Scanned from P. Naur and B. Randell, "Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968", Scientific Affairs Division, NATO, Brussels, 1969, 138-155. Layout has not been preserved. Isolated misprints in the original have been corrected, and a hodge-podge of English and American spellings has been resolved in favor of American. Scanning errors may remain.

- MDM, 15 October 1998

The entire report is at http://homepages.cs.ncl.ac.uk/brian.randell/NATO.

A photo taken at the lecture can be seen at http://www.cs.ncl.ac.uk/old/people/brian.randell/home.formal/NATO/N1968/index.html

8.2. MASS PRODUCED SOPTWARE COMPONENTS, BY M.D. McILROY

ABSTRACT

Software components (routines), to be widely applicable to different machines and users, should be available in families arranged according to precision, robustness, generality and time-space performance. Existing sources of components - manufacturers, software houses, users' groups and algorithm collections - lack the breadth of interest or coherence of purpose to assemble more than one or two members of such families, yet software production in the large would be enormously helped by the availability of spectra of high quality routines, quite as mechanical design is abetted by the existence of families of structural shapes, screws or resistors. The talk will examine the kinds of variability necessary in software components, ways of producing useful inventories, types of components that are ripe for such standardization, and methods of instituting pilot production.

The Software Industry is Not Industrialized

We undoubtedly produce software by backward techniques. We undoubtedly get the short end of the stick in confrontations with hardware people because they are the industrialists and we are the crofters. Software production today appears in the scale of industrialization somewhere below the more backward construction industries. I think its proper place is considerably higher, and would like to investigate the prospects for mass-production techniques in software.

In the phrase `mass production techniques,' my emphasis is on `techniques' and not on mass production plain. Of course mass production, in the sense of limitless replication of a prototype, is trivial for software. But certain ideas from industrial technique I claim are relevant. The idea of subassemblies carries over directly and is well exploited. The idea of interchangeable parts corresponds roughly to our term `modularity,' and is fitfully respected. The idea of machine tools has an analogue in assembly programs and compilers. Yet this fragile analogy is belied when we seek for analogues of other tangible symbols of mass production. There do not exist manufacturers of standard parts, much less catalogues of standard parts. One may not order parts to individual specifications of size, ruggedness, speed, capacity, precision or character

set.

The pinnacle of software is systems - systems to the exclusion of almost all other considerations. Components, dignified as a hardware field, is unknown as a legitimate branch of software. When we undertake to write a compiler, we begin by saying `What table mechanism shall we build.' Not, `What mechanism shall we use?' but `What mechanism shall we build?' I claim we have done enough of this to start taking such things off the shelf.

Software Components

My thesis is that the software industry is weakly founded, and that one aspect of this weakness is the absence of a software components subindustry. We have enough experience to perceive the outline of such a subindustry. I intend to elaborate this outline a little, but I suspect that the very name `software components' has probably already conjured up for you an idea of how the industry could operate. I shall also argue that a components industry could be immensely useful, and suggest why it hasn't materialized. Finally I shall raise the question of starting up a `pilot plant' for software components.

The most important characteristic of a software components industry is that it will offer families of routines for any given job. No user of a particular member of a family should pay a penalty, in unwanted generality, for the fact that he is employing a standard model routine. In other words, the purchaser of a component from a family will choose one tailored to his exact needs. He will consult a catalogue offering routines in varying degrees of precision, robustness, time-space performance, and generality. He will be confident that each routine in the family is of high quality - reliable and efficient. He will expect the routine to be intelligible, doubtless expressed in a higher level language appropriate to the purpose of the component, though not necessarily instantly compilable in any processor he has for his machine. He will expect families of routines to be constructed on rational principles so that families fit together as building blocks. In short, he should be able safely to regard components as black boxes.

Thus the builder of an assembler will be able to say `I will use a String Associates A4 symbol table, in size 500x8,' and therewith consider it done. As a bonus he may later experiment with alternatives to this choice, without incurring extreme costs.

A Familiar Example

Consider the lowly sine routine. How many should a standard catalogue offer? Off hand one thinks of several dimensions along which we wish to have variability:

Precision, for which perhaps ten different approximating functions might suffice
Floating-vs-fixed computation
Argument ranges 0-pi/2, O-2pi, also -pi/2 to pi/2,
-pi to pi, -big to +big
Robustness - ranging from no argument validation through signaling

of complete loss of significance, to signaling of specified range violations.

We have here 10 precisions, 2 scalings, 5 ranges and 3 robustnesses The last range option and the last robustness option are actually arbitrary parameters specifiable by the user. This gives us a basic inventory of 300 sine routines. In addition one might expect a complete catalogue to include a measurement-standard sine routine, which would deliver (at a price) a result of any accuracy specified at run time. Another dimension of variability, which is perhaps difficult to implement, as it caters for very detailed needs is

Time-space tradeoff by table lookup, adjustable in several `subdimensions':
(a) Table size
(b) Quantization of inputs (e.g., the inputs are known to be integral numbers of degrees)
Another possibility is
(c) Taking advantage of known properties of expected input sequences, for example profiting from the occurrence of successive calls for sine and cosine of the same argument.

A company setting out to write 300 sine routines one at a time and hoping to recoup on volume sales would certainly go broke. I can't imagine some of their catalogue items ever being ordered. Fortunately the cost of offering such an `inventory' need not be nearly 300 times the cost of keeping one routine. Automated techniques exist for generating approximations of different degrees of precision. Various editing and binding techniques are possible for inserting or deleting code pertinent to each degree of robustness. Perhaps only the floating-vs-fixed dichotomy would actually necessitate fundamentally different routines. Thus it seems that the basic inventory would not be hard to create.

The example of the sine routine re-emphasizes an interesting fact about this business. It is safe to assert that almost all sines are computed in floating point these days, yet that would not justify discarding the fixed point option, for that could well throw away a large part of the business in distinct tailor-made routines for myriads of small process-control and other real-time applications on all sorts of different hardware. `Mass production' of software means multiplicity of what manufacturing industry would call `models,' or `sizes' rather than multiplicity of replicates of each.

Parameterized Families of Components

One phrase contains much of the secret of making families of software components: `binding time.' This is an `in' phrase this year, but it is more popular in theory than in the field. Just about the only applications of multiple binding times I can think of are sort generators and the so-called `Sysgen' types of application: filling in parameters at the time routines are compiled to control table sizes, and to some extent to control choice among several bodies of code. The best known of these, IBM's OS/360 Sysgen is indeed elaborate - software houses have set themselves up as experts on this job. Sysgen differs, though, in a couple of ways from what I have in mind as the way a software components industry

might operate.

First, Sysgen creates systems not by construction, but rather by excision, from an intentionally fat model. The types of adjustment in Sysgen are fairly limited. For example it can allocate differing amounts of space to a compiler, but it can't adjust the width of list link fields in proportion to the size of the list space. A components industry on the other hand, not producing components for application to one specific system, would have to be flexible in more dimensions, and would have to provide routines whose niches in a system were less clearly delineated.

Second, Sysgen is not intended to reduce object code or running time. Typically Sysgen provides for the presetting of defaults, such as whether object code listings are or are not standard output from a compiler. The entire run-time apparatus for interrogating and executing options is still there, even though a customer might guarantee he'd never use it were it indeed profitable to refrain. Going back to the sine routine, this is somewhat like building a low precision routine by computing in high precision and then carefully throwing away the less significant bits.

Having shown that Sysgen isn't the exact pattern for a components industry, I hasten to add that in spirit it is almost the only way a successful components industry could operate. To purvey a rational spectrum of high quality components a fabricator would have to systematize his production. One could not stock 300 sine routines unless they were all in some sense instances of just a few models, highly parameterized, in which all but a few parameters were intended to be permanently bound before run time. One might call these early-bound parameters `sale time' parameters.

Many of the parameters of a basic software component will be qualitatively different from the parameters of routines we know today. There will be at least

Choice of Precision. Taken in a generalized sense precision includes things like width of characters, and size of address or pointer fields.

Choice of Robustness. The exact tradeoff between reliability and compactness in space and time can strongly affect the performance of a system. This aspect of parameterization and the next will probably rank first in importance to customers.

Choice of Generality. The degree to which parameters are left adjustable at run time.

Choice of Time-space behavior.

Choice of Algorithm. In numerical routines, as exemplified by those in the CACM, this choice is quite well catered for already. For nonnumerical routines, however, this choice must usually be decided on the basis of folklore. As some nonnumerical algorithms are often spectacularly unsuitable for particular hardware, a wide choice is perhaps even more imperative for them.

Choice of Interfaces. Routines that use several inputs and yield several outputs should come in a variety of interface styles. For example, these different styles of communicating error outputs should be available:

a. Alternate returns
b. Error code return
c. Call an error handler
d. Signal (in the sense of PL/I)

Another example of interface variability is that the dimensions of matrix parameters should be receivable in ways characteristic of several major programming languages.

Choice of Accessing method. Different storage accessing disciplines should be supported, so that a customer could choose that best fitting his requirements in speed and space, the addressing capabilities of his hardware, or his taste in programming style.

Choice of Data structures. Already touched upon under the topic of interfaces, this delicate matter requires careful planning so that algorithms be as insensitive to changes of data structure as possible. When radically different structures are useful for similar problems (e.g., incidence matrix and list representations for graphs), several algorithms may be required.

Application Areas

We have to begin thinking small. Despite advertisements to the effect that whole compilers are available on a `virtually off-the-shelf' basis, I don't think we are ready to make software subassemblies of that size on a production basis. More promising components to begin with are these:

Numerical approximation routines. These are very well understood, and the dimensions of variability for these routines are also quite clear. Certain other numerical processes aren't such good candidates; root finders and differential equation routines, for instance are still matters for research, not mass production. Still other `numerical' processes, such as matrix inversion routines, are simply logical patterns for sequencing that are almost devoid of variability. These might be sold by a components industry for completeness' sake, but they can be just as well taken from the CACM.

Input-output conversion. The basic pieces here are radix conversion routines, some trivial scanning routines, and format crackers. From a well-designed collection of families it should be possible to fabricate anything from a simple on-line octal package for a small laboratory computer to a Fortran IV conversion package. The variability here, especially in the matter of accuracy and robustness is substantial. Considerable planning will evidently be needed to get sufficient flexibility without having too many basically different routines.

Two and three dimensional geometry. Applications of this sort are going on a very wide class of machines, and today are usually kept proprietary. One can easily list a few dozen fundamental routines for

geometry. The sticky dimension of variability here is in data structures. Depending on which aspect of geometrical figures is considered fundamental - points, surfaces, topology, etc. - quite different routines will be required. A complete line ought to cater for different abstract structures, and also be insensitive to concrete structures.

Text processing. Nobody uses anybody else's general parsers or scanners today, partly because a routine general enough to fulfill any particular individual needs probably has so much generality as to be inefficient. The principle of variable binding times could be very fruitfully exploited here. Among the corpus of routines in this area would be dictionary builders and lookup routines, scanners, and output synthesizers, all capable of working on continuous streams, on unit records, and various linked list formats, and under access modes suitable to various hardware.

Storage management. Dynamic storage allocation is a popular topic for publication, about which not enough real knowledge yet exists. Before constructing a product line for this application, one ought to do considerable comparison of known schemes working in practical environments. Nevertheless storage management is so important, especially for text manipulation, that it should be an early candidate.

The Market

Coming from one of the larger sophisticated users of machines, I have ample opportunity to see the tragic waste of current software writing techniques. At Bell Telephone Laboratories we have about 100 general purpose machines from a dozen manufacturers. Even though many are dedicated to special applications, a tremendous amount of similar software must be written for each. All need input-output conversion, sometimes only single alphabetic characters and octal numbers, some full-blown Fortran style I/O. All need assemblers and could use macroprocessors, though not necessarily compiling on the same hardware. Many need basic numerical routines or sequence generators. Most want speed at all costs, a few want considerable robustness.

Needless to say much of this support programming is done suboptimally, and at a severe scientific penalty of diverting the machine's owners from their central investigations. To construct these systems of high-class componentry we would have to surround each of some 50 machines with a permanent coterie of software specialists. Were it possible quickly and confidently to avail ourselves of the best there is in support algorithms, a team of software consultants would be able to guide scientists towards rapid and improved solutions to the more mundane support problems of their personal systems.

In describing the way Bell laboratories might use software components, I have intended to described the market in microcosm. Bell laboratories is not typical of computer users. As a research and development establishment, it must perforce spend more of its time sharpening its tools, and less using them than does a production computing shop. But it is exactly such a systems-oriented market toward which a components industry would be directed.

The market would consist of specialists in system building, who would be able to use tried parts for all the more commonplace parts of their systems. The biggest customers of all would be the manufacturers. (Were they not it would be a sure sign that the offered products weren't good enough.) The ultimate consumer of systems based on components ought to see considerably improved reliability and performance, as it would become possible to expend proportionally more effort on critical parts of systems, and also to avoid the now prevalent failings of the more mundane parts of systems, which have been specified by experts, and have then been written by hacks.

Present Day Suppliers

You may ask, well don't we have exactly what I've been calling for already in several places? What about the CACM collected algorithms? What about users groups? What about software houses? And what about manufacturers' enormous software packages?

None of these sources caters exactly for the purpose I have in mind, nor do I think it likely that any of them will actually evolve to fill the need.

The CACM algorithms, in a limited field, perhaps come closer to being a generally available off-the-shelf product than do the commercial products, but they suffer some strong deficiencies. First they are an ingathering of personal contributions, often stylistically varied. They fit into no plan, for the editor can only publish that which the authors volunteer. Second, by being effectively bound to a single compilable language, they achieve refereeability but must perforce completely avoid algorithms for which Algol is unsuited or else use circumlocutions so abominable that the product can only be regarded as a toy. Third, as an adjunct of a learned society, the CACM algorithms section can not deal in large numbers of variants of the same algorithm. Variability can only be provided by expensive run time parameters

User's groups I think can be dismissed summarily, and I will spare you a harangue on their deficiencies.

Software houses generally do not have the resources to develop their own product lines; their work must be financed, and large financing can usually only be obtained for large products. So we see the software houses purveying systems, or very big programs, such as Fortran compilers, linear programming packages or flowcharters. I do not expect to see any software house advertising a family of Bessel functions or symbol tabling routines in the predictable future.

The manufacturers produce unbelievable amounts of software. Generally, as this is the stuff that gets used most heavily it is all pretty reliable, a good conservative grey, that doesn't include the best routine for anything, but that is better than the average programmer is likely to make. As we heard yesterday manufacturers tend to be rather pragmatic in their choice of methods. They strike largely reasonable balances between generality and specificity and seldom use absolutely inappropriate approaches in any individual software component. But the profit motive wherefrom springs these virtues also begets their prime

hangup - systems now. The system comes first; components are merely annoying incidentals. Out of these treadmills I don't expect to see high class components of general utility appear.

A Components Factory

Having shown that it is unlikely to be born among the traditional suppliers of software I turn now to the question of just how a components industry might get started.

There is some critical size to which the industry must attain before it becomes useful. Our purveyor of 300 sine routines would probably go broke waiting for customers if that's all he offered, just as an electronics firm selling circuit modules for only one purpose would have trouble in the market.

It will take some time to develop a useful inventory, and during that time money and talent will be needed. The first source of support that comes to mind is governmental, perhaps channeled through semi-independent research corporations. It seems that the fact that government is the biggest user and owner of machines should provide sufficient incentive for such an undertaking that has promise for making an across-the-board improvement in systems development.

Even before founding a pilot plant, one would be wise to have demonstrated techniques for creating a parameterized family of routines for a couple of familiar purposes, say a sine routine and a Fortran I/O module. These routines should be shown to be usable as replacements in a number of radically different environments. This demonstration could be undertaken by a governmental agency, a research contractor, or by a big user, but certainly without expectation of immediate payoff.

The industrial orientation of a pilot plant must be constantly borne in mind. I think that the whole project is an improbable one for university research. Research-caliber talent will be needed to do the job with satisfactory economy and reliability, but the guiding spirit of the undertaking must be production oriented. The ability to produce members of a family is not enough. Distribution, cataloguing, and rational planning of the mix of product families will in the long run be more important to the success of the venture than will be the purely technical achievement.

The personnel of a pilot plant should look like the personnel on many big software projects, with the masses of coders removed. Very good planning, and strongly product-minded supervision will be needed. There will be perhaps more research flavor included than might be on an ordinary software project, because the level of programming here will be more abstract: Much of the work will be in creating generators of routines rather than in making the routines themselves.

Testing will have to be done in several ways. Each member of a family will doubtless be tested against some very general model to assure that sale-time binding causes no degradation over runtime binding. Product test will involve transliterating the routines to fit in representative hardware. By monitoring the ease with which fairly junior people do product test, managers could estimate the clarity of the product, which is

important in predicting customer acceptance.

Distribution will be a ticklish problem. Quick delivery may well be a components purveyor's most valuable sales stimulant. One instantly thinks of distribution by communication link. Then even very small components might be profitably marketed. The catalogue will be equally important. A comprehensive and physically condensed document like the Sears-Roebuck catalogue is what I would like to have for my own were I purchasing components.

Once a corpus of product lines became established and profit potential demonstrated, I would expect software houses to take over the industry. Indeed, were outside support long needed, I would say the venture had failed (and try to forget I had ever proposed it).

Touching on Standards

I don't think a components industry can be standardized into existence. As is usual with standards, it would be rash to standardize before we have the models. Language standards, provided they are loose enough not to prevent useful modes of computation, will of course be helpful. Quite soon one would expect a components industry to converge on a few standard types of interface. Experience will doubtless reveal other standards to be helpful, for example popular word sizes and character sets, but again unless the standards encompass the bulk of software systems (as distinguished from users), the components industry will die for lack of market.

Summary

I would like to see components become a dignified branch of software engineering. I would like to see standard catalogues of routines, classified by precision, robustness, time-space performance, size limits, and binding time of parameters. I would like to apply routines in the catalogue to any one of a large class of often quite different machines, without too much pain. I do not insist that I be able to compile a particular routine directly, but I do insist that transliteration be essentially direct. I do not want the routine to be inherently inefficient due to being expressed in machine independent terms. I want to have confidence in the quality of the routines. I want the different types of routine in the catalogue that are similar in purpose to be engineered uniformly, so that two similar routines should be available with similar options and two options of the same routine should be interchangeable in situations indifferent to that option.

What I have just asked for is simply industrialism, with programming terms substituted for some of the more mechanically oriented terms appropriate to mass production. I think there are considerable areas of software ready, if not overdue, for this approach.

8.2.1. DISCUSSION

Ross: What Mcllroy has been talking about are things we have been playing with. For example, in the AED system we have the so-called feature-feature. This enables us to get round the problem of loaders. We

can always embed our system in whatever loader system is available. The problem of binding is very much interlocked there, so we are at the mercy of the environment. An example is a generalized alarm reporting system in which you can either report things on the fly, or put out all kinds of dynamic information. The same system gives 14 different versions of the alarm handling. Macro-expansion seems to me to be the starting place for some of the technical problems that have to be solved in order to put these very important ideas into practice.

McIlroy: It seems that you have automated some of types variability that I thought were more speculative.

Opler: The TOOL system produced six years ago for Honeywell was complementary to the one McIlroy described. It has facilities for putting things together, but it did not provide the components. The difficulty we had was that we produced rudimentary components to see how the system would work, but the people for whom we developed the system did not understand that they were to provide their own components, so they just complained that the system was not good. But I am very enthusiastic about what you suggest.

Perlis: The GP system of the first Univac was a system for developing personalized software as long as you stayed on that machine. The authors of this system asked me: how would one generalize this to other computers? They did not know how to do it at the time, and I suppose it has not been done. I have a question for Mcllroy. I did not hear you mention what to me is the most obvious of parameterizations, namely to build generalized business data file handling systems. I understand that Informatics has one out which everybody says is OK, but - . This seems to be a typical attitude to parameterized systems.

McIlroy: My reason for leaving that out is that this is an area that I don't know about.

Perlis: Probably it would be one of the easiest areas, and one with the most customers. Before d'Agapeyeff talks I have another comment. [Laughter]. Specialists in every part of software have a curious vision of the world: All parts of software but his are simple and easily parameterized; his is totally variable.

d'Agapeyeff: There is no package which has received more attention from manufacturers than file handling. Yet there is hardly a major system that I know of that is relying solely on the standard system produced by the manufacturer. It is extremely difficult to construct this software in a way that is efficient, reliable, and convenient for all systems and where the nature of the package does not impose itself upon the user. The reason is that you cannot atomize it. Where work has been successful it tends to be concerned with packages that have some structure. When you get down to small units it is not economic to make them applicable to a large set of users, using different machines with different languages, and to do all the binding work, such that it doesn't take twice as long to find out how to load it. The problems with Sysgen are not to be dispensed with, they are inherent. But why do we need to take atoms down from the shelf? What you want is a description which you can understand, because the time taken to code it into your own system is really very small. In that way you can

insert your own nuances. The first step in your direction should be better descriptions.

Endres: Two notes of caution: You discarded the algorithms in the Comm. ACM in part because they are written in high-level language, so I understand that you refer to routines written in a more machine oriented language. I think you oversimplify the problem of transliteration. Or do you assume a de facto machine standard? Second question: You refer to the problems of Sysgen, where you cut out pieces from a large collection. If instead you want to put together systems, I think the problems of Sysgen become a dimension larger. Who will bear this cost, and maintain the system?

McIlroy: The algorithms in the Comm. ACM effectively use one language, which is suitable for a particular class of applications. This may not be the right one for things like input/output packages. On the second question: I am convinced, with you, that at first it will be harder to build systems by accretion, rather than by excision. The people who build components will have to be skilled systems builders, not run of the mill users.

Kjeldaas: I strongly favor this idea. I think the examples mentioned are within the state of the art. However, later we will want macros needing parameters having more intricate relations, for instance if you want some functional relationship between the parameters. We will need some language for describing the parameters. Another point: documentation can also be included in this. When you have given the parameters to the program, you can give the same parameters to the documentation, and the documentation for the particular use can be produced automatically. Catering for different machines will raise big problems, needing research.

Kolence: May I stress one point: Mcllroy stated that the industrialization is concerned with the design, not the replication process. We are concerned with a mass design problem. In talking about the implementation of software components, the whole concept of how one designs software is ignored. Yet this is a key thing.

Naur: What I like about this is the stress on basic building principles, and on the fact that big systems are made from smaller components. This has a strong bearing on education. What we want in education, particularly at the more elementary level, is to start indoctrinating the knowledge of the components of our systems. A comparison with our hardware colleagues is relevant. Why are they so much more successful than we are? I believe that one strong reason is that there is a well established field of electronic engineering, that the young people start learning about Ohm's Law at the age of fourteen or thereabouts, and that resistors and the like are known components with characteristics which have been expounded at length at the early level of education. The component principles of our systems must be sorted out in such a form that they can be put into elementary education.

Gill: Two points: first on the catalogue question. I hope we can do better than the Sears-Roebuck catalogue. Surely what we want is a computerized conversational catalogue. Second point: what is it that you actually sell when you sell a piece of software, what exactly does a

software contract look like?

Barton: Mcllroy's talk was so well done that it took me about three minutes to realize what is wrong with this idea. Another compliment: If I were running Intergalactic Software, I would hire Mcllroy for a manager. Now the serious point: Over the last few years I have taught the ACM Course `Information Structures' and used the game not to let anyone code or write anything in any programming language at all. We have just thought about data representations. If in this way you get people over the habit of writing code right away, of thinking procedurally, then some very different views on information representations come to view. In McIlroy's talk about standard components having to do with data structures I have the feeling that this is not a problem to take out of the universities yet. Now a heretical view: I don't think we have softened up enough things in machines yet. I don't think we will get anywhere trying to quantify the space-time trade-off unless we discard fixed word sizes, fixed character sizes, fixed numerical representations, altogether in machines. Without these, the thing proposed by Mcllroy will prove to be just not quite practical.

Fraser: I wish to take issue with d'Agapeyeff. I think it will be possible to parameterize data representation and file management. From a particular file system experience I learned two lessons: first, there are a large number of parameters, to be selected in a non-mutually-exclusive manner. The selection of the parameters is so complicated that it is appropriate to put a compiler on the front end of the software distribution mechanism. Perhaps we are talking more about compilers than we realize. Concerning catalogues: in England a catalogue of building materials is a very ad hoc catalogue, you have left hand flanges to go with left hand gates, etc. I think the catalogue is likely to be ad hoc in that nature, rather than like an electronics catalogue where the components are more interchangeable.

The second issue is the question of writing this compiler. Our file management generator effectively would generate a large number of different file management systems, very considerably in excess of the 500 that Mcllroy mentioned. There was no question of testing all of these. We produced an ad hoc solution to this problem, but until more research is done on this problem I don't think McIlroy's suggestion is realistic.

Graham: I will speak of an adjunct to this idea. In Multics we used a subset of PL/I, although PL/I is quite inadequate, in that the primitive operations of the language are not really suited for system design. In Multics you do a lot of directory management, simple operations like adding and deleting entries, but in a complicated directory. With a higher-level language with these operations as primitives one could easily write a new system. By simulating the primitives one could test the performance of the system before actually building it. If one had Mcllroy's catalogue stored in the system, with the timings of a lot of routines, then the simulation backing up this higher-level language could in fact refer to the catalogue and use the actual timings for a new machine that this company offered and get realistic timings. Another point, I wish to rebutt McIlroy's suggestion that this is not for universities; I think it is. There are very difficult problems in this area, such as parameterizing more sophisticated routines, in particular those in the compiler area. These are fit for universities.

Bemer: I agree that the catalogue method is not a suitable one. We don't have the descriptors to go searching. There is nothing so poorly described as data formats, there are no standards, and no sign that they are being developed. Before we have these we won't have the components.

McIlroy: It is for that reason that I suggest the Sears-Roebuck type now. On-line searching may not be the right answer yet.