asked to make a bicycle they try to design a whole transportation system.

HOARE: There are analogous disciplines, for example, physical sciences, where the practice of documentation is very much better than programmers are willing to accept. The laboratory experiment is carefully written up, day by day every event is recorded in the laboratory note-book. If you take a science such as archeology: the degree of care with which every item and the exact place where it came from are documented although this information on such a level of detail will never be published. It might be for establishing something, you never know.

I can report a recent practice in my own university. We tried to accomplish the impossible task of giving the student-programmers the background to the writing of large programs when they still only have very small programs to write. But we have established a discipline: each programmer keeps a note-book, a day-book, in exactly the same form as a laboratory note-book for his chemistry or physics practical classes, and he records dates and times of every job submission, he records his flowcharts, and the most important - he records and explains every programming error that he detects, and tries to explain the correction if he can, and so on, reporting everything until the program is working, until he thinks it is working. His books are examined by junior staff and form the basis of an assessment of the student's programs.

TURSKI: Does anyone know of any real-life programming effort in which full documentation exists ? For example, documentation along the lines HOARE just mentioned. This is splendid, you may execute that for students with the junior staff examining and making assessments. But if you have real-life circumstances and you made a compiler for a fairly large language; does such documentation really exist ?

RANDELL: I have the impression that people try to follow it, at least in quantity, if not the quality, OS 360 certainly does. The documentation of the OS 360 development is kept on microfiche. There is a large number of microfiche readers, in which you project the picture so that you can enlarge and read the documentation, and each programmer has a very large set of microfiche cards and they, perhaps, have twenty or forty pages on each microfiche card;you will realize that at least the quantity of the documentation in the OS 360 development work-book is huge.

HOARE: There is another aspect of documentation in the management of software project, on which I would like to propound a theory, completely unsupported by practical experience: every piece of code and of documentation should be written by one person, and read by at least one other. The management of every project should insure that the responsibility for the code and for the documentation is carried both by its writer and by its reader, possibly at different times. And my suggestion is that the code and documentation should always be read by a programmer's manager. In no other discipline is a junior permitted to submit work completely

unchecked by his superiors. The second scan is made at the stage when your program is handed over for routine maintenance by different team because at this stage the maintenance team must take responsibility, and therefore must have the opportunity of checking that the product has in fact been properly developed. Such a handover procedure (to read the code and the documentation) will be more effective than carrying out the process of supplementary program testing, to which some manufacturers give curious name: quality control, or product test. But I do not know of any project which has in practice carried out this idea.

KUROCHKIN: I think that good documentation is not here yet just because the process of obtaining that documentation, for example as described by Mr. Hoare, is too complicated. Perhaps we shall be getting documentation at the same time as we will be getting the code - if we have the necessary aids to produce the documentation in a semiautomatic fashion. A programmer always starts by making a documentation, may be very unsatisfactory, and only later on he writes a code. At the beginning he has the input and output information, makes the flowcharts of the algorithm, very often he writes comments on his code and only afterwards he writes the code itself. If we had a suitable hardware, and - first of all - software to read out such preliminary programmers documentation and to reproduce it in more orderly fashion, we would have made a large step forwards toward the good quality of documentation. I think that the programmers would be prepared to make certain sacrifices, if they knew they would get something for that, a good documentation for example. I am not sure because I never saw a really good documentation, and I do not know how to get it, but I think it could be possible to get it.

HOARE: I would like to express my prejudice that documentation produced by machines is only fit for machines to read.

KUROCHKIN: If the documentation is printed on a typewriter, it does not mean that it is designed for typewriters.

LAVROV: I do not see much difference between documentation and program itself. Both of them are products of my considerations and are realized in the form of an ALGOL program or several words in Russian.

When I am writing a program, it is quite natural that I make corrections in it and, of course, towards the end of the programming process my program is quite different from that used at the beginning. This is an outcome of a sequence of my decisions. It is suggested that this set of decisions should be recorded, presumably in some natural language. But when I start writing the natural language, I am going to introduce another set of corrections to the text. Should I also justify all corrections which I introduce to the description in the natural language ?

KUROCHKIN: I think that the documentation should reflect first of all what has been made, and not how it has be made.

STOLIAROV: We have observed that there is a need for several levels of documentation. If a translator was written in, say, ALGOL, it would not be enough for the user to have the ALGOL text of this compiler. Maybe we are exaggerating the problem of documentation and we ask that it should be overoptimal; let it be at least acceptable, the programmers will soon recognize that they belong to ranks of normal people, and they will make normal documentation as any member of any other profession does. It seems that when we discover that about 30% of effort goes to the documentation this is new only for the people who have university background and not for the ones who have industrial background.

RANDELL: I wish that there was a supply of programmers trained, as HOARE said, to document what they have done, in the same way that laboratory experiments are documented, and examined by this method. That clearly produces one kind of documentation, mainly about the program. Certainly the information in those note-books could be very useful. But I do not think we should confuse the idea of the information about a program which is gathered together at the particular stage of its development with the information organized in each individual program.

LAVROV: When we compare an amount of effort which goes into the keeping the laboratory note-books, with the amount of effort necessary to perform all the documentation required from the programmers, we shall see that the amount of the work the programmer should put into their documentation exceeds the amount of work necessary to produce the laboratory note-books by a factor of several tens or perhaps even hundreds. I do not think that keep-

ing laboratory notes takes more than 5% of the work-time of people working in laboratory. And here you are asked that you should spend about three times as much time on documenting than on writing the program. I am familiar with the documentation not only in programming but also in other kinds of industrial production and it is quite clear that documentation is poor not only in programming.

RANDELL: How much documentation in the laboratory note-book would you regard as acceptable ? How many pages would you fill per day ?

HOARE: I think it will be very much more of 5 % variety of total time spent.

RANDELL: So about 5 %.

TURSKI: No, very much more than 5 %, about 30 %.

MANACHER: Certainly.

HOARE: Part of the documentation is of course identical with the design process, with the program. First of all, the student must write out a copy of the problem as set. And then he must say how he is going to tackle the problem. Then he usually puts on the flowchart, and perhaps even a copy of the program itself. And then he just records the errors and the corrections, and with each error he states what the effect of it was, and why it was wrong and why he expects the corrections to be better.

RANDELL: How much extra effort goes in these laboratory note-books ?

HOARE: I think the person would have to write some sort of flow-chart and description anyway, so the extra effort is almost measured by the fact that some students make a fair copy in their note-books rather than writing directly in their note-books. It is, I think, as compared with normal documentation practice, perhaps 5 % more than that. Of course, we have the ordinary documentation as well, as a part of the process of designing a program; keeping the laboratory note-book does not add very much to the task.

RANDELL: So, what HOARE is suggesting, is a very much smaller thing than you tought.

MANACHER: Is there a general agreement that one of the objectives of documentation should be that someone reading the program good deal of time after it was originally done, should be able to pick up sufficiently the thought processes of the original programmer, in order that he will not miss an important nuance.

RANDELL: I do not need stressing the advantages of recording this from the point of view of the original programmer. I think it improves the design to be conscious of why you took those decisions. How much this documentation will be later studied by somebody else would depend very much on circumstances.

MANACHER: HOARE has suggested that there is a very thorough review of the code. I just wonder whether that review should have the criterion I just mentioned.

HOARE: That is absolutely the criterion, really enforced by the fact that the people who have to make changes in the code are those who check that the documentation is adequate to enable them to understand it.

RANDELL: The task of the programmer is not to produce a program, but program plus documentation. And the customer will accept this only when he is convinced that it is safe to take responsibility for it.

MANACHER: I may say something, perhaps very obvious, about one of the necessary conditions of documentation, and that is that a program does not realize an algorithm simply by the short-hand code one sees on every page, but rather by the assumptions which that code is making, as to the current condition of the data on which it is operating.

I believe it would be easy to show that it is not trivial, for instance: if one gave a page of code, and said: "this code is for following purpose: given certain integers it will compute so and so; however, it will only do so for certain integers". I would maintain that a great many programs are written mostly, but not exclusively, for the purposes of efficiency in making precisely such assumption, that is to say, there will be patched code which sets up these integers and then these integers which it generates are fed into, let us say, one hundred lines of code. These hundred lines of code know something about those integers which are being fed into it; and consequently are a good deal less general than

they would be were any integers allowed. Now, for a person reviewing those lines of code, or even for the original programmer going back and having forgotten the motivation, looking at the one hundred lines of code and realizing that more or less its intention is to compute a certain function, but forgetting the restrictions which that code is assuming, is simply a suicide.

Therefore, all remarks such as we have heard, suggesting that the lines of code are good approximations to the actual state of affairs with regard to the algorithm which is being encoded, and that mere casualistic remarks are sufficient to reconstruct the nature of what has been done are, of necessity, totally erroneous. I maintain that the documentation must at least have the ability to make this type of reconstruction. Obviously, this type of situation involving an easy potential for a situation which is mathematically unsolvable, does not exist in a physical laboratory. Therefore, there is a qualitative difference between two situations, and they are not to be directly compared.

RANDELL: Let me make one last remark by saying that one of the first documents written on programming (which is in fact the report by Burks, Goldstine and von Neuman) contains in it suggestions that the method of programming involves writing pieces of program and making assertions, even uses that term, about the state of a variables set that are assumed before a piece of program, and will hold after the execution of this piece of program; and that the business of convincing yourself that the assertions are consistent with each other, and with the program is part of the task of programming.

WORKING SESSION III : PROVING CORRECTNESS OF PROGRAMS

LAVROV: In recent years papers have been published in which the desirability, the possibility, and the necessity of proving the correctness of programs were established. In this connection the names of J. McCarthy, R. Floyd, P. Naur, C.A.R. Hoare, E. Dijkstra should be mentioned. The ideas developed by these authors are very interesting and useful. Their practical implementation will certainly assist the development of programming methodology.

In my opinion, however, such an approach can be considered neither universal nor even widely applicable. Those cases in which the correctness of a program can be proved are rather exception to a rule. I shall try to argue in favour of this point of view, but I am not certain that my opponents, being adherents of formal proofs, would consider my arguments as being strong enough.

A program is a formal object. This irrefutable statement is given as one of the grounds for the correctness of a program to be proved. Not every formal object, however, needs a proof of its correctness. Axioms do not need proof, and neither do formal definitions.

For example, one can write a program (in LISP):

```
(FACT (LAMBDA (N) (COND
    ((LESSP N 1) 1)
    (T (TIMES N (FACT (SUB1 N)))) )))
```

or (in ALGOL 60):

```
integer procedure fact (n);
    value n; integer n;
begin integer i, k; k := 1;
        for i := 1 step 1 until n do k := k × i;
        fact := k end
```

The question if any one of these (and many other) programs does calculate the factorial function n! for $n \geqslant 0$ has no sense. Every one of the programs can be treated as a formal definition of the factorial function. One may ask if all these definitions are equivalent, or which language is more appropriate for producing such definitions, but these questions have no direct relation to the subject under consideration.

If one considers an algorithmic language as being used merely to describe algorithms for their subsequent execution by computer then the treatment of a program (or of a part of a program) as a formal definition of an object may seem to one quite unnatural. With this approach to the language a program is but an image of some different, primary object. Then the question to what extent this image is precise, is legitimate.

I am certain, however, that the algorithmic language used by a man is not merely programming means of inputting programs to a computer language. It becomes a part of his thinking device, one of the means of describing either the physical or abstract world. Therefore, if only in some cases, a program itself is a primary object and there is no other object in comparison with which one can judge whether the program is correct. If somebody does not agree with such a point of view on language, may I ask him to say what is the sense of the long and still continuing attempts to create machine-independent, problem-oriented, or as one even says human-oriented, languages.

In support of this view I can mention the well-known idea that an algorithmic language can only be properly defined by its processor. There exists a number of practical (usually but partial) implementations of the idea. If the author of a language actually uses this method of definition, then the program of the processor can only be checked upon the correspondence to his intentions. But intentions are such elusive objects that such a check cannot possibly be formalized.

Let us consider now the opposite situation when there does exist an object, which is quite distinct from the program, with which its correspondence should be checked. I shall call it a problem statement.

So we have a program and a problem statement. Besides, we can add to this picture one more object, that is a problem itself. And the problem itself is an object, which exists independently of our statement. Problem is something which exists independent from us, whether we realize its existence or not, whether we try to formulate and solve it or not.

At present, a problem statement in most cases is a verbal description of what the program has to do and what conditions its results must satisfy. Natural language does not allow this description to be exact. A verbal description is hardly ever complete.

These circumstances make it impossible to check formally whether a program meets the informally stated requirements. The adherents of formal proving of the correctness of programs recognize this fact. Therefore, they consider the existing situation as unacceptable and demand that a problem statement should be a formal object and hence should be described in a formal language. So in fact, if we want to have a formal proof of the correctness of the program, then we should have not only an informal problem statement but a formal problem statement, too.

Let us put aside the practical problem of development of such a language (though it can hardly be a much simpler problem than that of the creation of a universal programming language acceptable for nearly all users and appropriate for almost all kinds of problems). In fact, there should not be two separate languages but two parts of a single one.

The point of the size of a formal problem statement is not one of principle, although it is of some practical importance. If the problem statement is much larger than the program itself (and this is not so unlikely) then a user would be rather burdened with its composition. Even if the user is certain that the expanded efforts guarantee the proper work of the program under all possible circumstances, he will not too often agree to the efforts - human wisdom has its own limits.

Let us assume, however, that a most conscientious programmer has tackled the job. The transition from an informal problem statement to its formal description cannot be formalized itself. The statement of a large and complicated problem would inevitably be inexact and incomplete. This implies that program composition should be based rather on an informal problem statement than on a formal one. Then it will be possible to detect some errors in both the program and the formal problem statement while checking the former with respect to the latter. Certainly, the errors may well correspond to each other and may therefore not be detected.

Suppose then that a complete formal problem statement as well as a corresponding program be created. If a sufficiently wide class of programs is considered, then it is rather evident that the search for a formal proof of the correctness of programs (of their correspondence to the problem statements) is an algorithmically unsolvable problem. That is, the formal proof of cor-

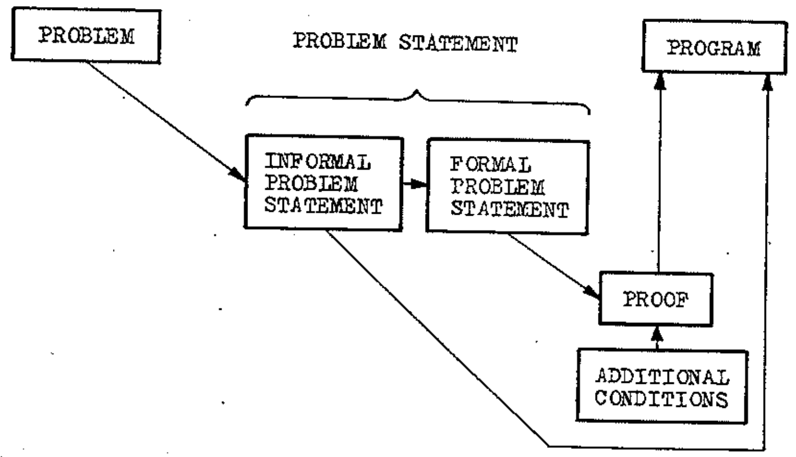
respondence between a program and a formal problem statement usually cannot be found automatically, but if such a proof is found by a programmer, then there should be a possibility to check whether this proof is correct. Or even it may be possible that the full proof is not necessary, and one may only formulate the basic ideas of such a proof. So we have the proof, and then the program.

At this point lies the weakest point of the whole conception. First of all, the complete record of the formal proof is much larger than the program to be proved. One can see this even from the very simple examples that have been published. Again, the difficulty is not one of principle. It could be alleviated if only key points rather than the complete proof are required. The psychological factor mentioned above still has its value. If this proof is very hard to find and to describe, then users do not plan to make such a thing.

Now many users will not be able to find the proof because of the intricacy of the task. Let such users be not allowed to run their programs on computers. I am afraid, however, that many very competent programmers may happen to find themselves in such a situation.

It is well known that the successful solution of a computational problem often critically depends on the stability of a computational method, its convergence, its speed of convergence, etc. There exist a great many fine theorems on this subject for various numerical methods. Unhappily, the conditions of the theorems are expressed in terms of high order derivatives, proper values of some matrices, the distribution of the roots of some polynomials, etc., and are usually nor checkable in practice. These theorems give the user some guidance when choosing between various methods, rather than help to draw a definite conclusion on the possible success of the chosen method. Therefore, the proof of the correctness of a program for the solution of a computational problem is of necessity a conditional one: if such and such additional conditions are fulfilled with the given initial data then the program gives a desirable result. So we have additional conditions, which follow neither from the informal nor from formal problem statement, but we stay on our mathematical knowledge, and mathematical capability of proving such statements.

As a rule, these conditions are sufficient but not necessary ones.



Approximately the same comments are true for almost any program. Suppose even that it has nothing to do with calculations, so that the question of rounding errors and of the representability of final and intermediate results in the machine code, does not arise. Even in that case there exist the problems of insufficient storage of some level and of insufficient running speed of the program.

The strict theoretical bounds are often overpessimistic and the practical performance in practice of a computational method is usually much better than that given by the theory. Therefore, a strict proof of the correctness of a program if it does not use a very sophisticated technique may be extremely discouraging to the user. As a matter of fact a conditional proof makes it necessary for a user to insert in his program the check of all additional conditions which guarantee the correctness of the result. So a program should not only implement the main reasons, but also the check of these additional conditions, often not related to the essence of the problem. Again, this may complicate the program very much. At the same cost one may often improve the performance of the algorithm and/or quality of the result if it would not require a new proof of the correctness.

The distinct understanding of these circumstances brings a user to a particularly difficult situation when his program has given him a very plausible result but the conditions ensuring its correctness are not satisfied. A mathematician should reject the result as not deserving of his confidence. An engineer never does so.

Unhappily, one cannot wave the matter aside by insisting that every solution of a problem should be proved and improved till absolute certainty has been reached. There exist such things as algorithmically unsolvable problems. I completely agree with van der Poel's assertion that nearly all interesting problems of practical value are unsolvable. No program can be proved to solve such a problem. Nevertheless in computing centres throughout the world people do solve transcendental equations, find extrema of functions, extrapolate functions, handle experimental data, simulate technological, biological, economic and social processes, and all this is carried out using programs which do not ensure the correcntess of a result or even the possibility of getting one.

One has no more reasons to demand that all this activity should be stopped than to insist on stopping any scientific research unless it is proved that its result will lead to the destruction of the humanity on Earth.

I would like to conclude by returning to a point I made at the very beginning, namely: the concept of a strict proof of the correctness of programs makes an essential contribution to programming methodology. Such a direct proof is sometimes neither possible nor useful as we have seen it. This only means that one has to find other ways to check the correctness of a program. And there are many.

Let for example a program be written which calculates the greatest common divisor d of two natural numbers m and n. Sometimes it may be strictly proved that the program gives the right answer with any $m, n > 0$. However, one can instead supplement the program with further instructions which calculate other two integers a and b, such that $d = am + bn$, and which check that this equality actually holds and that d divides the given m and n. Such a program gives a confirmation of the correctness of its result on every run which is quite sufficient, without any formal proof of the correctness of the program.

One can also combine the proof of some properties of a program with familiar usual debugging techniques.

Let a program contain a parameter x and let us denote the statement that the program gives a correct result when executed with a value a of the parameter by $S(a)$. The statement $S(x_0)$ which may emerge from a trial run is by itself of no importance. If, however, one proves the statement.

$$\forall a(S(a) \rightarrow \forall x\ (0 \leq x \leq a \rightarrow S(x)))$$

and if x_0 is the upper bound of possible values of x, then the empirical result $S(x_0)$ would complete the proof of the correctness of the program. So one needs not prove all the correctness of the program, but only this property of the program, and this result can be got when program is being debugged.

One can carry on with such examples. But the conclusion seems to me to be clear enough: although the straightforward approach seldom leads to our goal, one always can find means if not to assure the correctness of a program, then at least to make us very confident of its results. That is what makes programming to be an art. It is also what makes it so attractive.

RANDELL: I agree with what you have said. One thing you missed out was the comment, which I think is true, that attempts to satisfy yourself as to the correctness of a program sometimes lead you to ways by which the program can be improved. Sometimes when it is difficult to satisfy yourself, or your boss, that some part of the program is correct, it will indicate you a way in which it is to be simplified.

TURSKI: We seem to talk about correct or incorrect programs; I do not know what does it mean that a program is correct, or not. What is the meaning of the statement that a program is correct ?

LAVROV: In my opinion this means at least that the program corresponds to the informal problem statement. What does this mean I cannot say because this cannot be formalized.

TURSKI: Well, there are so many different forms of correspondence!

LAVROV: The problem statement contains the requirements for data on which program has to perform, and for results of its action. I think that, in fact, the main difficulty is to state this state-

ment, because usually we do not well understand the problem itself. An example form my own experience: when in 1960 we decided to write an ALGOL 60 compiler, we even did not understand that ALGOL 60 had such features as the recursive procedures. From reading the report om the language this was not quite obvious for us, and after that we, of course, recognized that fact and then we decided that in our compiler there would not be such thing as the recursive procedures. After this compiler was written, and when people began to use it, many new features of that problem have been recognized. For example, such things as debugging means and error messages should be added to this compiler. And there is another thing: the problem usually becomes more and more understood by the people who solve it and many additional features of the problem are derived from this. And this, of course, should be improved in the informal problem statement, or in the formal one if it exists. I do not know what one can say more about the correspondence between the program and the problem statement. It is rather obvious for me that program should satisfy some requirements, and the set of these requirements I consider as the informal problem statement.

HOARE: I think that in your picture a line should be drawn from the problem to the program.

LAVROV: No, the problem is a thing which is not in our minds. All that we understand is informal problem statement. This we can formulate in some language, usually in natural language and in respect to this understanding we can check the program. Correspondence to the problem itself...

HOARE: ...means that the program, when it is put to use, is acceptable. And the informal problem statement can be wrong, incomplete. The real test of the pudding is in the eating and not in the description.

MANACHER: I would like to take a guess that the formal problem statement will remain impossible. That guess is based on the following idea: if a problem is big then probably it has many data structures, and therefore it will be necessary in a formal problem statement to specify the transformations from one complex data structure to another. For that purpose one would need something almost certainly as complex as Iverson's language, for instance. And that gives only a lower bound on complexity that such a formal problem statment language would have to have. What I am saying is that if such formal problem statement can be given, almost certainly it will be almost always more complex than the program itself.

LAVROV: I do not know.

RANDELL: I think that at least in the case of small programs you can see cases in the other way.

MANACHER: Yes, but if the program is sufficiently large, and its data structures are not trivial, then I suspect that there will never be any improvement in specifying formal problem statement which probably will be as complicated (or more complicated) as the program itself.

HOARE: It is very difficult to guess that, but as you have showed the same sort of structuring methods can be applied to formal problem statement as one does to the program. If you could split it into subroutines then the main program may be quite simple. And the statement of the main program, as it were corresponding to the problem statement, at least for the time being, is almost all that is required.

One assumes that the problem-statement language will have a feature for introducing the definition of abbreviations, and skillful use of such abbreviations can very often help.

MANACHER: But surely the complexity of description will be comparable to the complexity of programs.

HOARE: I think this is fairly variable. One point about a program is that it has to use an acceptable amount of storage and be computed in an acceptable amount of time, which is very often the factor which makes it complicated. In such cases a formal statement of the problem does not have to be complicated, in this sort of way. One can see examples where a data structure which is very difficult to build up and to manipulate by the program, can be regarded as an implementation of an abstract structure which does not have to be complicated and whose formal behaviour can be described very simply. One thinks of something like the dictionary part of a compiler. All that you require of it is that it should give you back what you put in, and that is very easy to state formally. In practice, the actual way in which you organize the look up,

the way in which you use storage, perhaps two level storage, and extra points that you put in for efficiency, e.g. the buffering, complicate the program.

LAVROV: In fact I do not understand at all, what is the formal problem statement for such a thing as an ALGOL compiler.

HOARE: It can be made and it has been made (cf. the article by Landin on the algebraic definition).

LAVROV: I am not certain that checking whether the formal statement corresponds to the informal one is also there.

HOARE: One dreams, perhaps, of a day in which the author of the language can use the formal problem statement to state his intentions.

TURSKI: I would like to find out if I understand it correctly that the notion of proving correctness of the program depends entirely on the inclusion of certain criteria into the problem statement. That is, you define the problem as consisting of, roughly speaking, two parts. First of all, you describe the problem by structure of its data, or relations which are known to exist among these data, and then by the relations which should be satisfied by the results of this problem. This is, essentially, the description of the problem. Then you write your program which is, vaguely speaking, transformation of data into results, and you say it is a correct program iff you can prove that, given the data satisfying the relations which were in the description of the problem, it produces some results, which satisfy another relations in the problem description. Is it all that is implied by proving the correctness of programs ? Or is there something else ?

MANACHER: With Prof. Turski's point of view, is it possible that we need also something like a proof, but between the informal and formal ? Naturally it is not proof in a formal sense, but I think, that what Prof. Turski is suggesting is that something approximating "proof" is also needed between informal and formal. Is this correct ?

TURSKI: First of all, I was not suggesting anything; in my model I do not see any distinction between informal and formal problem statement. I want to simplify this picture. It is so complex. I just want to find out whether my simply-minded understanding of the term "to prove correctness of the program" fits the ideas expressed by LAVROV, seemingly supported by RANDELL.

RANDELL: We are talking about the "proof" as used by the people who have written earlier on "proofs". And these people used the word "proof" to mean "giving mathematically satisfactory demonstration" of the equivalence of a program and exactly what TURSKI said: that the results do satisfy relations if the inputs do satisfy the conditions. "I have proved" means, in other words, "I have satisfied a mathematician or myself, or my boss, that a formal problem statement and the program correspond". The question is: does the formal problem statement correspond to the informal problem statement ? Does the informal problem statement really correspond, in the common understanding of myself, my boss, my customer, with the actual problem ?

HOARE: What about the documentation ? It has the same relationship to the program, as the informal problem statement does to the formal problem statement.

TURSKI: First of all, I would like to record my complete ignorance of the existence of any problem except the ones which have been stated. It is probably a question of philosophy and does not enter the scope of our discussion, but I never yet saw a problem which would not have been stated. I would not know that there was a problem if it were not stated !

There is a variety of means to state a problem, but I am not going to be concerned with the first box of LAVROV's diagram at all. I do not understand the relation between problem and its informal statement; the thing begins for me with the informal problem statement.

RANDELL: I think we can avoid what might be quite fascinating but not necessarily fruitful discussion, by assuming that programmers usually try and help other people with their problems. They have customers. The customer believes he has some problem, some understanding of it, and makes some informal statement of this problem. There is a question of communication from the customer to the programmers. Let us just work from there and assume for the moment that informal statement is what the programmer is going to use. Though we all know that the informal statement may well

be changed when the customer, or somebody else, realizes that it is not an appropriate documentation of any common understanding of the problem.

TURSKI: In this connection I entirely share HOARE's point of view: let us hope that we should use the same language in informal and formal problem statement, that we shall state our problems in such a way, that it will be possible to express, in a unique fashion, those relations which we intend to have in the solutions, and which we assume should exist among the data.

RANDELL: I would like to have such a hope, but I do not believe in that one. The LAVROV's comment, that for the really interesting problems, one barely has a decent informal problem statement, and the chance of having initially a worthwhile, acurate, complete formal problem statement is negligeable. Very often it is not quite known what a program should do, how to find out what is possible, and what is not possible, until somebody is a long way in. Any statement of the intentions of the program would be so vague as to be far from being something which can be formalized and checked. It would be something like "build me an operating system which would satisfy 90 % of our different installations and will capture 70 % of the computing market". Now you formalize that !

STOLIAROV: It looks like this flowchart on the blackboard is not complicated enough to confuse everybody. Is it true, that we are trying to prove the correspondence between program and problem ?

LAVROV: In my presentation I reviewed the contents of papers published so far on the problem of so-called "proving correctness of programs", and I indicated that the methods suggested therein can be used to formalize only a small part of the mental process of getting satisfied with your program. In reality, when we are writing a program we strive to achieve a correspondence between program and problem, and not only between program and our understanding of the problem, let alone formalization of the problem.

STOLIAROV: I want to belittle the role of such proofs even farther by one step to the left, saying that the original thing is not the problem, but a certain goal which we would like to achieve, and only afterwards we have problems and so on. For example, to clarify this point: the goal may be to increase the income, and then there is the number of problems to be solved, and we should also consider procedures which transform the data possessing given qualities into results which should also possess required qualities.

MANACHER: Suppose you take the view that the formal problem statement is an interesting possibility for specifying a complicated problem, as it may be. Let me give an example for the problem of going from informal to formal, which I think is, perhaps, a very important part of this process. Suppose, in my informal problem statement I say: I want to take a list of unordered objects and I wish to extract the median object, the object in the middle and, furthermore, I want to extract it in linear time. Several questions arise. First: is the requirement of linearity of processing time to be expected in the informal problem statement ? Certainly we do not expect to see it in the formal problem statement. But if we expect it in the informal problem statement then, clearly, there is a problem of demonstrating that the property of the problem specified in the informal problem statement is in fact satisfied in the formal problem statement. Secondly, if we admit that the statement of property may be in the informal problem statement, we may need very much of algorithmic thinking and proof procedures in going from informal to formal. And indeed it is not trivial to find the median in an unordered list in linear time, although it can be done. But in order to do it, you need a moderately complicated program in the formal problem statement, and then you notice the difference in complexity. Informal problem says: find median in linear time. Formal problem statement gives a rather complicated algorithm, which is necessary. In this situation that check becomes very important.

TURSKI: I should like to say that it really does suit me very much to restrict the problem of proving correctness of the programs to this little part.

The programming languages which we use today tackle the programming business from the bottom-up. That is you first specify all the little constituent parts, you declare your reals, your integers, then you declare your procedures and so on, and only afterwards you draw the total overall picture: the problem itself. Of course you may object - no one prevents you from

writing in the opposite way, from starting with the program and then adding at the bottom of it all the declarations and specifications and the procedures which you had. You may use the layered programming approach, proposed by Dijkstra, whereby you first consider a very powerful sort of machine, which can do things like generate prime numbers and skip to the next prime integer and build your program in terms of that machine and then you look which other machines are needed, which are the specifications of each step you are taking in the overall strategy of solution and you fill it with more detail and so on, so that you descend from the general description of the program to fine detail. But, nevertheless, in this approach you always ask for specification of every detail.

What I am going to say now is sort of publicity for deferred declarations of much, much more generality than in few odd places where they appear in known programming languages. Namely, if you think about the way in which our minds work, you will discover that you specify not always, but only when the specification gives you something useful. When you talk about a number, you do not specify that it is integer or real or complex, unless you are going to draw a conclusion from the fact that it is integer, real or complex. As long as it does not matter, as long as it is immaterial, and in a certain human data processing it is immaterial whether the object is a real number or an integer number, you do not have to specify it. That is,you specify only when you are going to use this restriction of the generality. Well, you can extend this and say that as long as we deal with elements of a very general algebraic field, you do not have to impose the restrictions to select more narrow structures, unless you are going to use the fact that they are so restricted. The same, I think,is true for the data structures. I am now consciously forgetting all the sordid details of representation of data structures in the computer.

Thinking about the mental processes,I claim that the precise specification of data structures is something alien to our thinking, to our problem solving capabilities, and that we start to think about the actual structure of a data assembly only when we are going to take an advantage of the specific structuring in this data agglomerate.

What all I have said has to do with the picture on the blackboard, or with the subject of our discussion ? I am thinking about programming languages in which you would program rather differently. We came across the idea of such languages when working on properties of a language which might be useful in writing operating systems. When you start writing things like operating system, you discover that this nasty preponderance of detail which is present in the programming languages like ALGOL or FORTRAN is really maddening, because when you are writing operating system, you are most interested in certain sequencing of events, of certain phenomena which are much larger and whose internal structure is, at the moment of writing the operating system, entirely immaterial. Many people think about operating systems without interrupts, and I would sympathize with them. Just let me use an example: When you are writing an operating system for a present-day machine which does have interrupts, you really are not interested in what is the reaction to an interrupt, you just think about a unit of action, which is a reaction to the interrupt, and postpone to a later stage filling-in details of what is being done, what useful work is going to be performed by the reaction to a given interrupt. In general, you are much more interested in the fact that it is a unit of action with certain properties. And immediately you will reel-off the properties of this piece of useful work, CPU processing done on the reaction to the interrupt, without really knowing what is going actually to happen inside of this capsule of action. You know that you have to preserve and then restore the status word or whatever you used for that purpose, your registers, you have to stop execution of this, to branch off to that or something. These things you know because they are invariant, they are the features of any interrupt reaction.

I think about the programming languages of the following sort: an executional program consists of a number of processes, and each of them consists, in the simplest form, of three parts: A procedure of the process, viz. a piece of code, or whatever you have to describe the actual work. It may use a number of external variables and then create a world of its own, but this has no influence on the external world. And there are two sets of conditions attached to each process, one set of the necessary, and another of the sufficient conditions. These conditions are state-

ments, written in any logically acceptable form, using as variables objects which are either global variables in a given program, or which are predicates - and I have forgotten another thing: a process has a fourth component - its status, and now if you would ask what a status of a process is, I shall answer that when you describe a process, you specify: this process may be in state 1, 2, 3, 4, just list the possible states for each process, and each process at any given time is in one of the states. The sufficient and necessary conditions are statements in which may occur global variables of the program and predicates involving the states of other processes. Together with this thing, you have an analogue of transition matrix in which you will list the states of all processes in your system. The sufficient condition is a condition under which you execute the procedure of a given process - that is the relation on the data which have to be satisfied in order to activate your process. So you have both the structural description of data and the relation between values of these data. The sufficient condition may also include, for synchronizing reasons, the predicates on states of other processes. When the procedure becomes active, and as long as it is active, the necessary conditions are satisfied. So a necessary condition is a set of statements which are true while the process is being executed. These may be statements like: "CPU occupied", or they may be things like: "we have X which satisfies such and such relation"; as long as we are executing this procedure, this condition is true. Now if we have our programming language, structured in this fashion, it is quite easy to prove that, given the overall structure of data, and overall relations among the data, you obtain certain overall general effect.

KUROCHKIN: I do not understand why there arises a question of correctness of the transition from a problem to a formal statement. There can be a point of view that as long as problem is not formulated, formally it does not exist. Nevertheless, if someone prefers to separate two objects: problem and formal problem statement, then I think there is no possibility at all to prove the correctness of the transition from problem to formal problem statement. As far as the proof of the correspondence of a formal problem statement to a program is concerned, I am more optimistic and I think that such a proof could be established, but this is not a simple task. If the formal problem statement is written in a high level language, for example in ALGOL or FORTRAN, then it would be virtually impossible to prove the correspondence to a program written for this algorithm heuristically. Therefore, the question of the equivalence of the program to the formal problem statement may arise only if the program is obtained through some kind of automatic compiler.

The question is how to obtain an automatic compiler. It is a program obtained from a formal problem statement for this automatic compiler, therefore you have to have a formal description of the compiler. And from this formal description of the compiler, you wish to obtain the automatic compiler but this can be done through a compiler-compiler, which at this moment can be thought of as written heuristically. But if you have a heuristic compiler-compiler, you can write a formal description of compiler-compiler, and apply heuristic compiler-compiler to formal description of compiler-compiler, and to obtain an automatic compiler-compiler. Then you can apply automatic compiler-compiler to formal description of compiler-compiler, and you should obtain the same automatic compiler-compiler. Let us assume that we are at this stage: if you obtain the same automatic compiler-compiler, it does not yet prove that the formal description of compiler-compiler is correct with respect to obtaining, from formal description of a compiler, an automatic compiler.

LAVROV: To continue our discussion, may I recall that the formal problem statement may be a statement of an unsolvable problem. Moreover, the problem may be quite solvable but either we do not know it or we know an algorithm solving the problem, but the algorithm converges too slowly, and we cannot use it in the program. And it may happen so that it could be formally proved that there exists no faster algorithm for this problem, and for the practical purposes we should write another program. In all these cases the formal proof of correctness of the program, of that the program actually solves the problem, does not exist. Then we may prove that the program solves the problem under additional conditions only; on many occassions that is sufficient for the users.

KUROCHKIN: The formal problem statement, is it not an algorithm ?

LAVROV: Of course, not. These are the conditions imposed on the initial data and on the results.

KUROCHKIN: Any program is just an expression of an algorithm, so how can you have programs without algorithms ?

LAVROV: No algorithm is necessary since you have got problem descriptions from which you can derive programs directly. I have never intended to convey the impression that the program in my reasoning is necessarily a machine code program; it may well be a program in any programming language.

RANDELL: If the program could be derived automatically form a formal program statement, then this would render entirely valueless any idea that we would get "proofs". I think the important thing here is that we have a program which describes an algorithm in one way and a formal problem statement which describes the results in another way, and because we managed to satisfy ourselves, that these two are equivalent, we have a greater hope, in fact, an implication that both of them correspond to the informal problem statement. In other words, there is a redundancy. For example: if I wish to make sure that you understand something, I shall probably tell it to you in several different ways. The important thing is that there should be redundancy between the formal problem statement and the program. The programmer has had to generate by himself, separately, the formal problem statement and the program. You want to say that he has had to say the same thing twice, in two different ways: (1) indication of how to get some results; (2) an indication of the relation these results should bear to the input. That is a form of redundancy. If he manages to show that both of these are the same, he probably feels rather more confident that either of them, and therefore both of them, correspond to his informal problem statement. LAVROV has given two examples of the use of redundancy: one, in which you have a formal problem statement and a program, where before running the program you give a proof, a formal proof that they are in correspondence. The other was the example of the program for the common divisor, where he suggested that instead of giving an a priori proof that the algorithm gives the common divisor, you put some redundancy into algorithm, so that it checks itself on every run. So all we are talking about here is useful types of redundancy, and what we can do with them. If we do not have and would not have the redundancy, if the program was generated automatically from the formal problem statement, then at least in this aspect we would have done nothing to increase our faith; and our customers faith, and our bosses faith that the program is in fact in correspondence with the informal problem statement.

LAVROV: Yes, but my point is, that in general case the formal problem statement and the program are quite different things.

HOARE: I think you have made one mistake. You regard your additional conditions as an additional burden. In fact, the discovery of additional conditions is of very great importance, because they deepen your understanding of the program and its limitations, and they warn you about things that may be implicite and that you may have never realized before.

LAVROV: Yes, but actually these conditions have no relation at all to the problem itself.

HOARE: But it is better to realize these conditions than to forget them. Supposing I write a subroutine which I claim to solve that problem that you put on the board, and you used my subroutine and you find that it is very far from solving that problem.

LAVROV: Yes. But if additional conditions are used then the proof may be only of a very limited value; we are sure that under these conditions the program solves the problem but it may be so that it solves the problem in some other cases, too, only the proof says nothing about it.

HOARE: Of course, this may happen. Let me take an example: the Wilkinson's method for inverting of matrices. His analysis of the situation demanded in the end that the statement of the problem, formal or informal, was changed in a very unexpected way, before you could prove the program. But this present statement of the problem of inverting a matrix really seems perhaps even better than the original problem, that might have been informally stated. The original problem statement might be to find a matrix M which, when multiplied by X, produces the unit matrix I. The way in which he formalized the statement of the problem, is: find a matrix M which, when multiplied by X', gives I, where X' is nearly equal to X, and he defines also "nearly equals": every element of X' is within the range of the floating point representation of X. With that new statement of the problem he can prove the correctness of his algorithm. With the old state-

ment it was not possible. Obviously, we cannot all guarantee to be Wilkinsons. Most programmers would not be capable of doing this, I think it is not too early to say that it is going to be very difficult. But I think it is a bit early to say that it is going to be impossible.

LAVROV: You choose examples you like;and I choose examples which I like. Wilkinson's problem is nearly solveable; it is solveable in pure theoretical sense, and hard to solve in practical sense. On the other hand, many engineers need to solve such a problem as to find a maximum of any function of many variables, with some constraints on it, and a maximum of a functional, and so on. And this problem, in its general form, is quite unsolvable one in pure theoretical sense. People do try to write programs, solving unsolvable problems. It is very essential in the programming practice.

HOARE: First of all, I would suggest that you might try to reformulate that problem in the manner in which it would be solvable. It can be done.

LAVROV: My answer is that this re-formulation of the problem makes it quite different from the original one. And many cases are unsatisfactory from the practical point of view; the additional conditions usually are very hard to check.

HOARE: Let us take an example. I can write you a maximizing program, which will work very well for uni-modal function.

LAVROV: I am only claiming that no program solves the problem in its full generality. And the proofs that I can find are of very limited importance.

OLSZEWSKI: Returning to the original LAVROV comments, how to recognize whether something is a program, or a formal problem statement ?

LAVROV: I do not know.

OLSZEWSKI: It means that a program is a formal statement.

LAVROV: One of the points mentioned in my presentation was that in some cases the program may well be a formal problem statement, and in that case there is no need for any kind of correctness proof. It is in fact the expression of one's understanding of the problem. Some examples were given for programs evaluating the factorial function. In my understanding, each of these programs is just a definition of this function, at the same time it is a program in LISP or in ALGOL.

TURSKI: I would like to paraphrase this: given the program, I can always define the problem.

LAVROV: No, not always.

TURSKI: Taking the program as the definition of that problem, I can do it always.

LAVROV: Sometimes the formal definition may have such a form which no one would like to consider as an algorithm. And sometimes the formal problem statement may very well have a form of an algorithm. It depends on very many circumstances.

HOARE: Would you, LAVROV, say that it is a bit silly for a mathematician to put formal conditions in his theorems ? Is this your point ? There are many mathematical theorems which depend on quite complicated additional conditions for their truth to be provable.

LAVROV: There is a non-constructive proof that every function achieves its maximum in the interval. This proof is quite useful from point of view of mathematics, but has very little use in practice.

MANACHER: Of course, there are simple unique answers to problems which were never posed; for instance, suppose I give you a number and that it represents some class of Turing machines, and then I ask you to give me the number less than or equal to that number, which is the number of those Turing machines which will halt. Each one of Turing machines is a particular configuration, some of them halt, some of them do not, and there is a definite number of such that halt. So I say: "Find such Turing machines that halt". Just some three innocent words: "find", "such that", "halt". My point is, that obviously you cannot do that, I suggest that the formal problem statement must be built of primitives which are composites of Turing machine operations, otherwise it is just not a good formal problem statement.

HOARE: I should like to suggest a minor fault in LAVROV's paper. The gentlemen named do not lay so much emphasis on formal proofs,

particularly NAUR and DIJKSTRA are not at all keen on formal proofs, and I think, the other people are also quite willing to admit that formal proofs will not be easy to obtain.

MANACHER: LAVROV, could you answer the question I asked before: in your informal problem statement, are there also appearing properties of the program ?

LAVROV: The whole informal description of the problem represents certain description of the properties of the solution, the relations between the initial data and the results we would like to obtain.

MANACHER: Let us take another example: Find a median of a set in a linear time. "Find a median" is certainly the informal problem statement, but "in linear time" is a statement about the operation of the program. And so I want to know whether this statement belongs to the informal problem statement.

LAVROV: What do you mean by "linear time" ?

MANACHER: You have n objects, and the time that it takes to find a median is proportional to n.

LAVROV: The "linear time" part is an addition to the problem statement; undoubtedly in many cases it is reasonable to include things like that into the problem statement. I would consider that as a part of the formal problem statement. Are you satisfied with the answer ?

MANACHER: Yes, but I would point out that once we include such statements about the properties of the program, we are faced with major problems in going from informal to formal statement.

LAVROV: I would like to say also following. I think it is not accidental that for over a decade the programmers were satisfied with normal, generally used debugging aids, and did not need any formal proofs. The lack of desire for formal proofs reflects certain features of human psychology and programmer's approach to their task. It would appear that, apparently, majority of people do not care for mathematics as one of the most important tools for exploring the world and solving problems. I would like to quote a remark of a great American physicist Richard Feynman, who considering the relations between physics and other sciences, said: some people would be surprised that I do not say anything about the relationship between physics and mathematics, that is because I do not consider mathematics as a science. Such point of view exists and can be easily understood.

STOLIAROV: I wonder in which year and which academy of sciences is going to be the first one to make a resolution that it is not going to consider any proofs of correctness of programs.

LAVROV: I think you expect too much of academies. In many cases the formal proofsexist. They are very useful, they give much information, much orientation to the user. I am sure that they are very useful for our understanding of things. But there are many other cases when the formal proof either does not exist, or has no importance.

WORKING SESSION IV : THE USE OF HIGH LEVEL LANGUAGES IN LARGE PROGRAM CONSTRUCTION

C.A.R. Hoare prepared the following two these for this Session.

Thesis No. 1 : Programming languages are little help in the construction of large programs.

1. To design a "language" as part of design and implementation of a big system is essential.
2. To "implement" this language is disastrous.
3. To use a language designed and implemented for any other special purpose is of doubtful benefit.

Thesis No. 2 : For big new problems we need a small language reflecting closely the characteristics of hardware.

1. "Readable" version of assembly code - but still needs documentation.
2. Convenient notation for efficient operations.
3. Clear description of structure and purpose of data.
4. Strict discipline gives good chance that compiler should detect coding, punching and reading errors.
5. Option of run-time check against coding error.
6. No run-time system.
7. Compilation at speed of input.

HOARE: In Session III you heard that the proof of programs offers no assistance in the checking the correctness of large programs. In Session II you learnt that there is no systematic design methodology for writing large programs and the only recipe for success is to have been successful before. Today I am going to tell you that there is very little help that can be given by designers of high-level programming languages.

The basic thesis is that, in spite of all the work that has been expanded on the design of general and special purpose programming languages, the role they play in the design and implementation of a large system is at best minimal. At worst, the use of an inappropriate language and implementation can easily be a disas-

ter. The best you can hope for is that the programming language
is not an active hindrance to carrying out your tasks. I can re-
count an experience where a tender for a contract specified that
a certain language should be used in writing the programs for a
large real-time system. The effect was totally disastrous, and
in the end the whole program had to be written in machine code
and a pretence had to be made that it has been written in a
high-level language; it could be extremely expensive for the ten-
dering company. The reason why the role played by the language is so
small is that it only covers the transition between the two major
phases of a program design and implementation, that of the
analysis of the problem, and the decision taking involved, and
the actual start of testing the program. It is only a transition
between these two stages, that the language in which the prog-
ram is coded is of any relevance and the language can give -
and must give - very little assitance in the overall design stage.
There is very little that can be done to help by means of a prog-
ramming language in these stages. But this is rather a paradox,
because - as I state - in fact, in the design of any large prog-
ramming system the major concern of the designers and implemen-
ters is to develop a working terminology, a working conceptual
framework within which they can discuss the various decisions
that they have to make and various means of implementing those
decisions.

I heard from Mr. Maroński that it was impossible to make a
progress in the design of SODA until a series of seminars had
been organized at which the design team discussed together and
worked up a common framework of ideas in which they could con-
duct their future designs. And this method of starting a new
problem by a series of meetings at which some common, conceptual
approach and language are hammered out, had been used elsewhere
for the design of programs in new and difficult areas. Mr.Brinch
Hansen, who developed an operating system for the Regnecentralen's
R.C. 4000, said that he spent more time on these meetings and
seminars than on the actual coding of the programs that resulted.
So it does seem as though the development of some sort of
language for expressing the concepts which arise in the implemen-
tation of a large program is a necessary condition for the success
of the project. On the other hand this language which is worked-

out, an informal language of course, does not seem to be suit-
able for direct implementation on computer.

Those projects, in which the designers have attempted a
special purpose programming language as a preliminary to the
development of a particular programming system, as far as I know,
have never succeeded. The designers have become too occupied
with the problems of implementing the language and they very
seldom ever get back to the original problem or application
which gave rise to the language. A classical example of this is
the language LISP. It was invented by McCarthy as a means of
programming a very difficult project which he called the Advice-
Taker. The language LISP has had a long and independent history,
but, as far as I know, has never been used to program the Advice-
Taker.

We would like to know why the attempt to implement languages
is fatal to the success of a project in a new area. I think the
reason is that in order for a conceptual framework to be success-
ful it must be very related to the problem area and to a
large degree highly abstract, i.e. far from any mechanically
realizable implementation. The reason why the informal language
of problem description is useful, is because the implementers
realize that it is not a programming language, and they realize,
by the fact that they have to construct the code afterwards, that
the success of the project depends on certain grave restrictions
on the generality of the "why" and "which" the concepts are im-
plemented on the machine. They know, or they attempt to insure
that for a particular application, they can implement the con-
cepts in a highly non-general way, taking advantage of particular
acceptable restrictions on the generality and flexibility of the
application. When an attempt is made to implement these concepts
in their full generality, in the manner which is hoped to be
suitable for all applications of the language, the implementa-
tion becomes disastrously inefficient for any large scale use of
the language.

The language that appears to be excellent for programming
small problems which only take a few minutes anyway, applied to
large scale problems exhausts the resources of the machine very
quickly. And of course, by the time the language has been im-
plemented, it is such a large task, that there is no hope of

really changing either the language or its implementation. Thus the designers of the system have lost the vital flexibility of going back and reconsidering decisions taken many months previously, and adapting them in the light of a deeper understanding of the application and of the programming problems.

That is my attempt to explain why the implementation of special languages as the preliminary to the implementation of large systems is almost invariably disastrous.

I can dismiss perhaps more quickly the idea of using a previously existing special-purpose language. As we are talking about applications which are reasonably novel, it would be unlikely that previously designed special purpose language, or its implementation, would be suitable for a new problem area. Too many decisions about the "why" and "which" of the language features would be taken by its implementer to make it suitable for use in an area which the implementer had not thought of.

Having dealt fairly thoroughly with the negative side of my thesis I would like to proceed to the positive side.

What is it that a programming language can do to assist a designer and implementer of a large program ? I think that, in spite of my remarks, there is a great deal that can be done, and there is a very great difference between a suitable language and an unsuitable one.

My first thesis is that for large problems we need a fairly small language, a language which reflects closely the machine characteristics which are at the disposal of the programmer. Because at least this is safe, at least this is at the bottom, from which no programmer can ever escape, however high-level language he uses. If a language reflects the machine characteristics, it is possible to express in it everything that the machine can do, in a way, which achieves any desired degree of efficiency. As soon as you must, or attempt to hide the actual nature of the machine, you run a risk that you have reduced the flexibility of the system for the users of it. And although in many cases this may be a distinct advantage, the risk is extremely high that you may have reduced the freedom of the implementer to use the machine in the only way that would have made the application successful. And certainly, for almost any method of hiding the characteristics of the machine, one can think of

applications, large problems, in which this masking would lead to difficulties, if not to failure.

So really all we can do, is to attempt to do something a little bit better than an assembly code which is a little more readable than what we traditionally may have today. What it tells the machine to do is the same as that which is expressed in the assembly code, but perhaps, it is slightly more pleasant to read. And this can be achieved by the use of nesting structures, very largely by the use of mathematical symbols rather than machine function names, but the amount that can be done; is rather limited. I think that the advantage of even the best known notations over assembly code is by no means as great as the advantage of assembly code over octal code. We cannot, again, make such a big improvement in a way in which we express our programs, but we can do something.

The second point is that the notations must be chosen with care to reflect those operations which are efficient on the machine. You must attempt by all means to insure that it is not possible to invoke inefficient operations by means of a brief and convenient notation. If operation is going to be inefficient, then the notation for invoking it should also be clumsy. This, I believe, is a major fault in the use of macros, which is - I know - extremely fashionable. Once you have constructed a macro, perhaps the code which it generates is 3-6 instructions long, perhaps even invokes a closed subroutine, and then the programmer is encouraged to think of a macro, as of a single machine operation. And while his program is still quite small, this is a very useful simplification for him. When his programs, in conjunction with the programs of his colleagues, begin to get large, then the design may collapse as a result. Here I can bring my own experience of a large programming system which was attempted to be implemented on a machine with rather small and unextendable core store. The nature of the problems was almost completely masked until the last minute by the fact that each programmer individually has been using macros, and therefore had little idea, and little control over the length of the code that he was producing. The result was that when the whole system was attempted to be loaded into the machine it just could not be loaded. My suggestion is that the convenience and the brevity of the notation should reflect, to some extent, the efficiency and convenience of its implementation on the machine, so that the coder retains as it were a finger-tip control over the quality of the code that he was producing.

Third point is that the language, I think, should pay far more attention than is usual to the methods of describing the structure of data. This is an extremely difficult area where there are a number of problems. But after a certain level it is possible to have efficient means of describing the primitive, simple data structuring concepts which are adequate for many purposes, and out of which more complex structuring methods can be built. To go any deeper into this point would involve a rather lengthy exposition of the - what I believe to be the basic concepts in data structure; but in my view the most important aspects of data structuring do not demand elaborate schemes for dynamic storage allocation, garbage collection or any of these sophisticated techniques, and that it is possible, roughly speaking by copying COBOL, to single out a very large class of data structuring principles which do not involve the use of elaborate storage allocation systems.

The fourth point is that the best advantage that can be given by the language is an early detection of the coding, punching and reading errors, that as far as possible, errors of this sort should not be permitted to exist in a running program. Coding errors of the sort that can usually be detected by somebody else reading your code, should, as far as possible, be detected by the compiler. This can be achieved only if the notation for writing the program contains considerable redundancy and there are extremely few default options, very few automatic conversions and very strict discipline on the way in which the program is written. Even so, of course, there are many errors of coding nature which cannot be fully checked by a compiler. An obvious example is the use of an index which is larger than is envisaged in the array which has been indexed; subscript errors in ALGOL can only very rarely be detected at compile-time. And yet to detect such an error in the running program would be to contradict many of the other desirable characteristics of the language and its implementation, it would involve the compilation of extra code and inefficient operations. And therefore

I see no solution to this problem with present day machines, except to make the checking of subscripts and of similar construction an optional feature at run-time. This is a very dangerous thing to do. It is like the man who carefully keeps a fire-extinguisher in his car, as long as his car is stationary, and takes the fire-extinguisher out of it, whenever he goes on a trip. Because, after all what does it matter if you have a subscript error when you are testing your program, no money and no life depends on it. But if you have a subscript error when the program is in productive use, then this may be disastrous, that is when you need the checking. So I think it is very unsatisfactorily that the feature such as checking of subscripts should have to be done at run-time and by a special option. But I do not know any solutions to this problem and I am fairly certain that in most of present-day machines there is none.

I left some very important points to the last, an absolutely vital point is that the language should not involve the use of any standard subroutines which are used at run-time. A standard subroutine, firstly, it can only be a restriction on flexibility of the programmer's use of the computer. It can only be constructed if the man who writes the standard subroutine believes that he knows something about the way in which the programmer is going to want to use the computer. And if - as we have supposed - we are dealing with new and difficult applications, it will be extremely unlikely that any guess in advance is going to restrict the freedom of the implementer in the right way. And the second point is even more obvious, that if our language for programming of large programs is successful, it can itself be used to write any subroutines which are required. And to build-in to the language any standard subroutine whatsoever must be an admission of failure in the design of the language. So I think that the principle of no run-time subroutines must be accepted as a guiding principle in the design of language intended to be used for new and large programs.

Finally, there is a point which I have considered important for many years, and which is extremely unfashionable. I believe that benefits of a fast compilation are such that those who had privilege of experience in them, will never wish to forgo, the benefits of that your programs can be rewritten at any stage without extra penalty, that programs can be modified. You can avoid problems of a linkage, editing and most of all, you can get a fast run, a fast test of your program, even on a machine, that is being used by many other people. Without fast compilation it is not possible to promise fast turn-around for program testing purposes. And anyway, compilation of a decently designed language is an extremely efficient and fast process, we know how to do it extremely fast; why do we not pass on the benefit of this knowledge to the users of our languages.

My reason for believing that it is possible to design a language with these characteristics is that I have the privilege to have inside information of the progress of work which is being carried out by Nicolaus Wirth from Zurich. He has already designed a language, known as PL 360, which exhibits many of the characteristics which I have postulated as desirable in a language of large systems. The only problem is that it does not contain very powerful means for describing the structure of data, nor it permits the option of run-time check against coding errors. But apart from these two factors, it seems to me to satisfy all requirements that I have stated. The present efforts are devoted to an attempt to remedy these two deficiencies. And he is developing a language which he calls Pascal, which seems to me to have a good hope of showing what can be done in this area. It is different from PL 360, in this that it does not give direct control over the use of machine registers. This is due, I think, partly to the characteristics of the machine which he is implementing it on, CDC 6500, which has such a peculiar register arrangement that it is almost impossible to make it look like a conventional assignment-oriented language. Partly this is due to other circumstances, and other requirements which he wishes his language tu fulfil. At this point it is relevant to notice that the CDC ALGOL compiler is almost unbelievably unusable, extremely inefficient.

RANDELL: My view of PL 360 is that it is as a high-level language as you can afford on a machine as inappropriate as System 360. System 360 is just too inhospitable to really high-level languages. Machines which are more suited to - shall I say - more high-level languages than System 360, allow to provide a language, which is small from the point of view of the difficulty

in writing a compiler, and the size of the necessary run-time system, but which is from the point of view of human convenience really quite large, certainly quite large compared to PL 360.

Secondly, you talked about using a language which already existed. I think I should point out, or only remind you, that a considerable proportion of the programming for one large system, which I regard as successful, the project Apollo Ground Support was in fact done in FORTRAN; these were mainly the separate modules, some of them quite big, for analysing data, not the very basic part of the system. Taking that system as a whole, I am thinking of that group of 300 programmers as a whole, a very large proportion of what they were doing was in fact in FORTRAN.

HOARE: Well, FORTRAN is a language which is fairly close to thesis No. 2 on the blackboard, and of course, the reduction of data, I think, is largely a numerical application, so this is not as it were an application wholly unenvisaged by designers of the language.

RANDELL: Third point. You were saying that a language should not have a run-time system. There is one very large run-time system that a very large proportion of today's programmers are using unwittingly. It is called the macro-programming of System 360. If they insist, or unthinkingly accept that they will not program the actual hardware that IBM is produced, but instead rely on a very big and general, and probably quite unsuited large language, with a very big run-time system, called System 360 Machine Code, then they will in fact have lost the possibility of a considerable increase in speed and in efficiency. Clearly, I am not really arguing that they should do that, and I think this is at least in part, a criticism of your comment about no run-time system. I think you are making too big a distinction between hardware and software. It is necessary for some group of designers to decree a level below which the vast majority of programmers must not be allowed to penetrate.

HOARE: Of course, speaking of environment of OS 360, the statement that there is no run-time system is somewhat misleading. Obviously, when you have many people using the same computer and submitting jobs at different times, some sort of operating system is required to enable the computer to be efficiently shared among those people. And this is a necessary evil, which programmers must be told to accept. But sharing the bulk of the computer is no excuse for the language to impose additional run-time system.

RANDELL: I have a question. You were saying that you did not wish inefficient operations to have a convenient notation. You criticize macros; I assume, that you also, to a certain extent, criticize the subroutines, or at least you would like to distinguish between the ease with which we invoke a simple subroutine and the ease with which we invoke a very large subroutine. Are you, for example, suggesting that the number of letters in the identifier of a subroutine should be related to the length of time it will run, or something of this sort ? If not, what suggestion do you have ?

HOARE: I suggest that subroutine calls should be very clearly marked off.

RANDELL: Just as being subroutine calls ?

HOARE: Yes, I think mainly about the small innocent looking evils. For example, the macros look so tempting and involve only a small overhead on each use of them, and in the end clog up the code to such an extent that it is almost useless. I am not against macros absolutely, either. In fact, every principle sometimes has to be bent, and sometimes there is a very good reason for abandoning it, even for writing a run-time system.

TURSKI: I would like to ask you a question. I did not quite get your point, perhaps it is just misunderstanding of terms; is it true that you really would negate the use of any pre-coded parts of software ?

HOARE: I suggest that, for a language of the flexibility and efficiency which I am proposing, it should be possible to code any required piece of software. Therefore, there is no need to import this into the language. This is not absolutely true, however.

TURSKI: For instance, things like sine subroutine ...

HOARE: Why should you build it in ?

TURSKI: Should every programmer be obliged to provide an own sine subroutine ?

HOARE: There is such a thing as a library.

TURSKI: I see. You do not negate the usefulness of libraries.

HOARE: The libraries can be written in the same language.

There is one feature that I feel can be deserving of a run-time system in a strictly nested fashion. This is the administration of a stack. PL 360 does not even have that, but in Pascal the decision was taken to permit the use of recursive subroutines.

Any principle that I have put on the board, does not absolutely dictate what features the language is or is not to have. Particularly, when - as I suggest - the remarks should be interpreted with a small degree of flexibility.

The inclusion of a stack administration to permit programming of recursive procedure is well justified in those cases where the problem to be tackled has a recursive structure. And I know of no way of programming a stack administration decently, even in a quite low-level language. It seems to me that it has to be built in. On the other hand, if one has an application which requires storage management schemes very much more sophisticated than a stack, even the tiny, little code that administers the stack can be highly inconvenient. So this is a genuine choice to which I give no absolute answer. There are some applications in which we do not wish to have a stack built-in, some in which you do not mind, and some in which you will welcome it.

TURSKI: In the operating system SODA, as a programming tool we use a language which has a very curious origin; it is entirely machine-oriented. Its primary purpose was to start working on development of the system for a new computer while still using the old computer. So first of all, it was a language which permitted to simulate one machine on another, a bootstrapping language. And therefore it has many of the features which you listed in the thesis No.2, except any checking facilities, and we regretted this very much later on when we actually started using the language for producing the software and not only for bootstrapping parts and bits of it.

Because of an entirely different philosophy we did not have many facilities for complicated data structure description in this language. First of all, I do not understand "description of purpose of data" and what you might conceivably mean by this. It seems to belong to the informal problem statement - using terminology of Session IV, and moreover, I am a great believer in a very simple description of even very complicated data structures.

HOARE: I think your point is correct. The description of the purpose is not directly the problem of programming language' but of the accompanying annotation. But on the other hand it is a certain gain you may achieve by telling the computer the "what for", at least by making it look more obvious.

TURSKI: It is not for the computer. "Computer" cannot make any use of the knowledge of the purpose of data, unless you choose to use the term "purpose" as some sort of indication of when and how the data are going to be used, like indicating that from now on the program may proceed by making call on certain part of data, or alternatively, attaching to a chunk of data some label saying: "This might be useful, only in connection with such and such part of the program", so that the computer may be quite indifferent to the fate of that chunk of data at any time, except the time indicated in the label of the data. Coming back to the gist of what you have said, is it that for any large scale software project you have to devise an entirely new language tool ?

HOARE: I think there are concepts which will be common to many applications, for example, one may find that the conceptual framework of set theory can be useful in discussing many different areas. The fairly familiar theories may provide very good basis which is common to great many application areas, but probably only in the most abstract sense; as soon as you get to slightest details, the way in which these theories are used, the way in which particular concepts are defined, is likely to be different in different areas. Even more important, the way in which they are implemented, or represented on the computer is varied even more drastically, much more than the abstract use of the word "set", for example, would indicate.

Although many applications can be described using set theory, in the implementation of these concepts, and their representation on the computer, the techniques used will be radically different, depending on the frequency of different kinds of operations concerned, ways in which the data are held in storage, ways in which the storage is used and so on. Even when you have a general

purpose concept, these things are so abstract, so far away from the way in which machines actually work that each implementation of the concept has got to take into account the characteristics of the application to such a great extent that there is no such thing as general concept implementation; which is really another way of stating the basic reason why we cannot implement an application language in a general way and still have sufficient freedom to suit particular languages application.

I should like to take up a point which you mentioned earlier about checking. Of course, the idea of checking against coding errors conflicts strongly with the requirement for complete flexibility in the use of the language. There are occassions in any program, certainly in the large program, where it is necessary to cheat, to somehow by-pass the declared structure in an intent of the program to achieve some effect which cannot efficiently be achieved in any other way. I refer particularly to certain subroutines for binary-decimal conversion, perhaps, and certain methods of computing such things as square roots. They depend on your knowledge of details of representation of floating point numbers, and on your ability to treat them as integers or bit patterns and perform operations on them which are, in the abstract theory, completely unacceptable.

For these operations, it seems to me, your language must provide some means of by-passing any checking which you built in. You hope that such cheating can be confined to reasonably small number of fairly low-level routines which can be written rather carefully at quite an early stage in the design. If, of course, this is not so, you are constantly needing to make puns, as it were, on the representation of your data; then you have an application which is so complicated that really nothing will help you, except the ability to think in binary. Any language which masks the binary representation of the data is going to be something contrary to something which is going to help you. The great advantage of the symbolic language is that you can ignore, to some extent, the binary representations of the data, but one can imagine applications in which to ignore this will be disastrous.

I can give a slightly more complicated example; the Pascal compiler is itself written in Pascal, which is very obvious way of checking the viability of the system, of the design for the construction of large programs. A very large part of the compiler is concerned with error recovery, with detecting an error and continuing the checking process on the rest of the program; it is estimated that between 1/3 and 1/4 of the total volume of the system is devoted to doing something sensible after one syntactic error has been discovered. One reason for this is that the language does not support the use of labels to any very great extent, in particular, it does not like passing labels as parameters, or dealing with labels in a recursive situation. The fact is that dealing with syntactic errors can be achieved very economically if you permit the use of labelled jumps in recursive situations. But this is not permitted in the language, and therefore dealing with error situations becomes very much more complicated.

RANDELL: What conclusions would you draw ?

HOARE: I draw my usual pessimistic conclusion, that is that when I have to pay with one thing for another, then there is no design perfect for all applications. Writing a compiler is a fairly special application, the language does permit a recursion, which is a very great help in this application, but does not permit to use the labels, which is a nuisance. So, even at this very low-level, there are design decisions which are going to have very severe effects on the way in which the application is programmed.

MADEY: May I just give another example. The batch processing operating system for RC 4000 is written entirely in an ALGOL 60 representation. As far as I have understood their approach, the authors wanted first to write this operating system in ALGOL, debugged it, learned something about the problem, and then rewrote it in an assembler code.

HOARE: Is it rather extended ALGOL which includes some facilities for message passing ?

MADEY: Yes. It is very much extended, it has also facilities for record manipulation, and, of course, for bit manipulation.

HOARE: And for communication between parallel process.

MADEY: Yes. It means: for communication with the monitor system.

HOARE: Monitor system itself, which is not a trivial program, is written in machine code.

MADEY: Yes.

TURSKI: A little piece of information: the compiler for the URODA language, the language in which we wrote the operating system SODA was written in the same language as well.

HOARE: How long is the compiler ?

TURSKI: Very small. The language is simple. The size of compiler is about 3000 instructions.

HOARE: Mr. Stoliarov, what language did you use for your large system ?

STOLIAROV: Of course, we use an assembly code, but we would rather have slightly higher level. It was more advantageous to have a single language for all parts of the system.

HOARE: Mr. Kurochkin, what language are you using in your current project ?

KUROCHKIN: We use two languages. One, high-level, for discovering errors and for checking our programs, and the other, low-level, assembly language for writing working versions.

HOARE: What high-level language do you use ?

KUROCHKIN: ALGOL. We have no others.

HOARE: Why do not use LISP ?

KUROCHKIN: It is because of traditional experience in ALGOL.

HOARE: I think your approach is very dangerous, because there are many fairly abstract cocepts which are also efficient, but are not represented in ALGOL 60.

KUROCHKIN: Why not. You can have any extension, any manipulation with characters, therefore you can do everything.

HOARE: In this case you are using ALGOL 60 as a very low-level language.

KUROCHKIN: Yes. But there are procedure calls in it, and procedure calls are very convenient to use as programming tools, especially as we have no others. When we have LISP interpreter, may be we will use it, too. Because of the seventh point of thesis No. 2 we do not use our ALGOL compiler for production runs, but for small program and for debugging of programs it is very convenient.

HOARE: I am glad to have support for my seventh point.

TURSKI: Since you are collecting different pieces of information may be people from Computation Centre of the Warsaw University would tell something about their implementation of several list-processing languages in ALGOL 60.

MADEY: I do not think we can give any details now, but in fact we can have some lists written in ALGOL 60, and we have also recently got an Euler written in ALGOL 60, precisely in GIER-ALGOL IV. This second task - an Euler compiler - was done by two students having no previous experience either in programming large problems or in compilers. This was done approximately in 6-7 months.

HOARE: Of course, Euler is not a very efficient language but would you say that their implementation was usable ?

MADEY: From two unexperienced students one could not expect anything special, but it works. I do not remember any restriction on the original Euler, as far as I know, everything is included.

HOARE: What I am interested in is the size of compiler, compilation speed and, particularly, run-time system.

MADEY: Unfortunately it is very, very low speed, but you must remember that the machine on which it was done has got only 1 K core store; in fact 1 + 4, because there is a buffer which can be used in certain ALGOL representation as a core store, to store arrays, and there is also a drum.

I should like to give another example, the language BCL (Hendry D.F and Mohan B., 1968), two applications of which I actually remember; one, its own compiler is also written in BCL to certain stage, of course, and secondly, the language L^6 (Knowlton K.C., 1966) is implemented for Atlas in BCL. Implementation was done about two years ago, and the author has given quite a lot of arguments why he had used BCL for this purpose. I do not remember this, but it was published later on in the Computer Journal, Vol. 12, No. 4 (November 1969).

HOARE: I think, perhaps, that we know almost enough about the writing of compilers to be able to design a high-level language

like BCL, suitable for constructing single pass, or nearly single pass compilers for languages of simple syntactic and semantic structure. I would be very doubtful about possibility of writing a PL/1 compiler in BCL.

RANDELL: Within IBM, a considerable amount of experimentation is being done with a language which is - shall I say - nearer to PL 360 than anything else, which in fact has been derived from PL/1 rather than from ALGOL. This, certainly, seems to make it less attractive than PL 360, at least to my eyes. I believe that the tests that have been made of it, showed it to be reasonably competitive, by any appropriate measure, with the assembly language; but there are, of course, tremendous impediments to its being used in place of the assembly language: questions of compatibility, of retraining, of politics, etc.

Secondly - the operating system for the B 5000 was originally the only piece of software, which was not written in high-level language. Everything else, all compilers, all application programs were written either in a quite extensive subset of ALGOL, or in yet higher languages. The operating system, however, was written in a rather strange assembly language. This operating system was the main cause of the initial commercial failure of the B 5000. After two years, a fixed head disk was added to B 5000, one or two other changes were made, and it was renamed the B 5500. At this stage, some of the people, who had worked on ALGOL compiler, rewrote the operating system in ALGOL subset to replace the assembly language operating system. This operating system provided more facilities than the machine language operating system, it worked, was reliable, faster and used less core store. To the best of my knowledge, that part of the Borroughs Corporation has not used the assembly language since, in any of the software for the B 5500 or for any successor machine. It is no longer a point of argument. However, I should point out that the original machine and all its successors have all been designed with languages such as ALGOL very much in the forefront of the designer's mind. Bringing the story completely up to date, the on-line banking system, the one which will have 2000 terminals and a B 6500, uses the Burroughs supplied very basic operating system, again written essentially in ALGOL, adds quite a lot of new work again done in ALGOL, and includes a tremendous amount of software of a more specialized banking ma ture, all written in COBOL. There is absolutely no question of them trying to get below this level of language. I should point out: one of the differences of the B 6500 from the B 5500 is that rather more attention has been paid to the problem of making COBOL efficient. Even so, most of the B 5500 were in fact used in places like banks, and were essentially COBOL machines; even though I tend to think of the B 5500 as a university ALGOL machine, its commercial success has come from COBOL and banks.

HOARE: Very nice. I am sure that this is an encouraging news.

RANDELL: It all happened a very long time ago.

HOARE: One question which springs to my mind is that I would suspect that the amount of ALGOL actually used in the design of these systems is in practice a very little more powerful than a language like Pascal, at least for any particular application.

RANDELL: I do not have enough knowledge of Pascal to contradict you. However, I would assume you are right.

If one assumes that somebody designing a large system takes as unquestionable and unchangeable specifications of the hardware that he is going to use, then this will have a very great effect on the appropriate level of language that is safe for him to use. When we talk about really large programs, we should remember that the cost of the software is probably very high as compared to the cost of the hardware. So that at least from the point of view of choice between commercially available systems, this possibility is often open.

HOARE: There is very dangerous implication in what you just said. Many language designers have used it as an excuse for an extreme inefficiency of their implementation: since after all, programming costs are higher than the hardware costs, therefore you should buy more hardware in order to use my wonderful language.

RANDELL: Could I suggest "more suitable", not necessarily "more"?

HOARE: Yes, you can suggest that. The main point is this: any extra money that is available, rather than enable the programmers to program in less than most efficient manner, should be made available to enable them to use a more modular approach to the design of programming languages. In other words, the benefit of

having extra hardware should be passed intact to the user of the language, and not selfishly wasted by its implementor.

RANDELL: I cannot disagree, and I do not see that there is too close following from what I said to what you said.

HOARE: Is there any one else with experience of implementing difficult programs who has attempted to use, or decided not to use a high-level language ?

LAVROV: I agree with the theses and many additional points made today by Prof. Hoare. Particularly, I would like to note the whole thesis No. 2 and the first point of thesis No. 1. It follows that what we need is both: a problem-oriented language and a machine-oriented language. Apparently, the gist of HOARE's opinion is that there is not a single language which would satisfy those both conditions, and, practically, I agree that it is really the case nowadays. Nevertheless, I do not think that the task of designing a single language suitable for programming large problems is hopeless. It is well known that there are many attempts to construct a universal programming language; I am a great supporter of continuation of these efforts. All evidence given to prove the non-existence, at this moment, of such a language, and the fruitlessness of searching for one appeared to me as indication of a necessity of a new and different approach to the problem of designing such language. Even if these efforts should turn out to be unsuccessful, they will undoubtly make some impression on the programming practice and theory, and, therefore, they will be useful.

I should like to describe certain details of an approach taken in the Computation Centre of the Soviet Academy of Sciences in Moscow, where we try to define such a language. But before doing this I should like to make several comments.

Many people indicated today, that the existing languages are being used for writing large programs. It has been said, particularly by Prof. Hoare, that the actual coding, the actual write-up of a program is relatively simple and easy task. Our practice, as Prof. Kurochkin has said before, is to begin by programming in a high level language, and only when the structure of the program, or of the system, becomes clear to us, we change over to programming in the assembly language. I should like also to remind you one of the theses of my talk in Session IV, namely that the high-level programming languages are one of our mental tools in thinking and working on problems.

Now I should like to describe some details, some salient features of a language which we are developing, which has not yet an official name, but is provisionally called an Extendable Algorithmic Language (EAL).

One of the most important starting points of developing this language were points 1 and 2 of the thesis No. 1. Indeed, there are so many different problem-oriented languages being designed all over the world, and people spend a lot of time making implementation of these languages with really disastrous results. Therefore, as one of the design goals for our language, we have chosen the facilitiation of implementation of problem-oriented languages. We think that if the term "language" is applicable here at all, then certain features common to all languages should be present in our product. The failures of the attempts to create a universal programming language are probably due to the fact that the general principles for language construction have not been yet clearly defined, or even realized. Of course, it follows from this that we hope that we have progressed in this direction somewhat further than others; it is also quite possible that we are being overconfident.

Since our language is going to be extendable, it should contain a nucleus for all possible extensions. This nucleus consists - according to our designs - of three essential parts. We call them: Base, Analyser and Calculator.

The Base is a set of general notions, present, in one or another form, in most of the high-level programming languages, for instance: action, quantity, its value, its name and its mode. The actions will have following states: initiated, suspended, interrupted, completed, resumed.

The Analyser has been included in the nucleus, for following reasons: since we are going to consider the extensions of the language, it should be possible to read-in texts in the extended languages. It should be enough to mention that the basic notion in the Analyser is the notion of string of characters, and certain very simple operations on characters: concatenations of two strings, search for occurrence of one string in another, together with delimiting the string predecessor and string successor of a given

string and, finally, the extraction of a given number of characters. We are confident that using these simple operations we can describe any analysing grammars for the language extensions.

Finally, the last part of the language nucleus is the Calculator, which at the moment causes us most problems. Its main task is to respond to the point 2 of the thesis No. 2, and perhaps to the point 1 of this thesis, too. The Calculator is, in a sense, an assembly language for a contemporary computer. For the success of our language it will be crucial whether this part of the nucleus could be efficiently implemented on specific computers.

In the Calculator, the notion of memory of different levels is introduced, and it is assumed that the smaller number of the memory level, the higher its speed and the smaller its volume. Memory of each level is divided into cells, and sequences of cells may be combined into, what we call, words. Words are used as storage units for values of different modes. We have following modes: integer, approximate number, bit sequence, address (since all layers of memory are addressable), texts, and strings (used by the Analyser). Texts are suitably packed sequences of characters.

We differentiate between the words and values represented in them. Certain operations are performed only on values, other operations are performable on words. For example, words may be transferred from one level memory to another. We may extract subwords from words, that is sequences of fewer cells than in the original word, and the extraction may be performed according to the length of the subword and its relative address.

We hope that the Base will permit us to describe the semantics of languages, and therefore will correspond to point 1 of the thesis No. 1. The Analyser will analyse input streams and we hope that the Calculator will permit efficient realization of programs written in extension of any level.

In the language, we had positively avoided any implicite coercions in the sense of ALGOL 68. It was very valuable to us to hear Prof. Hoare saying that the notion of array, used to such an extent as in ALGOL 60, may lead to gross inefficiencies; we do not have arrays in our language. To a certain extent, the notion of array is replaced by the just mentioned notion of words. In our language, we have foreseen a large number of so-called "inbuilt interrupts", which are connected with the inability to perform a particular operation. To each of these interrupts the language relates a suitable reaction, exactly as you can program usual interrupt routines. I should like to mention one example of such interruption. One of the operations foreseen in the language is the following: create a word in the memory of a given level. Should it happen that there is not enough space left in the memory of given level for this operation to be completed, the interrupt will follow.

And the other feature which I should like to mention is that the Calculator is parametrized. There are several such parameters for instance, the length of cell in bits, address length in the memory of any level, and so on.

TURSKI: If I understood it correctly, it is possible to extend the language, to build on this core system. What are the rules for the extension ? How they are specified ?

LAVROV: It is fairly simple. In the Analyser there are rules for operations transforming the strings into objects of any mode, and particularly into actions.

TURSKI: Should not there be a grammar for these operations ?

LAVROV: There exists, of course, a fully defined syntax for all the operations which I have mentioned, but we do not impose any restrictions on the syntax of possible extensions to the core language. The extensions may be done step by step; the general scheme of one step is following: program written in the extension of (N+1)st level consists of two parts, first of which is written in the extension of Nth level and is in fact a pre-processor, transforming the second part into the Nth level extension.

HOARE: Is the pre-processor written in the language of the Analyser ?

LAVROV: The pre-processor for (N+1)st level language is written in Nth level language and the nucleus is indeed 0 level extension. So the last pre-processor which will be used to translate the given level programs is written in fact in core-language. At any given time we may pass from the execution to the translation of the parts created during the preceding execution.

Few words on the state of the development. The greatest progress has been made on the Base part of the core-language. The

Base was fully modelled in LISP and we are quite satisfied with the outcome. Now this part is being coded in the assembly language for BESM-6. For the Calculator, we do not like to use LISP, it seems to be unsuitable; probably the Calculator will be programmed from the very beginning in the assembly language. The Analyser could be modelled in SNOBOL, but since this is rather new part of the language, and since we do not have SNOBOL compiler, it is being coded the assembly language directly.

RANDELL: Am I right saying that you use the word "extendable" in exactly the same sense that it might be used about any machine language ? For example, IBM 360 machine code language is extendable, and many people do extend it, by using compilers and interpreters. It has the ability to analyse strings, to generate instructions, to put those in storage and to start executing them. I do not think I have got the sense of your use of word "extendable".

LAVROV: Of course, I agree that the process that I have described is very similar to "extending" machine code by compilers and interpreters.

WYSMUŁEK: I should like to find out, whether I understood correctly Prof. Lavrov's comment that in designing a large programming system you start with programming in the high-level language and afterwards you switch to the assembly language coding. Are you using the high-level language to provide the general outline of the solution, and only having obtained that, you fill in the details in an assembly language, or is there something else ?

LAVROV: Sometimes we use a high-level language to obtain pieces of program which could be used directly, but equally frequently, or may be even more frequently, we are using the high-level language to gain a better insight in the problem, and when we can solve the problem in general way we transfer it into the assembly language by hand. Just because of this I consider the high-level languages as mental aids, tools of thinking.

WYSMUŁEK: If you re-program things written in the high-level language in the assembly language, you are faced with two questions. First, why you do not use a compiler, why do you use hand-coding ? The other question is: if you use hand-coding, how do you check the correctness of the translation ?

LAVROV: I can give you an example to show why we are not using an automatic translator for the high-level languages. Suppose we are interested in debugging a piece of program which deals with memory of our system. We may think about such memory as of a two-dimensional boolean array, say rows representing cells and columns representing the individual bits. It is quite clear that if such a program, written in a high-level language, was to be translated automatically, the resulting program would be terribly inefficient. Whereas hand-coding of the program written in the high-level language can use the insight gained, while making the program in high-level language, and in the same time be more efficient than the automatic translation. To the second part of your question I would answer that we do not use any formal means of checking the correctness, we rely on the normal means of program debugging and on our common sense. This could be done just because the preliminary phase, in which we use the high-level language, permits us to gain a valuable insight into the nature of the problem. And that is why in the machine coding we can avoid a large number of errors which would we have made if we started coding in the assembly language at the beginning.

HOARE: I find it difficult to understand the distinction between the Base and the Calculator.

LAVROV: In my opinion, those two parts are entirely different, and we use entirely different techniques in implementing them. The Base is essentially an interpretative part of the system, whereas we hope to design the Calculator in such a way that its operations could be implemented directly as operations of a computer.

HOARE: Do I understand correctly that the Base is therefore the control system for the use of the other two parts. In programming an extension step at level one, and higher levels, it is possible to use the language of the Analyser, is it not ? The normal tool for programming an extension will be the string-language rather than the Calculator language.

LAVROV: Yes, but it depends on which level we are at the moment. It is quite possible that for certain level the preceding levels accumulated anough tools, so that we can describe the next level without relying on the language of the Analyser or any other part of the core. It is not necessary that the pre-processors, which I have mentioned, are restricted to translating one string into another. They may create also objects of entirely different nature, like correspondence tables.

HOARE: I am very interested in the Calculator, because I think it corresponds most closely to the design of a machine-oriented high-level language as described in the thesis No. 2. Do you agree ?

LAVROV: I am interested in the Calculator, too.

HOARE: I believe that there are two aspects in which your design differs from Pascal. One, that it does not deal with data structures in the same way. Your data structures are very much more machine-oriented, in terms of bits and words; the Pascal's structures are defined symbolically, in a more systematic way. The other aspect is that Pascal is not intended to be implemented in a machine independent fashion, that although the major notations can be constant from one machine to another, the implementation restrictions, and certain details of the semantics are intended to be implemented efficiently on each computer, and there is no guarantee of transferability of programs from one machine to another. Have you not attempted to make the Calculator too machine independent ?

LAVROV: We have attempted to make the whole nucleus machine independent. It may be that we shall not be altogether successful, but we shall be fighting to the death.

HOARE: I believe that the excessive emphasis on machine independence of the Calculator may very well counteract many of the benefits of your overall approach and that the major part of the benefit of your approach may be lost because of this requirement. There are many large applications of programs for which machine independence is not a requirement. And it is not fair to burden these applications with an irrelevant and complicating, costly factor, such as machine independence. When it is desired by the applications programmer, he has the power, perhaps, if the rest of your project is successful, to construct for himself a machine independent language, by deciding to use only extensions which are translated in different ways for different machines, so that on each machine it is efficient, because the details expressed in the Calculator language have been adapted to that machine, but the language which is seen by most programmers, is machine independent. I believe this is probably the only practical solution to the problem of machine independence.

LAVROV: I agree with almost all what you have said. And I should like to mention that there is no unanimity among the people engaged on that project with respect to those subjects. But while I, as a team leader, am quite convinced that the decisions taken with respect to the Base are quite correct, I am not so much convinced that we have chosen the only correct solution with respect to the Calculator.

And now few words about data structures. We can construct various data structures in the following way: the notion of the quantity mode can be explained as follows. The mode is the action required in order to introduce a new quantity of that mode. This action may be arbitrarily complex; in the process of executing this action, quantities of other modes may be created and destroyed. All these quantities, created and not destroyed at any given stage of execution of a program, form what we call the set of attributes of the quantity being created. You may use this set of attributes after the introduction of a given quantity, as well as during the reaction to interrupts which may arise during this process. As you may have observed, there is no notion of procedure in our language because the things which we were just talking about replace quite successfully that notion in our language. I should like to mention in this connection Garwick's language GPL, familiarity with which has influenced our design considerably.

HOARE: Including rubber arrays ?

LAVROV: No, with respect to arrays I express my full agreement with Prof. Hoare's opinion. Speaking about acknowledgements I should like to mention ALGOL 68, first of all its pitfalls and all the criticism expressed on it during the IFIP/WG 2.1. meetings.

HOARE: If the project is to be successful, why do you not use it as its own implementation tool ?

LAVROV: It may be done, but at the moment we think it would be premature to do so. Moreover, it would be rather meaningless to describe the Calculator in terms of the Base and Analyser. A much more useful thing would be to use the Calculator for description of two other parts of the nucleus language.

WORKING SESSION V : MISCELLANEA

The first part of this Session is devoted to the presentation of Prof. Lavrov's comments on "Preliminary Scope Outline", cf. Appendix 1.

1. The largeness of programs is expressed in all their attributes: the size of program, the volume of data, the development and debugging time, the number of programmers and the running time. "Large" programs cause the same difficulties as "small" programs, but the accumulation of these difficulties calls for a carefully developed design methodology and strict adherence to it. Violations of programming laws, while only slowing down the development of small programs, make it virtually impossible to complete successfully large programs.

Large programs are not much different from other large projects e.g. in the construction, machine industry or in government of society. We should, therefore, consider both general questions of efficient development of large projects and specific questions of construction of large programs.

2. A large program should, of course, be user-oriented. Specific features of the equipment and convenience for the personnel maintaining the program should also be taken into account, but are of secondary importance.

The production of a large program consists of all stages present in any large project:

- specification
- model construction
- flow-charting
- debugging
- maintenance

Specifications determine the objectives and main requirements for the projects. The model stage comprises the feasibility study, investigation of the possibility to meet the requirements, and planning of means necessary to satisfy them. Following stages determine the program structure and interaction between its modules. Special attention is paid to the definition of data formats to be used in communication between modules. All stages should be well documented and discussed by the designers, workers and future

users. The discussion in early stages should seriously consider whether the program is needed at all.

I am convinced that there is no optimization criterion for program production, nor there could exist one. A program developed from the scratch should solve the problem for which it was designed. It is desirable that the solution obtained is efficient, i.e. fast, cheap and reliable, but (at the beginning) this is neither necessary nor realistic requirement. The efficiency and even more optimality, if it exists in any sense, can be achieved in the process of maintenance and modernization of the program.

We should realize that many immature projects are being started and developed all the time. Such projects should be considered very carefully. If they open any principally new possibilities such projects should be supported and permitted to develop so as to implement these possibilities. Otherwise the designers should find the courage to abandon the project - the sooner the better.

3. I am not aware of any precise effort estimates for large program development, or for any other large project. As a rule, more difficulties will arise than were foreseen; the difficulties almost never are imaginary.

Large programs (by definition) should and could be developed simultaneously and in parallel by many people. This increases considerably (1.5-2 times) the total effort because of the additional work required for fitting together parts developed separately; but it enables to complete the development within an acceptable time, i.e. while its goal remains unchanged.

Various network analysis techniques, like PERT, should be welcome, but a good manager should be able to produce intuitive estimates practically identical with those obtained by PERT (even though both such estimates are equally over-optimistic).

4. Automatic composition of large programs is desirable, possible and will gradually spread. Complete automation shall not be, however, achieved (unless artificial intelligence exceeds the natural one, but what should then scientists do ?).

Automation is hindered by incomplete understanding of problem solving (absence of algorithms of mental processes). If such an understanding is available it can be easily implemented in the form of a program, even in machine language. In general, the only way to understand is to apply successive approximations,

WORKING SESSION V : MISCELLANEA

The first part of this Session is devoted to the presentation of Prof. Lavrov's comments on "Preliminary Scope Outline", cf. Appendix 1.

1. The largeness of programs is expressed in all their attributes: the size of program, the volume of data, the development and debugging time, the number of programmers and the running time. "Large" programs cause the same difficulties as "small" programs, but the accumulation of these difficulties calls for a carefully developed design methodology and strict adherence to it. Violations of programming laws, while only slowing down the development of small programs, make it virtually impossible to complete successfully large programs.

Large programs are not much different from other large projects e.g. in the construction, machine industry or in government of society. We should, therefore, consider both general questions of efficient development of large projects and specific questions of construction of large programs.

2. A large program should, of course, be user-oriented. Specific features of the equipment and convenience for the personnel maintaining the program should also be taken into account, but are of secondary importance.

The production of a large program consists of all stages present in any large project:

- specification
- model construction
- flow-charting
- debugging
- maintenance

Specifications determine the objectives and main requirements for the projects. The model stage comprises the feasibility study, investigation of the possibility to meet the requirements, and planning of means necessary to satisfy them. Following stages determine the program structure and interaction between its modules. Special attention is paid to the definition of data formats to be used in communication between modules. All stages should be well documented and discussed by the designers, workers and future

users. The discussion in early stages should seriously consider whether the program is needed at all.

I am convinced that there is no optimization criterion for program production, nor there could exist one. A program developed from the scratch should solve the problem for which it was designed. It is desirable that the solution obtained is efficient, i.e. fast, cheap and reliable, but (at the beginning) this is neither necessary nor realistic requirement. The efficiency and even more optimality, if it exists in any sense, can be achieved in the process of maintenance and modernization of the program.

We should realize that many immature projects are being started and developed all the time. Such projects should be considered very carefully. If they open any principally new possibilities such projects should be supported and permitted to develop so as to implement these possibilities. Otherwise the designers should find the courage to abandon the project - the sooner the better.

3. I am not aware of any precise effort estimates for large program development, or for any other large project. As a rule, more difficulties will arise than were foreseen; the difficulties almost never are imaginary.

Large programs (by definition) should and could be developed simultaneously and in parallel by many people. This increases considerably (1.5-2 times) the total effort because of the additional work required for fitting together parts developed separately; but it enables to complete the development within an acceptable time, i.e. while its goal remains unchanged.

Various network analysis techniques, like PERT, should be welcome, but a good manager should be able to produce intuitive estimates practically identical with those obtained by PERT (even though both such estimates are equally over-optimistic).

4. Automatic composition of large programs is desirable, possible and will gradually spread. Complete automation shall not be, however, achieved (unless artificial intelligence exceeds the natural one, but what should then scientists do ?).

Automation is hindered by incomplete understanding of problem solving (absence of algorithms of mental processes). If such an understanding is available it can be easily implemented in the form of a program, even in machine language. In general, the only way to understand is to apply successive approximations, hence the desirability of means for precise description and convenient modification of algorithms and good rules for expressing comments on algorithms.

As a rule, the more we write about a subject in our natural language, the better we understand this subject. It is pretty bad when the difficulties in description mask the real subject difficulties. In this consists the role of high level languages.

Automation can help not only in processing (translation) the algorithm description but also in its composition. Therefore, heuristic methods should be encouraged. Their application should rely at least on a complete and coherent set of notions, in terms of which the problem should be solved but even this set may be absent at the beginning. This is the reason why complete automation is impossible, even using so powerful (in principle) means as heuristic programming.

Automation or mechanization seem to be inevitable when we do something not for a first, second or tenth time but for a hundredth or even thousandth time. All new paths are traced by pedestrians.

5. The conflict between modularity and efficiency is more often than not imaginary. It arises when we do not know how to make modules or do not know what the efficiency is. The principle of modularity is inherent in all technical, biological and social systems. Why should it be bad for programming ?

6. The user is hardly likely to want to change anything in a well-designed system. We do not, usually, change TV sets or writing desks. Why should users change large programs ? Quite the reverse, the designers usually are not content with their products and change their programs, striving to achieve what they consider a perfection. This is the stimulus of progress and large programs ought to be constructed in such a way that reprogramming could be achieved as cheaply as possible. Modularity, application of high-level languages and good documentation (including the motivation of decisions taken) are factors facilitating the changes in large programs.

Parametrization is a facility useful for the users and maintenance people rather than for the designers. It is in fact quite burdensome for the latter, but since the large programs should be user-oriented, the designers are forced to accept it.

7. Enough has been said about possibilities of complete, detailed and timely documentation for program designers. The volume of documentation meant for the maintenance team could be reduced substantially and is minimal for the users (description of input/output system languages, user's and operator's manuals).

8. The necessity of program correctness proving and automatic proof-checking are recently among the most talked about topics. Some research in these areas is needed since one can expect results applicable to particular cases. I do not hope, however, for a general success.

Automatic proof-checking applied to proofs or program correctness presupposes:

- a formal, precise and complete definition of program objectives
- that the program should reach these objectives under any conditions
- that the program is fully debugged
- that a proof of the fact that the program achieves its objectives has been found automatically or manually.

I do not believe that any of these objectives could be really satisfied.

Computer solutions are usually approximate. There is no satisfactory formalization of the notion of an approximate value of a quantity, let alone a formalization of the notion of an approximate solution of a problem of any complexity, if we require that this notion should be both constructive and meaningful.

All practically interesting classes of problems are usually algorithmically undecidable (van der Poel). Therefore, no program can solve all problems for which it has been constructed. Hence it is very difficult, to debugg a program fully, for the same reason there is no need in complete debugging.

It is hardly easier to prove the correctness of programs than to establish proofs of theorems. No fewer programs (not too large) are being written than hypotheses (would-be theorems) are being formulated. How many of the latter are being proven ?

I consider a program as a technological object. No one proves mathematically that an airplane or a bridge are designed correctly. Their correctness is supported by more or less plausible calculations and when the object is constructed it is thoroughly tested before being handed over to the users. Nevertheless, breakdowns do occur. The same with programs. Human life seldom depends on the correctness of a program, therefore the release of an incompletely debugged program is not immoral (even though it endangers the professional reputation of its designers). The degree to which a program should be debugged, should be, as a rule, determined from economic considerations: what would be more expensive - incorrect solution of some problems or impossibility to solve any problems at all (while a program for their solution is being debugged) ?

Any formal criterion to determine the bug-free state of programs, as well as for its optimality, would be of doubtful value, best of all some common sense should be applied. Admittedly, however, the common sense of some users selecting a program for solving their problems, is open to great doubts.

I have mentioned (in p. 4) programming languages for writing large programs. In principle, high level languages facilitate the description of an algorithm, low level ones - enhance the efficiency of its execution. When an algorithm is debugged it is not very difficult to transcribe it in another language, even for a large program. Therefore, in many cases it is advisable to make the preliminary programming in a language of as high as possible level, and the final version in a low-level language.

No language has ever been designed for description of a single algorithm. I am not convinced that classes of large problems, for which it would be worthwhile to construct special programming languages, have, as yet, been formed. One of these classes: compiler writing for special purpose languages seems, however, to be coming of age. For problems of this class we have proposed an extendable algorithmic language. Professional honesty (cf. p. 10) forces us to refrain from evaluating the prospects of this project.

10. Any experience gained in developing large project helps in the production of large programs. One of the elements of this experience consists, however, in that nothing from an earlier experience can be mechanically applied in new environment, everything should be critically analysed.

Almost all above given considerations are more philosophical than scientific. This reflects, in my opinion, the reality - programming is not yet a science, it has no axioms, no theorems, only more or less viable recipes. Nevertheless, programming and computer solving of small and large problems has already won its place in society even though yet not so important as, say, cooking. Programming starts bringing in some theoretical results, in particular it has enriched our understanding of languages, symbols, meanings and values.

RANDELL: There is much argument as to whether organic chemistry is a branch of cooking, or cooking is a branch of organic chemistry.

Early on in the set of LAVROV's answers there was a comment "failure to adhere to the laws of programming will have a bad effect on a small program, and a disastrous effect on a large program". What are the laws of programming ?

LAVROV: I have already said that, in my opinion, such laws do exist and I have tried to formulate some of them. These, of course, are not so much scientific rules as prescriptions for organization, data structures, control sequencing and so on.

The second part of Session V is devoted to discussion of written comments made by H. A. Kinslow and M. E. Conway.

TURSKI: I would like to quote a letter which was sent in by Mr. Kinslow, who could not come and take part in the conference.

"I have been concerned for some time about the attitude of large segments of the programming profession toward the 'large program' problem. In a sense the profession - at least the articulate members of it - seem polarized between those who believe that a small dedicated group of artists can build the best system and those who believe in the so-called Chinese Army approach.

For many years I believed in the former approach but I have come around to think that it is wrong. It was viable in its day, but the sophistication of present and proposed software is beyond the point where dedicated artistry alone will suffice.

Unfortunately this fact does not seem to be common knowledge yet. I sense that many of the influential people in software and most if not all of the academic people, have no real interest in the organizational or logistical problems of large programs. Their attention is drawn to intellectual challenge, but they have neither training nor aptitude nor interest in the administrative challenge. The expression ´Chinese Army´, usually used with reference to the 1000 people who worked on OS 360, seems to me to sum up the prevailing attitude. A more useful approach would be to examine the organization of such a project and try to form objective conclusions about the right or wrong of it.

I would suggest that in a Workshop of the type you propose excellent use could be made of the ´case-study´ method. I have attended such conferences before, and there tends to be much discussion of proposed tools and techniques (i.e. higher level language, software factories) but relatively little examination of specific historical cases to see what went right (or, more usually, wrong). There is also a great temptation to dwell on laboratory successes and extrapolate these to industrial circumstances without an attempt to define the difference in laboratory vs. industrial environments."

HOARE: He is talking nonsense. In the organization of a large Chinese Army, there are many organizational problems. I have some familiarity with them. There are problems like: how much should we pay this man ? this man is going to leave, shall we put his salary up ? shall we recruit this man ? this man does not like that man, can we make sure they are in different teams ? These are administrative problems, anybody in administrative position has had to deal with them. But they are not, after all, the subject of computer science, they are not to concern directly any scientist working in this field. What the scientists can assist, is in providing techniques for the sensible, technological break-down of the large tasks into small tasks. And this is not an administrative but a technical matter, on which I know that many scientists, in universities even, are working. Therefore I think that Mr. Kinslow is beside the point, when he accuses us of not working on the problems of pay-roll and personalities, and instead concentrating on those areas where a technically sound appreciation of the possibilities of modularity are just as much important to the success of the project - I do not say more important - than the administrative competence of those who lead it.

TURSKI: I shall try to defend Mr. Kinslow's point of view. I think that he did not exactly mean only those organizational problems, but he had in mind also the problems discussed in our session on design methodology, in which it was quite obvious that the pessimistic conclusion drawn by the chairman of the session, and which was not really contested by the participants of the meeting, reflects a lack of knowledge of the managerial side of the project, of the technical management of the project. We do not know how to manage technically the large scale software development. It is one of the problems, on which I think our workshop did concentrate a bit, and therefore this may be the real refutation of Kinslow's point. We were not sitting in an ivory tower, we did recognize the problem though we did not see any solution to it. Perhaps it is much better attitude than if we just said: "well, it is easy".

LAVROV: It is easy to understand how the approach, called by Kinslow "the Chinese Army", can arise. In the development of any large project, there is plenty of small, particular problems to be practically solved. If an organization is such that it can afford to recruit such a large number of people, and put each of them to a separate problem one gets the Chinese Army approach. At first sight it seems to be a good idea: it ensures that each question is carefully considered and suitable solutions are obtained. But these difficulties arise also quite naturally: such a large number of people cannot have enough experience, nor necessary knowledge to comprehend the problem in its entirety. Therefore, the decisions made by such large number of people must be evaluated by their own project leaders, so you get a complete hierarchy of bosses. But a boss cannot go through all the details of the solutions worked out by their subordinates, consequently there is a substantial loss of information in the hierarchy. Therefore, what happens is that as the result of very long series of decisions, each of which in itself is quite acceptable, arises a system which is unacceptable, or only marginally acceptable. A perfect example is OS 360. The second difficulty is that the Chinese Army has a colossal inertia: if it starts moving in a certain direction it is very difficult to stop it. That is why I emphasized in my contribution the necessity to answer the question: is the project necessary at all, and to discuss it not only among its designers, but also by its future users. Here we have the unfortunate experience that the discussion took place, decisions were taken yet the project, because of the inertia, progresses straight on as before.

RANDELL: In Datamation, 1968, there was a paper by M.E. Conway (cf. Appendix 2). The basic thesis of this paper is that in the design of a large system, for example, a large programming system, by a large group of people, of necessity the structure of final system will bear a very close correspondence to the organizational structure of the group that produced it. Conway gives a fairly detailed reasoning as to why this may happen, he gives also some very amusing examples. Similarly, commenting on OS 360, George Mealy, who was on the original design team, said that the first few months they considered some basic problem areas and set up one group of people to carry on further work in each problem area. When later it was found that there were one or two other problem areas of equal importance, it was no longer possible to fit such groups into the structure. These problem areas really did not get tackled at all. Therefore the structure of the system, and really the extent of the problem area that was tackled, reflect the way in which the organization of group was structured. Conway describes this as almost a rule, or a principle, that the structure of a system will reflect the structure of the large group of people that are working on it. If there is such a rule, then one thing you must do is to try to take advantage of it. And you must consciously structure your team in a way you hope that your final product should be structured.

APPENDIX 1 : PRELIMINARY SCOPE OUTLINE

Following is the text distributed by W. M. Turski prior to the commencement of the Workshop.

1. What is a "large program" ? Which of the characteristics of a program make it "large" ? What are the essential attributes of a "large" program ? Why producing "large programs" is (or should be) different from other forms of programming activity ?
2. How does one set the design criteria and specifications of large programs ? Which should be the proper attitude in designing a large program:

   i. user-oriented ?
   ii. manager-oriented ?
   iii. hardware-oriented ?

   What are the optimization criteria in designing a large program ?
3. How should one proceed in directing the actual production of a large program ? How to estimate the "man-months" cost of production ? What is the required level of "tooling" necessary to start such production ? Is it possible to organize the production of large programs "collaterally" or should one rather decide on a "serial" schedule ? Is it conceivable "to PERT" such undertaking ?
4. What about automatic production of large programs ? Is the "total automation" of this process anything but a pipedream ? How to select parts of a large program to be produced automatically ? Is it feasible to make such decisions not exclusively by intuition ? What are the circumstances under which automatic design could be not only more challenging but also more economical approach ?

5. What is the best (in view of the up-to-day experience) solution to the essential controversy between modularity of the product (a large program) and efficiency of its use ? Is the (usually high) run-time overhead incurred by making the program modular justified by the gained flexibility of use and relative simplicity of programming ?

6. Is it really worthwhile to design large programs in such a way that they may be "easily" changed by the user ? What is the nature of changes the user may be expected to make ? Is the modularity of large programs the only practicable way to let the user incorporate such changes as he may reasonably hope to be able to introduce ? What is the difference between modularity and parametrization from the user's and designer's point of view ?

7. What is the proper way to document large programs ? What levels of documentation could be distinguished ? What degree of compatibility between these levels could be achieved and how ?

8. How does one check the "correctness" of a large program ? Is debugging of large programs possible ? How can one be sure that a large program is bug-free ? What is the extent of experiments to be carried out before a large program may be considered empirically O.K. ? Supposing, that a large program is found to be empirically bug-free, how does one check it against the design goals ? Is it possible, with our present-day knowledge and expertise, to decide which of several large programs, produced to apparently same specs, supposedly bug-free, is the best one ?

9. Assuming that large programs are not necessarily written in low-level languages, what are the language features needed to be able to compose large programs ? Are any of the existing higer-level languages suitable for writing large programs ? If not - why ? Is it a "fault" of languages or complexity of the problem ?

10. To what extent the production of large programs is aided by the past experience and to what extent it is hampered by it ? Does the accumulated tradition of large-program making provide a good basis for theoretical consolidation or should we rather

start from scratch, building on a (hopefully sound) set of axioms ? What can be done (and how) to clarify and simplify considerably the gobbledegook currently en vogue in the trade of large-programs makers ? How the standards of quality and professional honesty can be established in the area of large programs ? Are they different in any way from those applicable to programming at large ?

APPENDIX 2 : HOW DO COMMITTEES INVENT ?

Following excerpts from Dr. Conway's paper are reprinted with kind permission of the "Datamation" magazine; the original paper was first published in April 1968.

That kind of intellectual activity which creates a useful whole from its diverse parts may be called the design of a system. Whether the particular activity is the creation of specifications for a major weapon system, the formation of a recommendation to meet a social challenge, or the programming of a computer, the general activity is largely the same.

Typically, the objective of a design organization is the creation and assembly of a document containing a coherently structured body of information. We may name this information the system design.

The design organization may or may not be involved in the construction of the system it designs.

It seems reasonable to suppose that the knowledge that one will have to carry out one's own recommendations or that this task will fall to others, probably affects some design choices which the individual designer is called upon to make. Most design activity requires continually making choices. Many of these choices may by more than design decisions; they may also be personal decisions the designer makes about his own future. As we shall see later, the incentives which exist in a conventional management environment can motivate choices which subvert the intent of the sponsor.[1]

---

[1] A related, but much more comprehensive discussion of the behavior of system-designing organizations is found in John Kenneth Galbraith's, The New Industrial State (Boston, Houghton Mifflin, 1967). See especially Chapter VI, "The Technostructure".

Stages of Design

The initial stages of a design effort are concerned more with structuring of the design activity than with the system itself.[2] The full-blown design activity cannot proceed until certain preliminary milestones are passed. These include:

1. Understanding of the boundaries, both on the design activity and on the system to be designed, placed by the sponsor and by the world's realities.
2. Achievement of a preliminary notion of the system's organization so that design task groups can be meaningfully assigned.

We shall see in detail later that the very act of organizing a design team means that certain design decisions have already been made, explicitly or otherwise. Given any design team organization, there is a class of design alternatives which cannot be effectively pursued by such an organization because the necessary communication paths do not exist. Therefore, there is no such thing as a design group which is both organized and unbiased.

Once the organization of the design team is chosen, it is possible to delegate activities to the subgroups of the organization. Every time a delegation is made and somebody's scope of inquiry is narrowed, the class of design alternatives which can be effectively pursued is also narrowed.

Once scopes of activity are defined, a coordination problem is created. Coordination among task groups, although it appears to lower the productivity of the individual in the small group, provides the only possibility that the separate task groups will be able to consolidate their efforts into a unified system design.

Thus the life cycle of a system design effort proceeds through the following general stages:

1. Drawing of boundaries according to the ground rules.
2. Choice of a preliminary system concept.
3. Organization of the design activity and delegation of tasks according to that concept.

---

[2] For a discussion of the problems which may arise when the design activity takes the form of a project in a functional environment, see C.J. Middleton, "How to Set Up a Project Organization", Harvard Business Review, March-April, 1967, p. 73.

4. Coordination among delegated tasks.
5. Consolidation of subdesigns into a single design.

It is possible that a given design activity will not proceed straight through this list. It might conceivably reorganize upon discovery of a new, and obviously superior, design concept; but such an appearance of uncertainty is unflattering, and the very act of voluntarily abandoning a creation is painful and expensive. Of course, from the vantage point of the historian, the process is continually repeating. This point of view has produced the observation that there's never enough time to do something right, but there's always enough time to do it over.

The Designed System

Any system of consequence is structured from smaller subsystems which are interconnected. A description of a system, if is to describe what goes on inside that system, must describe the system's connections to the outside world, and it must delineate each of the subsystems and how they are interconnected. Dropping down one level, we can say the same for each of the subsystems, viewing it as a system. This reduction in scope can continue until we are down to a system which is simple enough to be understood without further subdivision.

Fig. 1 illustrates this view of a system as a linear graph - a Tinker-Toy structure with branches (the lines) and nodes (the circles). Each node is a subsystem which communicates with other subsystems along the branches. In turn, each subsystem may contain a structure which may be similarly portrayed.The term interface, which is becoming popular among systems people,refers to the intersubsystem communication path or branch represented by a line in Fig. 1.Alternatively, the interface is the plug or flange by which the path coming out of one node couples to the path coming out of another node.

Relating the Two

The linear-graph notation is useful because it provides an abstraction which has the same form for two entities we are considering: the design organization and the system it designs. This can be illustrated in Fig. 1 by replacing the following words:

1. Replace "system" by "committee".
2. Replace "subsystem" by "subcommittee".
3. Replace "interface" by "coordinator".

Just as with systems, we find that design groups can be viewed at several levels of complication.

A basic relationship. We are now in a position to address the fundamental question of this article: Is there any predictable relationship between the graph structure of a design organization and the graph structure of the system it designs ? The answer is: Yes, the relationship is so simple that in some cases it is an identity. Consider the following "proof":

Let us choose arbitrarily some system and the organization which designed it, and let us then choose equally arbitrarily some level of complication of the designed system for which we can draw a graph. (Our motivation for this arbitrariness is that if we succeed in demonstrating anything interesting, it will hold true for any design organization and level of complication.) Fig. 2 shows, for illustration purposes only, a structure to which the following statements may be related.

For any node x in the system we can identify a design group of the design organization which designed x; call this X. Therefore, by generalization of this process, for every node of the system we have a rule for finding a corresponding node of the design organization. Notice that this rule is not necessarily one-to-one; that is, the two subsystems might have beed designed by a single design group.

Interestingly, we can make a similar statement about branches. Take any two nodes x and y of the system. Either they are joined by a branch or they are not. (That is, either they communicate with each other in some way meaningful to the operation of the system or they do not.) If there is a branch, then the two (not necessarily distinct) design groups X and Y which designed the two nodes must have negotiated and agreed upon an interface specification to permit communication between the two corresponding nodes of the design organization. If, on the other hand, there is no branch between x and y, then the subsystems do not communicate with each other, there was nothing for the two corresponding design groups to negotiate, and therefore there is no branch between X and Y.[3]

What have we just shown ? Roughly speaking, we have demonstrated that there is a very close relationship between the structure of a system and the structure of the organization which designed it. In the not unusual case where each subsystem had its own separate design group, we find that the structures (i.e., the linear graphs) of the design group and the system are identical. In the case where some group designed more than one subsystem we find that the structure of the design organization is a collapsed version of the structure of the system, with the subsystems having the same design group collapsing into one node representing that group.

This kind of a structure-preserving relationship between two sets of things is called a homomorphism. Speaking as a mathematician might, we would say that there is a homomorphism form the linear graph of a system to the linear graph of its design organization.

Systems Image Their Design Groups

It is an article of faith among experienced system designers that given any system design, someone some day will find a better one to do the same job. In other words, it is misleading and incorrect to speak of the design for a specific job, unless this is understood in the context of space, time, knowledge, and technology. The humility which this belief should impose on system designers is the only appropriate posture for those who read history or consult their memories.

The design progress of computer translators of programming languages such as FORTRAN and COBOL is a case in point. In the middle fifties, when the prototypes of these languages appeared, their compilers were even more cumbersome objects than the giant (for then) computers which were required for their execution. Today, these translators are only historical curiosities, bearing no resemblance in design to today's compilers. (We should take particular note of the fact that the quantum jumps in com-

[3] This claim may be viewed several ways. It may be trivial, hinging on the definition of meaningful negotiation. Or, it may be the result of the observation that one design group almost never will compromise its own design to meet the needs of another group unless absolutely imperative.

piler design progress were associated with the appearance of new groups of people on territory previously trampled chiefly by computer manufacturers - first it was the tight little university research team, followed by the independent software house.)

If, then, it is reasonable to assume that for any system requirement there is a family of system designs which will meet that requirement, we must also inquire whether the choice of design organization influences the process of selection of a system design from that family. If we believe our homomorphism, then we must agree that it does. To the extent that an organization is not completely flexible in its communication structure, that organization will stamp out an image of itself in every design it produces. The larger an organization is, the less flexibility it has and the more pronounced is the phenomenon.

Examples. A contract research organization had eight people who were to produce a COBOL and an ALGOL compiler. After some initial estimates of difficulty and time, five people were assigned to the COBOL job and three to the ALGOL job. The resulting COBOL compiler ran in five phases, the ALGOL compiler ran in three.

Two military services were directed by their Commander-in-Chief to develop a common weapon system to meet their respective needs. After great effort they produced a copy of their organization chart. (See Fig. 3a.)

Consider the operating computer system in use solving a problem. At a high level of examination, it consists of three parts: the hardware, the system software, and the application program. (See Fig. 3b.) Corresponding to these subsystems are their respective designers: the computer manufacturer's engineers, his system programmers, and the user's application programmers. (Those rare instances where the system hardware and software tend to cooperate rather than merely tolerate each other are associated with manufacturers whose programmers and engineers bear a similar relationship.)

System Management

The structures of large systems tend to disintegrate during development, qualitatively more so than with small systems. This observation is strikingly evident when applied to the large military information systems of the last dozen years; these are some of the most complex objects devised by the mind of man. An activity called "system management" has sprung up partially in response to this tendency of systems to disintegrate. Let us examine the utility to system management of the concepts we have developed here.

Why do large systems disintegrate ? The process seems to occur in three steps, the first two of which are controllable and the third of which is a direct result of our homomorphism.

First, the realization by the initial designers that the system will be large, together with certain pressure in their organization, make irresistible the temptation to assign too many people to a design effort.

Second, application of the conventional wisdom of management to a large design organization causes its communication structure to disintegrate.

Third, the homomorphism insures that the structure of the system will reflect the disintegration which has occurred in the design organization.

Let us first examine the tendency to overpopulate a design effort. It is a natural temptation of the initial designer - the one whose preliminary design concepts influence the organization of the design effort - to delegate tasks when the apparent complexity of the system approaches his limits of comprehension. This is the turning point in the course of the design. Either he struggles to reduce the system to comprehensibility and wins, or else he loses control of it. The outcome is almost predictable if there is schedule pressure and a budget to be managed.

A manager knows that he will be vulnerable to the charge of mismanagement if he misses his schedule without having applied his resources. This knowledge creates a strong pressure on the initial designer who might prefer to wrestle with the design rather than fragment it by delegation, but he is made to feel that the cost of risk is too high to take the chance. Therefore, he is forced to delegate in order to bring more resources to bear.

The following case illustrates another but related way in which the environment of the manager can be in conflict with the integrity of the system being designed.

A manager must subcontract a crucial and difficult design task. He has a choice of two contractors, a small new organization

which proposes an intuitively appealing approach for much less money than is budgeted, and an established but conventional outfit which is asking a more "realistic" fee. He knows that if the bright young organization fails to produce adequate results, he will be accused of mismanagement, whereas if the established outfit fails, it will be evidence that the problem is indeed a difficult one.

What is the difficulty here ? A large part of it relates to the kind of reasoning about measurement of resources which arises from conventional accounting theory. According to this theory, the unit of resource is the dollar, and all resources must be measured using units of measurement which are convertible to the dollar. If the resource is human effort, the unit of measurement is the number of hours worked by each man times his hourly cost, summed up for the whole working force.

One fallacy behind this calculation is the property of linearity which says that two men working for a year or one hundred men working for a week (at the same hourly cost per man) are resources of equal value. Assuming that two men and one hundred men cannot work in the same organizational structure (this is intuitively evident and will be discussed below) our homomorphism says that they will not design similar systems; therefore the value of their efforts may not even be comparable. From experience we know that the two men, if they are well chosen and survive the experience, will give us a better system. Assumptions which may be adequate for peeling potatoes and erecting brick walls fail for designing systems.

Parkinson's Law[4] plays an important role in the overassignment of design effort. As long as the manager's prestige and power are tied to the size of his budget, he will be motivated to expand his organization. This is an inappropriate motive in the management of a system design activity. Once the organization exists, of course, it will be used. Probably the greatest single common factor behind many poorly designed systems now in existence has been the availability of a design organization in need of work.

---

[4] **C. Northcote Parkinson, Parkinson's Law and Other Studies in Administration (Boston, Houghton Mifflin, 1957).**

The second step in the disintegration of a system design - the fragmentation of the design organization's communication structure - begins as soon as delegation has started. Elementary probability theory tells us that the number of possible communication paths in an organization is approximately half the square of the number of people in the organization. Even in a moderately small organization it becomes necessary to restrict communication in order that people can get some "work" done. Research which leads to techniques permitting more efficient communication among designers will play an extremely important role in the technology of system management.

Conclusion

The basic thesis of this article is that organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations. We have seen that this fact has important implications for the management of system design. Primarily, we have found a criterion for the structuring of design organizations: a design effort should be organized according to the need for communication.

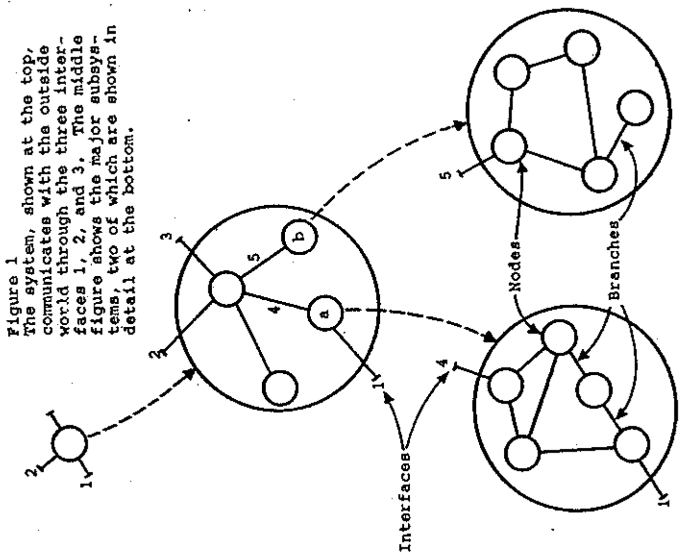
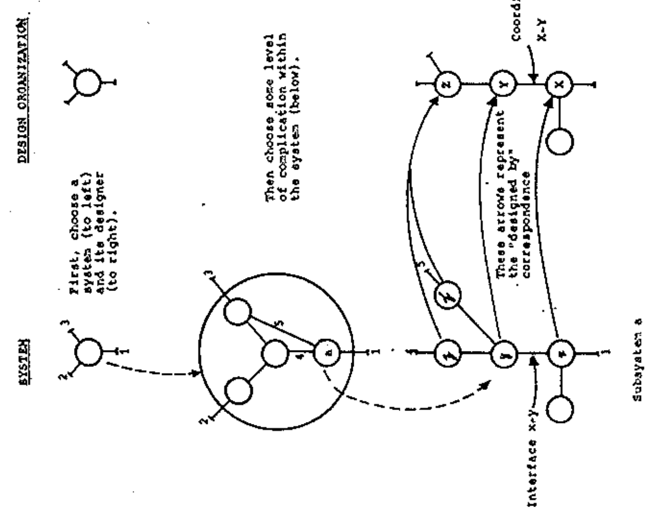


Figure 1
The system, shown at the top, communicates with the outside world through the three interfaces 1, 2, and 3. The middle figure shows the major subsystems, two of which are shown in detail at the bottom.

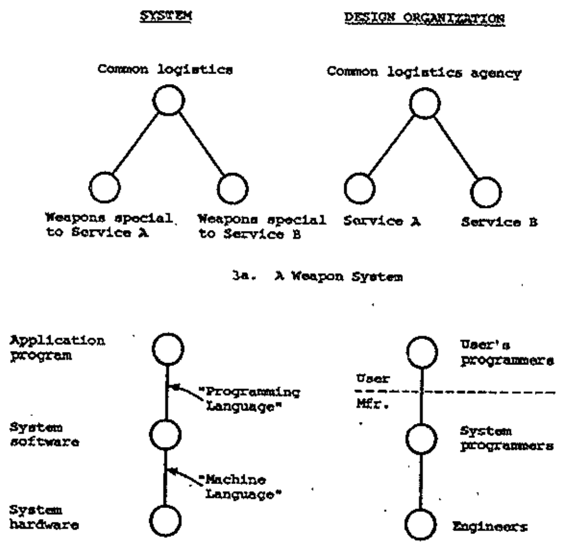
Fig. 1



SYSTEM
DESIGN ORGANIZATION
First, choose a system (to left) and its designer (to right).
Then choose some level of complication within the system (below).
These arrows represent the "designed by" correspondence
Coordinator
X-Y
Interface x-y
Subsystem a

Figure 2 Here is an illustration of the strong relationship between the structure (graph) of a system (left) and the structure of the organization which designed it (right)

Fig. 2



3a. A Weapon System

3b. A Computer System

Figure 3 Two examples of identity of structure between a system and its design organization.

Figs. 3a and 3b

Printed in Poland

PAŃSTWOWE
WYDAWNICTWO NAUKOWE

*

Wydanie I. Nakład 260+90 egz.
Ark. wyd. 8,00. Ark. druk. 8,75.
Papier offset. kl. III. 80 g. 70×100.
Oddano do reproduk. 13. IV. 1971 r.
Podpisano do druku 5. V. 1971 r.
Druk ukończono w maju 1971 r.
Zam. nr 210. Cena zł 24,—

*

ZAKŁAD GRAFICZNY
PWN w ŁODZI

Orders for this book should be addressed to:
CHZ Ars Polona-RUCH, Warszawa, Krakowskie Przedmieście 7.