Arbeidsnotater

STATISTISK SENTRALBYRÅ

OSLO: Postboks 8131 Dep, Oslo 1
Tlf. (02) *41 38 20

KONGSVINGER: Postboks 510, Stasjonssida, 2201 Kongsvinger
Tlf. (066) *14 988

WORKING PAPERS FROM THE CENTRAL BUREAU OF STATISTICS OF NORWAY

IO 78/33

29. December 1978

DATSY

PAST AND FUTURE

by David Walker *

CONTENTS

* This report is based on minutes written by Frank Siljan for a seminar held at the Central Bureau of Statistics in Oslo, on September 26, 1978.

1. Introduction, Summary and Historical Background

DATSY is a data manipulation language developed for solving large economic models. Its historical background is sketched later in this section.

This report owes much to contributions made by participants at the seminar "DATSY -- Past and Future" held at the Central Bureau of Statistics in Oslo on September 26, 1978. Those who attended were Olav Bjerkholt, Hans Petter Dahle, Inger Henningsen, Anne Hustveit, Hanne Modahl, Frank Siljan, Kjetil Sørlie and myself. Frank Siljan's minutes from this seminar form the basis of the present report. I would like to thank Olav Bjerkholt and the Central Bureau of Statistics for inviting me to this seminar.

I decided to take this opportunity to collect the opinions of people concerned both with using and maintaining (or altering) DATSY. It should be obvious to those who compare the minutes with this report that my own thoughts have changed somewhat as a result of the seminar. Just the same, I have taken this occasion as a good one to put forward many opinions of my own, and any mistakes in this report are my responsibility.

A great deal of effort went towards exchange of information and experience in this seminar. I did not digest all of it immediately and I doubt that anyone else did either. There was not really enough time to discuss opinions about future development. Hopefully this report will help here.

I owe my Norwegian readers an explanation for writing this report in English. The seminar and minutes of course used Norwegian. My earlier reports on DATSY were in Norwegian.

There are two main reasons for writing in English. First, visitors to the Central Bureau of Statistics can usually read English well and this report may assist in helping communication on the topics discussed. There is a shortage of English-language material on DATSY. Also,

the people behind TROLL and APL should be informed in some detail that there is a market for extended versions of these systems with the additional feature of being able to handle much larger quantities of data efficiently and flexibly, as DATSY does already.

Summary

No clear message for the future can be obtained from this report. It is too early for that. Several interesting historical conclusions arise:

1) DATSY has facilitated an astonishingly high level of reliability in the implementation of very large economic models, as Anne Hustveit pointed out.

   I think this is mainly due to the use of standard commands on standard data structures.

2) Similarly, DATSY has facilitated rapid alterations in such models without any fall in reliability being noticeable.

3) Fewer distinct operations (commands) are needed than was originally expected, but DATSY programs are much longer than originally expected. The length is no great disadvantage, since the logic is largely sequential and the language has a subprogram structure.

4) The implementation of large economic models is divided into a number of separate programs for practical reasons. As Anne Hustveit pointed out, in some of these programs DATSY'S entire available disk space of 4 million characters is fully utilized. This corresponds to storage on line of 1 million numbers. (Note further that zeroes are usually removed from matrices, so that most of the data stored is meaningful, and that automatic garbage collection is in use.)

5) Manipulation of sets of records is as important a part of the language as matrix manipulation, and the two are closely interrelated to increase reliability and flexibility.

Salesmen should note that typical American products for similar uses such as TROLL or APL were developed for dealing with much smaller model implementations than DATSY is now used for. DATSY's main strong point is that it is a practical tool for implementing uncommonly large models without the usual collapse of either reliability or efficiency (see Sections 2 and 5).

Norwegian Government models for economic decision-making involve much larger quantities of disaggregated cross-sectional national accounting data than the equivalent American models, as Olav Bjerkholt pointed out. Furthermore, they have a much larger number of exogenous variables due to the fact that they are used more extensively for decision-making. The solution of equations occupies a tiny fraction of the total machine use. More machine time goes for example to reading in sets of records, or cross-checking of various kinds aimed at revealing human, software or hardware errors.

Historical Background

DATSY was developed by the Norwegian Computing Centre in cooperation with the Central Bureau of Statistics, in the years 1968-74. The foundations of the system were laid in 1970, and several years development was needed after this, to establish the set of commands which is now in use. Parallel with this work, many extensions were made in the original foundations. These extensions have resulted in DATSY's ability to handle large quantities of data reliably and with some degree of efficiency. The implementation languages are Fortran and Assembly.

The original initiative for this project came from experience with the large economic decision models MODIS II and MODIS III, described in Section 2, in the years 1965-68. During the planning years 1968-69 a number of alternative formulations of DATSY were discussed at some length.

DATSY has now been in productive use in the Research Division of the Central Bureau of Statistics for about seven

years. During this period enough experience has been gained to warrant redesign. A list of DATSY's strengths and weaknesses is needed to aid further discussion, and I have tried to provide this in Sections 3 and 4.

At the Norwegian Computing Centre, Sverre Spurkland was the leader of the DATSY project, and Eva Kristoffersen and Helge Totland programmed most of the system routines. At the Central Bureau of Statistics the project was led by Erik Aurbakken and Olav Bjerkholt, whilst Eldar Børsum, Knut Kvisla, Pål Lynum and myself programmed most of the commands. The design of many of the commands was established in long discussions with Olav Bjerkholt, Inger Henningsen, Anne Hustveit, Svein Longva and Odd Ystgård, who as users had a strong influence on the development of the system. Ola Jacobsen programmed the so-called "version generator" which made it easier to install new commands in DATSY. Lars Espen Aukrust laid the foundations for DATSY's extensive user documentation. Many others too numerous to mention made significant contributions at various stages.

Note

The remaining sections of this report can be read independently except where specific cross-references are made.

My suggestions for improving my 1975 solution of the long term planning model MSG-3 using DATSY did not fit into the seminar's time period, so I have added them in an appendix. (They are now mainly relevant to the construction of a new runtime system.)

Quite a lot has been written about DATSY, nearly all of it in Norwegian. I have not referred to any of this material here except the Handbook for Use of DATSY (in Norwegian) published by the Central Bureau of Statistics, but the reader should be aware that a lot of documentation is available.

2. MODIS III Using Fortran/Assembly Compared with MODIS IV Using DATSY

This section is based on Olav Bjerkholt's presentation at the seminar, which is in turn based on his own experience with these models.

The reasons why the Bureau wanted DATSY can be boiled down to two difficulties which arose with MODIS III, which was programmed efficiently using Fortran and Assembly Language in 1967:

1) Addressing of Matrices

   Matrices were denoted by names in MODIS III but had to be referenced by a file number in the computer program. This led to frequent human errors when altering the model, due to its size and complexity. These errors were often difficult to find, because for instance many different matrices had the same dimensions.

2) Addressing of Rows and Columns Within Matrices

   In MODIS III, rows and columns were denoted by sector numbers, commodity numbers and so on. These were not necessarily sequential. In the computer program, absolute row and column numbers had to be used. This led to extreme difficulties when rows or columns were added or removed, since this usually affected very many matrices and the change had to be made separately for each matrix.

There were other problems as well, but the above two are representative. These problems were solved by DATSY with a corresponding vast increase in reliability and flexibility in MODIS IV as compared with MODIS II and III. MODIS IV is not only much larger than MODIS III but also contains many more exogenous variables. MODIS IV became fully operational in 1973, being delayed by reconstruction of the national accounting system.

(As discussed in Section 5, the two problems mentioned above were solved by other languages developed over the same period or earlier, but these languages were unable to deal with the same quantities of data as DATSY.)

3. Strong Points in DATSY

DATSY fulfils its main design goals -- clarity, flexibility and reliability. It manages, but not with ease, the large quantities of data which are required in the implementation of a large economic decision model such as MODIS. At present I do not know of any other system which meets these criteria (see Section 5).

For detailed comparison, a checklist of strong points may be helpful (this is followed in the next section by a checklist of weak points):

1) Addressing of data objects by name, and addressing of rows, columns, records etc by name (see Section 2).
2) Grouping of data objects by name (to avoid naming explicitly several hundred data objects every time they are moved).
3) Self-explanatory programs.
4) Ability to operate on single data objects for which there is not enough room in core, in every case where this might be relevant (it is not usually relevant for a list of row names, for example). This also keeps the total program size under 64K words (36-bit). At the same time, unnecessary disk transfers during the execution of a command are avoided (but see later).
5) Ability to hold a large total quantity of data (1 million numbers in single precision) depending on available hardware, split into several hundred data objects.
6) Ability to set up complicated but useful standard data structures in a user-oriented way, and to add extra attributes to individual objects. All attributes of a data object follow it everywhere automatically. For example, lists of row and column names can be attached to matrices as attributes. Excellent report-generating features for displaying data structures.
7) Ability to execute tens of thousands of command sentences in a single run, divided into a subprogram structure with possibilities for loops and conditional hops.

8) Good potential (but no experience) for later development of a powerful user-oriented interactive system, using the same set of commands but different system routines (see also point 9 below).

9) The design philosophy is to stop the run at the slightest hint of trouble, with good explanatory error messages (in Norwegian). This philosophy has been carried through very successfully. Both the interpreter and the runtime system contain very many useful checks on the program, data structures and commands. These checks are made possible by the use of standard commands on standard data structures. As noted earlier, a high level of reliability in use has been the result, which is much appreciated by the users.

10) It is possible to set up programs in DATSY in such a way that alterations in matrix dimensions and similar changes are made at only one point in the program, to avoid inconsistency. This increases reliability and flexibility. Command sentences and declarations in DATSY do not specify object dimensions. These are either specified in the data section or are implied by the commands themselves. Only the data is altered when dimensions are altered (e.g. a sector is added to a model). Allocation of space in core is dynamic. There is automatic garbage collection.

11) DATSY is modular in the sense that it is easy to program new commands in Fortran and add them to the system. Ola Jacobsen's "version generator" makes this an easy and reliable process.

12) Large quantities of data can be filed on tape (but in a special format). Excellent report-generating facilities are provided for this file system. Tape labelling is used to increase reliability.

In summary, DATSY is a well-rounded system in which reliability, flexibility and clarity have been pursued with success.

4. Weak Points in DATSY

Whilst DATSY has fulfilled its main design goals of reliability, flexibility and clarity as detailed earlier, and has many significant strong points as outlined in the previous section, the following weaknesses have unfortunately become apparent during use. In spite of these weaknesses, DATSY has been a success in its role as a language for building large economic models.

1) Unnecessary disk transfers are made whenever a new command sentence is to be executed. (This causes slow progress through the machine when thousands of command sentences, each involving small quantities of data, are carried out as in MSG-3.) See the Appendix and strong point number 4 in Section 3. This problem caught us by surprise as we never expected to execute more than a hundred or so command sentences in a single run, during the planning stages of DATSY. Later it became apparent that it was very desirable to be able to execute a much larger number of command sentences, particularly in loops.

2) The ability of DATSY to execute tens of thousands of command sentences, the subprogram structure (i.e. the macrosystem) and the loops and conditionals are ad hoc additions to DATSY which should be properly integrated in a new version of the language. For instance, a loop is reinterpreted every time it is traversed. This does not matter if the computations carried out on the data are large and few in number, but if they are small and many the reinterpretation takes too high a percentage of the total machine resources used.

3) The definition of data structures is logically separate from other aspects of the computations, but DATSY does not yet allow data structuring to be separated from the other operations in practice. Also, data structuring in DATSY is batch-oriented, whereas this function could well be carried out interactively, as an option. To be carried out interactively, it would have to be separated out from heavy computations in a separate run. A suitable report generator would then be needed to follow up, which

DATSY already has. Interative alteration of data structures is another obvious need. Batch mode involves unnecessary delays here.

4) Input of data to create individual data objects is also a logically separate phase which could be separated out in practice, with savings due to not repeating the expensive operation of reading in large quantities of data. This activity can be divided between tiny objects which are easily handled interactively and large objects which cannot be handled interactively. As it is, both data structuring and data input are repeated unnecessarily in production runs.

5) There is a much used facility for filing a string of data objects with all their attributes on magnetic tape. Some of the advantages of this system are listed in Section 3, point 12. With the benefit of seven years hindsight we are now able to see that although the system has given many benefits (including efficient data management), it nevertheless suffers from the following problems:

   (i) A special file format is used, to yield tapes which could be read by DATSY systems on any other machine. However this feature is of little practical use to the Bureau, and it meant giving up easy communication with system software via standardized file formats, which has caused some problems. DATSY's file system makes DATSY a closed system. Since it involves a lot of assembly language, it also makes DATSY machine-dependent.

   (ii) Slow progress of some DATSY programs through the machine has caused operating problems due to the method of assigning tape stations. (E.g. output tapes have to be mounted at the beginning of the run, but are only written on at the end.) Valiant efforts by Eva Kristoffersen have improved the situation as Anne Hustveit pointed out, but not all the problems seem to be soluble.

(iii) Another difficulty with the tape-oriented file system in a special format is that interactive use is ruled out until a new file system has been completed. Originally we felt that interactive use would not be cost-effective in the foreseeable future, due to the large quantities of data and heavy computations involved in solving MODIS. Hence this difficulty has arisen only recently. A small data base project carried out for a time by Ola Jacobsen would have solved this problem, but was not carried right through due to his resignation. Jacobsen's data base system would have made it possible to bypass DATSY's file system and thus move towards an interactive DATSY.

I think our goal of machine-independent tapes mentioned under point (i) was a good one, but it is not worth these other difficulties which arose later as a result of our decision. I suggest radically different design goals for DATSY's file system later on in this report (Sections 9 and 10). At the same time an effort should be made to retain the excellent report-generating feature of the present file system in any new version of DATSY.

6) As noted in Section 3, the runtime system has the very desirable feature of checking many details of data movement and storage. However in some details this went too far. Most of these checks should be retained as a guard against both hardware and software errors. However, some of them guard against software errors in routines which are never altered. This raises the overhead per command sentence (see the Appendix). Ideally, testing of new commands should take place interactively using a runtime system much like the present one. In batch production however, I think a new runtime system is needed with the assumption that commands have already been tested fully. Machine errors are surprisingly common in runs of this size, and need to be guarded against as in the present version by repeatedly checking the structure of data objects.

7) It is I think a weakness of the present version of DATSY that no account has been taken of developing an interactive DATSY, when the potential offered by standardized commands and data structures is clear. As noted earlier, originally we felt that interactive use would not be cost-effective due to the large quantities of data to be dealt with. With the passage of time our evaluation has changed (see Section 6). Many of the problems inherent in interactive development and production runs are solved by DATSY's strong points. See Section 3, points 1, 2, 3, 7 and 8 as well as the general comments about clarity, flexibility and reliability. TSP on the other hand has fully utilized its potential for using both batch and interactive runs interchangeably.

8) The interpretation of DATSY programs takes too much machine time. Some time was spent in the seminar discussing why this is so. Historically the general reason is that DATSY's interpreter assumed that the total number of data objects, sentences and words of text in any particular program would be much smaller than it turned out to be in practice. This assumption made it possible to add many desirable features, such as checking for empty input objects in command sentences which in turn save a lot of unnecessary machine time. If the assumption had held then the interpreter would have been efficient. If the so-called "combined commands" programmed in Fortran had been useful, the assumption would have held. In fact they took too long to debug and were not used.

9) The present implementation of DATSY takes too much core space for easy use in the daytime -- about 64K 36-bit words. It is however much smaller than TSP would have to be for solving large economic models, and can in addition solve larger models than TSP. Of course, TSP is two years older than DATSY. The version of TSP used interactively in the Bureau is for econometric analysis and not model solution.

10) It was suggested at the seminar that DATSY included a number of features for which the Bureau had no use. For instance, it is possible in DATSY to read in matrices with row and column sums which are automatically checked. For the Bureau it is usually preferable and more reliable to read in matrices as sets of records (where each record has three fields: row name, column name and value). These can be compared automatically with lists of row and column names fed in separately, and can be displayed more effectively. Also, as Inger Henningsen pointed out at the seminar, the feature of DATSY which allows extra words to be included in command sentences is not used because it seems to reduce clarity. Presumably this is because it leads to longer programs which take longer to read. (An APL-like syntax might be preferable, as it would allow more direct programming of complicated matrix expressions.)

5. TROLL, TSP, APL and Simula

In writing this section particularly, I hope to be corrected if the evaluation of any particular system is wrong in any detail.

TROLL

Hans Petter Dahle pointed out at the seminar that he had been working with Lorents Lorentsen to convert the long term planning model MSG from DATSY to TROLL, which is available in Oslo. Whilst TROLL has many attractive features, not the least of which is that it is interactive, Dahle said that he had just sat beside a VDU for a whole day waiting while some of the data was fed into TROLL for use in MSG. This points very much in the direction I suspected, namely that TROLL has not yet been developed to the stage where it can handle easily economic models of the large size used by the Norwegian Government. If TROLL cannot handle MSG easily then MODIS will present further problems as it is a much larger model than MSG.

Since the seminar it has become clear that in spite of the difficulties experienced feeding in data, which are partly due to TROLL's implementation in Oslo on a small machine, TROLL is nevertheless a useful tool for solving MSG, partly because it is interactive and partly because it solves equations efficiently. Because the interactive feature is so desirable, a much aggregated version of MODIS will be implemented by the Research Division in the Bureau using TROLL in the near future. However the main version of MODIS will remain too large for TROLL unless TROLL is developed so that it can handle larger quantities of data efficiently and flexibly.

Such development seems to me to be a real possibility. It is desirable because it enables solution of economic models which are integrated with detailed cross-sectional accounting data.

TSP

TSP (Time Series Processor) was a forerunner of TROLL, and is available much more cheaply than TROLL. It includes equation-solving routines originally aimed at solving the Brookings model in the U.S.A. These are sophisticated. Large core space (say about 200K words) seems to have been an assumption of the programmer, and individual data objects as well as the segmented program have to fit into the available core space. Large numbers of data objects are however accommodated on disk. The language is easy to use, is available at the Central Bureau of Statistics, and has good user documentation.

TSP has the desirable feature that it can be used interchangeably either interactively or in batch. Operating conventions (e.g. maximum program size in daytime) can restrict this however.

TSP's main disadvantages compared with DATSY are that it cannot deal with individual data objects which do not fit into the available core space, that it cannot attach attributes to data objects to the same extent, and that it is heavily oriented towards econometric analysis based on time series analysis rather than detailed cross-sectional national accounting data.

Olav Bjerkholt pointed out that MODIS IV is different from corresponding models in other countries in that it has a large national accounting system attached to it. He added that this is probably the reason why other countries have not constructed languages with the same goals as DATSY, and the same ability to handle large quantities of data.

APL

Dahle also commented on APL, and pointed out that the file system connected to this language could not handle the data processing problems involved in solving MSG and MODIS, as it was too small. It is difficult to transfer data from APL to standard system files (a difficulty shared by DATSY). It seems likely that APL was not designed for the quantities of data involved in MSG or MODIS, although I suspect that the basic concepts of the language provide an excellent basis for development in this direction.

As with DATSY, the APL user formulates many problems directly in terms of matrix operations and similar. Surprisingly compact formulations result from intelligent combination of commands, in both systems. The syntax of APL is entirely different from DATSY's syntax.

APL lacks two features of DATSY in addition to its apparent inability to handle large quantities of data. First, it cannot handle sets of records including both numbers and text, to the best of my knowledge. This makes it difficult to deal with large matrices in a flexible way (see Section 4, point 10). Second, its functions must be monadic or dyadic. This is an essential part of its powerful syntax at present. Many useful standard operations available in DATSY are ruled out by this limitation in APL.

A powerful feature of APL is its ability to allow the user to program and modify macros, via the interpretation of data as program. DATSY has this feature in its macro system.

Due to its elegant and simple structure, I believe APL could be extended to meet all the objections above. It is one of the two main languages available with microcomputers and is widely used by econometricians.

APL and TROLL seem to be available only interactively. For model runs taking an hour of elapsed time this is inappropriate. However APL opens possibilities for parallel processing in matrix operations which may meet this objection, in time.

Simula

Simula comes into a different category from the above languages. Simula is a completely general programming language with wide approval among computer scientists across the world. Many describe it as the best programming language for general use. One may ask why it was not used for modelbuilding in the Bureau, when it was developed at the Norwegian Computing Centre.

I believe that had we evaluated the available languages in 1970-71 instead of earlier, Simula would have been chosen. Prior to this, its support and maintenance group was struggling to keep up with the rapid spread of Simula. Prior to about 1970, Simula was rumoured to have too many bugs to make it the basis of very large programming projects. Also, earlier versions did not have the efficiency of later versions, or so it was rumoured.

The situation since 1970 has been very different. Many reliable versions of Simula are available (remember that most Fortran compilers contain bugs still), and efficiency is reasonable considering the likelihood of obtaining shorter, clearer and more reliable programs than with Fortran. Dahle mentioned a figure of 30% slower running time than similar Fortran programs. DATSY spends vastly more than this overhead to achieve flexibility and reliability.

In particular, the ability of Simula to encourage intelligent formulation of problems could be expected to improve the programming of difficult problems, both as regards efficiency and clarity.

The ability of Simula to define and use procedures and data structures allows users to build up a library of simple commands, structures and data objects, perhaps identical to those available in DATSY. Simula can be used either as a so-called high level language such as Fortran, or a very high level language such as DATSY (these terms

refer to the distance from machine language). Its complexities can be hidden from the user.

It would be possible with Simula either to use the library of procedures directly within Simula, or to process a DATSY program.

It may appear that I recommend re-implementing DATSY using Simula instead of Fortran/Assembly as the implementation language. Defining DATSY as the user documentation in the Handbook for Use of DATSY, this is a reasonable interpretation. One would not however use Simula as if it were Fortran to create tables of data representing data objects, when the definition and representation of such objects is inherent in Simula and can be utilized directly.

Such an implementation would have to be supervised closely by someone who was capable of utilizing all the features of Simula, since these are not immediately apparent.

A major difficulty for the Central Bureau of Statistics is that Simula is not available on the machine allotted for the Bureau's use. Remote access to an IBM or Cyber machine would solve this problem.

6. Interactive DATSY

DATSY is at present batch-oriented. On the other hand, the basic concepts of DATSY give it strong potential as an interactive language. In particular this follows from its strong points 1, 2, 3, 7 and 8 in Section 3.

Further reasons for regarding DATSY as a good basis for interactive development are as follows:

1) There is a reliable and extensive library of commands already in operation which could be used directly in an interactive runtime system.

2) This library of commands appears to be unique in that when relevant it assumes consistently that there will not always be room in core for any individual data object. Hence the core size of an interactive DATSY could be quite small, say 24K to 30K 36-bit words, without giving up the ability to carry out trivial operations such as matrix addition on large data objects.
3) Many members of the Research Division in the Bureau know very thoroughly the commands relevant to their own work.
4) Similarly, they are already conversant with a large collection of structured data objects which they use in their work.

These points could be summarized by the statement that the Research Division is now in a position to make effective and efficient use of an interactive DATSY, as a result of seven years experience and development.

Even DATSY programs involving large quantities of data and large single data objects usually contain many sentences which cost little in execution and might well be debugged interactively. It is particularly noticeable in MODIS that a small proportion of sentences takes most machine time. However, production runs involving heavy use of machine resources are ideal material for batch runs.

There seem to me to be six main uses for an interactive DATSY:

1) Debugging programs including quite heavy use of machine resources.
2) Defining data structures.
3) Production runs using small quantities of machine time, and from which results are in urgent demand.
4) Editing and manipulation of relatively small quantities of data.
5) Display of relatively small quantities of data.
6) Debugging new commands.

Points 1, 2 and 3 require easy communication between batch runs and interactive runs. This has implications

for DATSY's filing system. It would have to be disk-oriented rather than tape-oriented as at present, in any case.

The justification for an interactive DATSY rests on these uses and the comparisons with TROLL, APL and TSP in Section 5. Point 3 speaks for itself. The other points above increase the productivity of the users by eliminating delays. They may also save considerable quantities of machine resources for the following reasons:

(a) Debugging could be arranged so as to avoid re-running programs with errors. It is only really necessary to back up to the point where the most recent error was found. (This depends again on DATSY's filing system.)

(b) Activities such as 2, 4 and 5 above could be carried out much more efficiently by small, totally separate programs which use little core space. (Again, DATSY's filing system would need to take account of this.)

The fact that production runs of MODIS are best run in batch mode is fully accepted. No one wants to sit doing nothing at a VDU for an hour while MODIS runs. However this is no reason why the development of such models should be limited to batch runs.

An interactive DATSY might specialize initially on a subset of the six uses mentioned above, for instance points 2, 4 and 5. Point 1 is the most demanding and could perhaps be omitted at first. Separate programs communicating through a common filing system would simplify development of an interactive DATSY.

An interactive DATSY would have to be designed to fit into the core space available in daytime for interactive use, as a prime objective.

It would be very desirable for DATSY's filing system to be based on permanent disk files used in such a way as to allow the user to stop a run in the middle, and start again later from the same point.

7. Advantages and Disadvantages in Developing Software Locally

This fundamental question needs a section to itself. Why, it may be asked, should one develop software such as DATSY in Norway when in principle it could be bought from elsewhere?

The brief evaluation given earlier of TROLL, TSP, APL and Simula indicates that the ideal finished product cannot be bought elsewhere, although this may be possible in the future.

In particular it needs to be pointed out that MODIS is unusual in its size, function and methods of use (see Sections 1 and 2). The main advantage in developing software locally occurs when local needs are different from those found elsewhere, as they are in Norway at present.

Another advantage of developing software locally is that correction of bugs in the software is usually faster if it has been programmed locally. This can be important.

On the other hand, it is usually cheaper to purchase a finished product. Also, the purchased product may be superior to a particular local product.

By using Simula the Bureau would have the best of both possibilities, since Simula has a high reputation, is used nearly everywhere, and local support and maintenance is available. On the other hand, the procedures and data structures required by the Research Division in the Bureau would need to be programmed, as they are not delivered with the standard Simula system. Also, Simula is not currently available on the machine used by the Bureau. This could be solved by remote access to other large machines in Oslo.

8. Who Owns DATSY?

By DATSY, I mean the system routines programmed, debugged and supported by the Norwegian Computing Centre, and the subprograms for individual commands programmed by the Central Bureau of Statistics. DATSY in this wide sense cannot be owned purely by the Centre or the Bureau. The same applies to DATSY interpreted as the contents of the Handbook for Use of DATSY, i.e. the basic concepts of DATSY including the commands.

Thus for instance, when I added the MACRO system to DATSY in 1974 I asked for and obtained approval from the Norwegian Computing Centre, since they appear to have some claim to ownership.

A new version of DATSY would be ripe for possible marketing, in my opinion. However this may not be practical in the circumstances. The advantages of marketing are that other users would probably discover bugs, saving the Bureau considerable difficulty, and that development costs might be covered in advance by a suitable contract, or afterwards by sale or hire agreements.

There are several difficulties in the way of marketing a new version of DATSY. One difficulty is that it may well be very much bound to the Honeywell-Bull H6000 Series machine like the present version. Another difficulty is that personnel are probably not available to offer maintenance and support. A third difficulty is the relatively small number of similar organizations which might use DATSY, especially since DATSY appears somewhat specialized towards very large economic modelbuilding at present. A fourth difficulty is that the user documentation and error messages are at present in Norwegian.

These difficulties can all be overcome, by using extra resources. This increases the risk unless other institutions can be persuaded to cooperate from the beginning. Since the Bureau has good contacts with most of the possible participants in a cooperative project (I am thinking here mainly of Statistical Offices in other countries), it has much better possibilities than the Norwegian Computing Centre for finding participants.

It would be natural, I think, for the Bureau to retain ownership of routines programmed by the Bureau, and to avoid paying for ownership rights to new system routines programmed outside the Bureau, since it has no practical use for ownership rights unless it wants to market DATSY itself. All the Bureau needs is the right to use DATSY.

If new system routines are programmed in the Bureau, marketing through an agent would create the same difficulties of a practical nature as if the Bureau marketed the routines directly. This is because marketing would create a need for support and maintenance by the Bureau which could probably not be provided. Cooperation with other institutions as an alternative to marketing might be advantageous if agreement could be reached to share the burden of support and maintenance. I think an agreement of this type should be very clear about the need to respond quickly to documented reports of bugs.

I may appear to be very pessimistic regarding bugs. Yet any cooperation will probably stand or fall on this point. The frequency of bugs found in a new large-scale software product by a new user typically follows a cascade (falling) curve. Another user getting hold of a completed and apparently debugged product often finds a new sequence of bugs due to different patterns of use, again in a cascade pattern. Bugs are rarely finally eradicated from any large software system. Support and maintenance must be a permanent ongoing arrangement. This is as true of Fortran as it is of DATSY -- very few systems are immune to this problem.

9. DATSY Tries to Do Too Much

We have now reached the stage in this report where DATSY's future is pushed into the foreground, and DATSY's past becomes less interesting except as valuable experience.

One lesson which is apparent is that DATSY tries to do too much. This shows itself in two ways:

(a) DATSY combines in a single program and in a single run, operations which could be carried out more conveniently by separate programs in separate runs. I believe DATSY should be divided into several independent systems. However, alterations to the existing version of DATSY should be kept to the minimum possible. I am most concerned that a new version of DATSY should be programmed as separate programs communicating through a permanent file system.

(b) DATSY tries to do some things which are done more efficiently and easily by standard software available on nearly all machines.

These two opinions are discussed below. Point (a) above is discussed first.

DATSY carries out the following operations:

1) Data structuring for new data objects, and display of these structures.
2) Data input for new objects.
3) Data input for old objects.
4) Interpretation of the program including cross-checking and display.
5) Running the program.
6) Storage of selected data objects.
7) Associated report generation.

All of these points could be tackled relatively easily by separate programs, run as seven runs not always in the sequence shown, provided these runs used a standardized file system for data objects (together with their attributes). There seems no reason why the file system should not also

store the program, as with the macrosystem in DATSY. Usually, not all points need be run (for example, production involves points 2, 3, 5, 6 and 7 only).

Each program could be the responsibility of a particular programmer, since it would be manageable in size and separate from the others. There is an increase in reliability to be gained from this decentralization, for the following reasons:

(i) System software will protect independent programs from overwriting each other in core.

(ii) Smaller programs are easier to test and debug.

(iii) Separate programs can be tested earlier and in ways which are not possible if they are always linked to one another.

(iv) It is more reliable to focus responsibility on single programmers than on a group. Short programs can also be studied by other programmers more easily, if the usual programmer is not available.

(v) Smaller programs place smaller demands on system software, e.g. collectors (link editors).

We come now to point (b) above.

The main role for existing standard system software in carrying out the seven functions for DATSY listed under (a) above is in the storage of data. At present, interpretation and execution are specialized tasks requiring special programs. Data structuring is a specialized task at present also, since the new data base systems are probably not adequate yet for the purpose of economic modelbuilding.

However, points 3 and 6 above, namely data input and storage, leave much scope for existing system software.

To begin with, many machines have a filing system which stores data according to name and subnames so that absolute addressing of data objects is a thing of the past. But attributes do not usually follow automatically. Even so, extensive use of a standard filing system would greatly simplify the programming of a new filing system for DATSY.

(Since most machines have filing systems with names and subnames at least, DATSY concepts would still be machine-independent.)

Second, extensive use of a standard file system should reduce the core size of any future DATSY and make it less machine-dependent by reducing the quantity of assembly language programming.

Third, most machines have efficient and easy methods for copying standard disk files onto tape and keeping track of them there. Since DATSY copies all tapes to disk before using them, it would be natural to use this software, which is usually highly developed.

Fourth, movements are being made in the computer industry towards tape interchangeability between different types of machine. Non-standard files will probably be left out of any such development. Hence, machine independence will in the long run depend most likely on the use of standard system file structures.

I believe that DATSY should have no responsibility for tapes as such, and that it should be disk-oriented. Copying of disk files to tape can be left to system software. Also, I believe that the disk files should be in a standard format and utilize the naming features of the machine's file system. Far from making DATSY machine-dependent, this would make it easier to move DATSY to a new machine, since DATSY would contain much less assembly language than it does at present.

Obviously, tape will continue to be used extensively with DATSY for some years to come. DATSY already operates on the basis of transfers between disk and tape, and good system software usually exists for saving disk files and restoring them later.

MULTICS

Some machines have available advanced control languages which set up files and execute programs in a self-explanatory

and user-oriented way. MULTICS, developed at MIT around 1967, is available commercially through Honeywell and fills this role. Such a control language allows a more decentralized run-time system for DATSY in which individual sentences result in separate runs, without the user being aware of this. That is, each sentence starts a completely independent job or task in the machine which can run in parallel with others under a time-sharing system, and which is protected from the others. A new DATSY could make extensive use of the features available in MULTICS.

By splitting DATSY runs into many separate tasks as indicated above, reliability is increased since they cannot interfere with each other. The tasks would probably have to communicate with each other via disk, although hopefully they could share a limited area of core. New DATSY commands could then be kept isolated from existing ones during testing, so that they could not spoil the operation of other parts of the calculation. Software errors would be much easier to detect and hardware errors would do less damage than at present.

Communication between commands would be via standard data structures which could be interrogated and altered from a terminal if necessary. Error detection would be made easier by this possibility.

MULTICS offers many further advantages, one of which is parallel processing in the following sense. A DATSY program could be split into groups of sentences instead of single sentences, organized in such a way that some groups of sentences could be computed in parallel whilst others had to be computed in sequence. By creating (hidden from the user) separate tasks or jobs for the groups which could be run in parallel, these could be run simultaneously under the time-sharing system. Hence, the progress of a large DATSY run through the machine in elapsed time would be considerably reduced.

At the same time, general machine throughput is improved by splitting up large jobs into smaller jobs, particularly if the smaller jobs use less core space as well as less computing time.

10. Suggestions for Improving or Replacing DATSY

At the seminar I suggested that there is no reason to go on using a system simply because it was expensive in its development phase. Also, I advanced the idea that the easiest way to improve or replace DATSY was to split it into separate programs first (see Section 9, point (a)). Quite a lot of discussion resulted. My point of view is set out in the previous sections, so I will concentrate here on the discussion during the seminar, followed by a new concrete suggestion.

The discussion showed that in particular Hans Petter Dahle was less willing to split up the existing version of DATSY than I would have been in his place. This is perhaps natural, as I have the benefit of having "operated" (in the medical sense) on DATSY earlier. Since the seminar I have come to the conclusion that Dahle is right to the extent that further "operations" on DATSY should be avoided if possible. My concrete suggestions for altering the existing program now involve only the programming of new commands, using techniques which should be clear from Ola Jacobsen's "data base" system.

Dahle appeared to be interested in replacing DATSY completely, and rejected my opinion that DATSY should first be split up. His reason was that it would take too much time to split up. I agreed that splitting up DATSY was difficult. Against this I argued that it would be easier to replace DATSY if it were first split up. I greatly admired Dahle's understanding of the problems involved in "operating" on DATSY, and his willingness to replace DATSY. At the same time I felt that Dahle underestimated the size of DATSY and the effort involved in replacing the whole system at one bite.

This exchange of opinions is less relevant than previously in view of the following new suggestion: Instead of splitting up DATSY, I suggest that a new command be programmed to extract all data objects from DATSY and transfer them to a new file system. This file system would have to be developed first. A new runtime system could then be developed independently and separately from the existing DATSY.

This suggestion and its further ramifications are sketched below. It is meant as an input to further discussion and for this reason is not given in full detail.

The new runtime system would communicate with the old DATSY by receiving data and program through the new file system, but it would run as a separate program. It would not send any data to DATSY, and the results of its runs would be kept in the new file system. DATSY's runtime system would continue to be used until the point of time when the new runtime system was found to be superior. It would then fall out of use but remain as an option, for a time. The economic models and the way in which they are programmed could remain unaltered if desired.

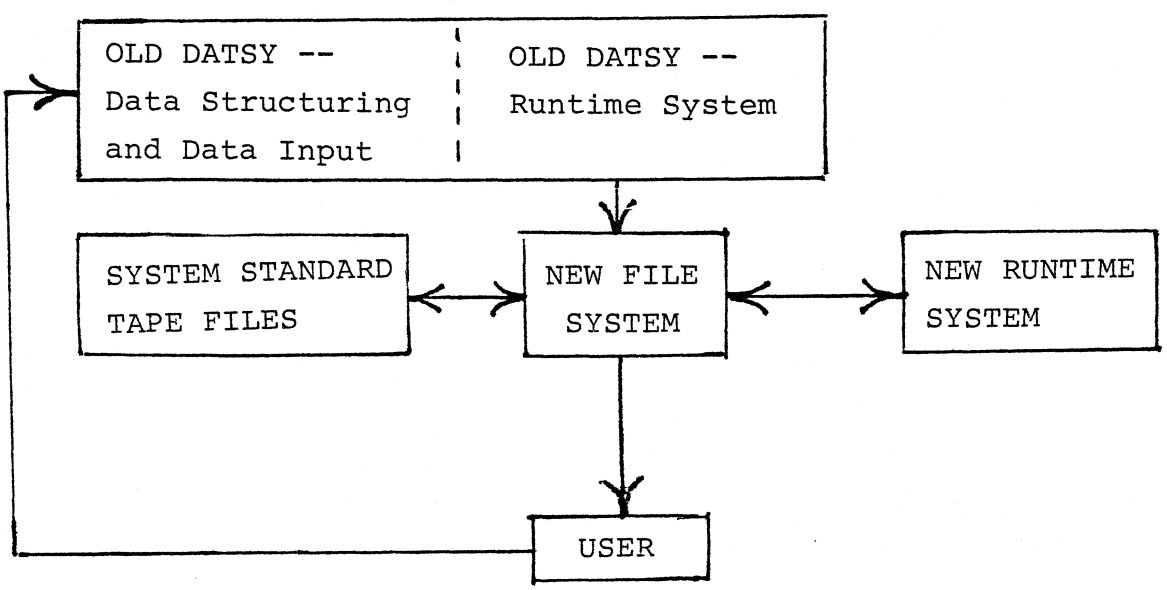
In case for some reason it became essential to move data back from the new file system into the old DATSY, the new file system should have a facility for writing data out in Hollerith so that the old DATSY can read it. This should however be largely unnecessary. The file system should in any case have this feature in order to communicate with other software, although there are naturally better means in particular cases. Data should only be fed back into the old DATSY via Hollerith as an emergency measure since this process is inefficient.

The benefits gained from the new file system can be summarized as follows: First, small errors in model development runs could be corrected interactively, in a way which would speed up development and reduce development costs. Second, further development of the runtime system is made possible in an operational environment without disturbing the rest of DATSY, which means that a new runtime system could be finished much earlier than if DATSY was replaced all at once. Third, the possibility of interrogating data objects interactively immediately before, during and after the runtime phase makes it much easier to detect and correct software errors. Fourth, all interactive use of the set of DATSY commands seems blocked unless a new file system is made operational. All the benefits of interactive processing can be made available as an option during the runtime phase.

How would the results of production runs using the new runtime system be input to later production runs? The simple answer is that they would not usually be input to the old DATSY system. But, it is easy to input them to the new runtime system. The structure of these data objects is already clear, and all that is needed to use them in the new runtime system is command sentences which the new runtime system could interpret without any help from the old system. Possibilities for "compiling" DATSY programs exist which could be developed in this new environment.

Later runs can use (mixed together) data objects transferred from the old DATSY system, and data objects produced by the new runtime system. Thus the input of new data from outside the machine still occurs via the old DATSY program, as a first stage in development. Later, of course, this function can be taken over by new programs as well, with continued use of the old DATSY available as an option quite compatible with the new system.

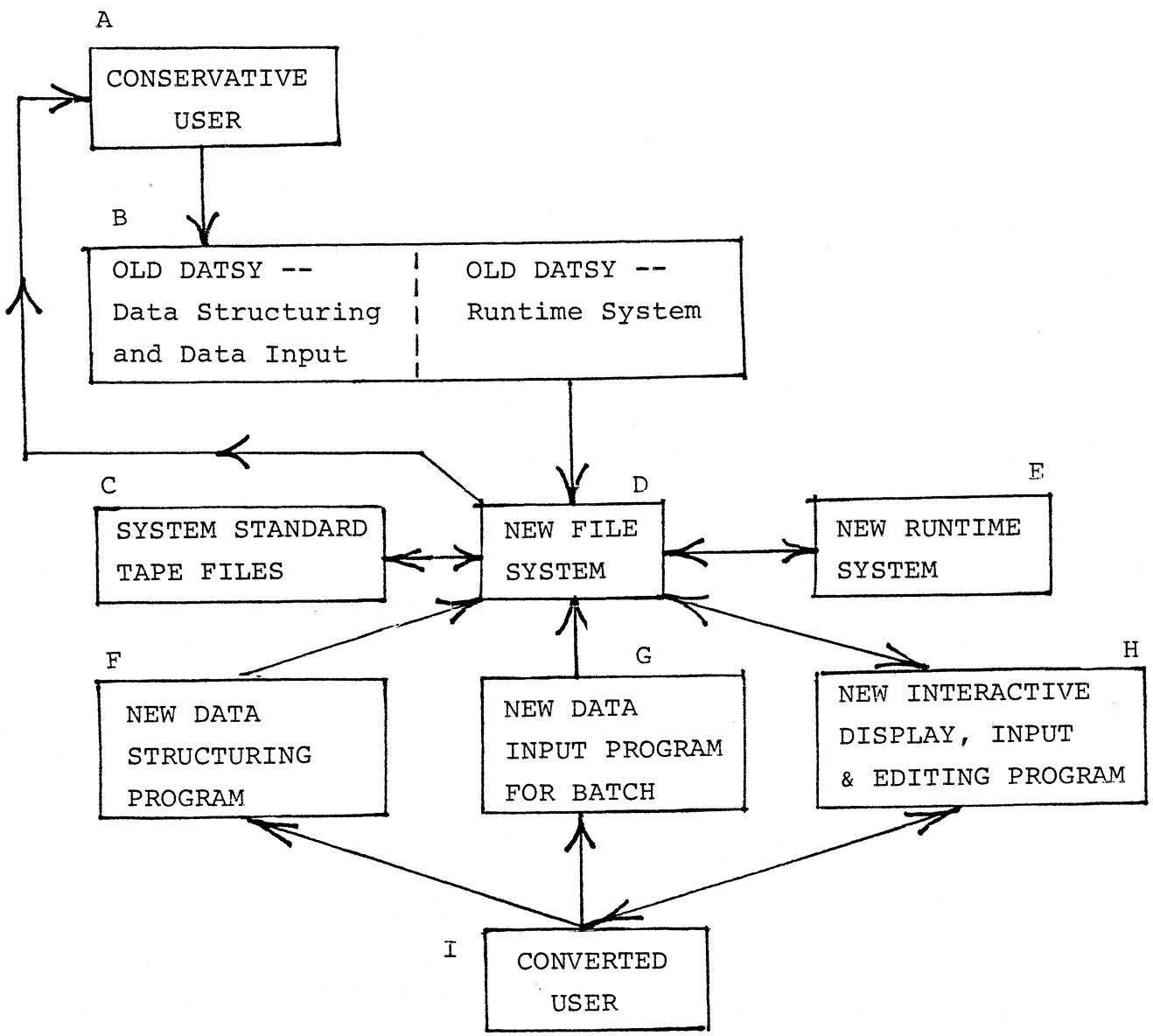
This first stage of development is illustrated below. In this diagram the arrows represent transfer of data objects. The old DATSY runtime system is used purely to transfer data objects to the new file system via the new command suggested above.



The advantages to be obtained from a new runtime system follow from the criticisms of DATSY in Sections 4 and 6.

Later, or parallel with the first stage of development sketched above, separate programs for defining data structures and for input of large and small objects could be added. Then all of the old DATSY would probably fall out of use. But there is no reason why it should not remain possible to continue to use the old system for these functions if desired. In short, I would hope to see a competitive relationship between the old and new systems as well as continued reliable operation.

The diagram above would then change as follows at the end of this second stage of development:



Programs would be defined as data structures using box F. They would be input as data objects using boxes G and/or H, and run from the filing system D where they would be stored. As before, arrows represent transfer of data objects in the above diagram. The system makes no fundamental distinction between program and data. In this way it is similar to APL. Programs are alphanumeric lists, as macroprograms are in DATSY. There seems to be no reason to make them a special class of data object, since the user who tries to execute a list which does not contain a program will receive adequate error messages.

The new programs for defining data structures and input of data could be written in any suitable language. The new runtime system would probably be in Fortran, to make it easy to call existing Fortran subroutines implementing the various commands.

The proposal put forward here really shifts the debate to the issue of a new file system. Easy conversion of stored data from the old system to the new one is assured by this proposal.

I note from the minutes of the seminar that Dahle argued against my suggestion that the machine's standard file system should be utilized in a new version of DATSY, since he believed that DATSY would no longer be machine-independent. He did say something like this, but I am sure he realizes that DATSY as a whole is not at all machine-independent except in its basic concepts and in the kind of magnetic tape it writes. (See my arguments in Section 4, point 5.) Also, I believe the goal of writing machine-independent tapes is better pursued via standard system file formats.

I argue that on the contrary, utilizing the machine's standard file system will reduce the quantity of assembly language in DATSY and thus make it much less machine-dependent. Nearly all machines have file systems with names and subnames which are named directly in executive request instructions. Why not let Fortran write system standard format binary files? Most machines allow the user sufficient control of these files through Fortran via special system routines. These routines usually have equivalents on other machines.

I think DATSY should have its own file system with extra features, based on the machine's standard file system. The desirable features of attributes, object definitions and descriptions which follow each object should be implemented. They might well be linked logically to their objects rather than physically. Then they could be kept physically separate in a dictionary file as at present. However it might be simpler and equally effective to store the attributes, definitions and descriptions physically together with the data itself for each object. Of course some attributes and descriptions refer by name to other data objects and these other data objects would be stored physically separate like any other object.

There are many choices which remain open here. It does not matter for my proposal whether each data object has its own file, or whether groups of them are stored as a single file as in Jacobsen's data base system. I originally viewed this data base system as a possible new file system for DATSY. Since it was not carried through due to Jacobsen's resignation it would probably be easier to start afresh with a completely new file system, but this is not certain.

As Frank Siljan pointed out, it would be desirable to keep an eye on the core space required by any new DATSY and to make this flexible to give faster turn-around for small runs. I would like to see the entire data buffer for the runtime system vary dynamically in size so that small runs automatically took very little core space.

The documentation produced by the Norwegian Computing Centre contains a thorough discussion of many points of design which are still relevant to new versions or replacements of DATSY.

Appendix. How to Improve the Solution of MSG-3 Using DATSY

Background

The suggestions which I put forward below are a little late, since MSG is now being converted to TROLL rather than DATSY. Nevertheless, I decided to include them here since they throw some light on DATSY's solution of MSG-3, and because they indicate how an experimental interactive runtime system for DATSY could be set up using existing commands from DATSY. However, such an experimental runtime system would not be possible until a new file system for DATSY is operational.

The suggestions made here are not really relevant to the solution of MODIS by DATSY, since they are meant to solve the problem of DATSY's high overhead resource cost per command sentence, which is unimportant in the solution of MODIS. In the solution of MODIS, large data objects are involved so that DATSY's overhead per command sentence of 80 milliseconds CPU time on the H6060 is only a tiny proportion of the total CPU time.

In MSG-3 as solved by DATSY from 1975 to 1978, there are many more command sentences executed than in MODIS, often in loops. Each command sentence often operates on very small data objects (such as a 2 x 2 matrix) although some larger data objects are also involved. Thus repeated references to disk are undesirable and serve merely to increase the CPU time and elapsed time needed to solve MSG-3. The present version of DATSY copies the results of every command sentence to disk and obtains its input from disk. Also, as discussed in Section 4, point 6, DATSY's overhead cost per command sentence is high because some unnecessary checks are carried out by the runtime system, which should really be restricted to debugging runs and omitted in production runs.

The suggestions below bypass both these problems by showing how to set up miniature runtime systems within commands, which are for data objects small enough to remain in core. Several commands are combined into a single command so that the overhead per sentence due to DATSY

is spread over several command sentences. Also, the elapsed time is reduced considerably by eliminating unnecessary disk transfers.

Any new runtime system for DATSY should take note of the problem solved by the suggestions below. It should of course be solved within the basic design of the runtime system rather than by ad hoc additions as suggested here. The same comment applies to the MACRO system within DATSY. However, since DATSY is large and operational, ad hoc additions are sometimes justified.

A general conclusion which emerges is that DATSY's new file system should not update the disk after every command sentence. It is necessary here to choose a suitable trade-off point between the elapsed time delay caused by excessive disk transfers, and the desirability of keeping the disk as up to date as possible (to help debugging among other things). Perhaps it would make sense to update the disk when either

(a) there is a shortage of core space after automatic garbage collection, or

(b) a certain quantity of CPU time is registered, such as half a minute.

Point (b) removes the necessity to repeat long calculations leading to an error when debugging, which is expensive when developing a model such as MODIS.

It would make sense for the quantity of CPU time mentioned in point (b) above to be under the user's control, so that it could be reduced for interactive debugging and increased for production runs. This would reduce costs.

The Suggestion

I have not spelled out this suggestion in detail, because it is no longer relevant to the solution of MSG as noted earlier. Hence it is not intended to be taken as a blueprint, and merely serves to focus attention on problems which a new runtime system should solve, almost certainly in a different way.

It is easy to combine subroutines from existing commands to make new commands. To reduce interrupts due to excessive disk transfers and bypass a number of runtime tests it is possible to write new so-called "command" and "test" routines which refer via a new "work" routine to a string of existing "work" routines (these three types of routine are standard within DATSY). By setting parameters correctly (see the relevant documentation by Helge Totland) it is then possible to force data objects to remain in core during execution of the new command. The system routine SID is called once only, to execute the new "work" routine which is really a miniature runtime system in itself.

DATSY has a mechanism for combining commands by Fortran programming which was little used, because it took so long to debug the results. This was the method of "higher order" commands, that is, a hierarchical combination of commands. Attractive as it seemed at first, I think this method is to be avoided as it does not reduce overheads noticeably although it does provide facilities for forcing objects to remain in core.

What I suggest here is simpler. It involves creating a new first order command from the "work" routines of existing first order commands. The method is as follows:

First, using an editor, take the "work" routines to be combined and rename them systematically. Then, using a global exchange instruction in the editor, change all IUR-routine references to IOR__. E.g.

```
    CALL IURW3( ...
```

becomes

```
    CALL IORW3( ...
```

(Of course some other combination of letters could be used consistently.)

IDIMA, IDIMB and similar entry point names referred to in the "work" routines should be altered systematically as well. This only concerns the "work" routines and not the "test" routines.

The new "work" routines for the new combined directive and now finished (after recompilation) except for the new miniature runtime system, which will also appear as a "work" routine. The intention is that instead of calling the usual IUR-routines the new "work" routines will call new IOR-routines which use core exclusively and not disk. The new IOR-routines should also neglect many checks carried out in the old ones. I assume here that the "work" routines converted in this way are fully debugged -- after all they come from an operational economic model which has run for three years. For this reason, further alteration of the work routines is risky and the alterations stated above should only be carried out using global editing instructions. Lines should not be replaced or altered individually.

The role of the miniature runtime system is to create arrays whose start addresses will be passed to the "work" routines, and to place and receive data in these arrays. If the "work" routines assumed that all data objects were always in core, that would be all. However, the "work" routines do not assume this. They call on IOR routines which must therefore be provided to fetch the data. The IOR routines must emulate the IUR routines, seen from the point of view of the "work" routine. Of course they will be much simpler since they do not have to deal with disk transfers.

How are the arrays created? The total length of array is calculated in a new "test" routine written for the occasion for the new combined command. Objects which are input to the new combined directive are treated as usual in the "test" routine and the same goes for output objects. They will be handled by DATSY.

Some data objects are output from one "work" routine and input to another, and are not input or output to the new combined command viewed from the outside. The user will not see them. Normally they would be treated as "dummy objects"

To avoid overheads, we create a single "dummy object" in DATSY of class zero for the combined command, force the object to remain in core, and dimension it in the new "test" routine so that it is big enough to be split up into separate objects by our own runtime system. We make sure it has room also for all the input and output objects mentioned earlier. If there is any difficulty due to size, DATSY will stop safely in the new "test" routine and tell the user.

To save as many overheads as possible, the commands being combined should yield a new command with few input objects, few output objects and many potential "dummy" objects.

The new "test" routine calculates as usual the size of each array. Instead of telling DATSY, it passes this information via COMMON to the new miniature runtime system from which the "work" routines are called. But it does set the dimensions of the output objects via DATSY in the usual way.

So much for the new "test" routine. How do we write the new "command" routine? It calls SID once, referring to the new runtime system which appears disguised as a single "work" routine. The objects named are the input and output objects, and our single "dummy" object. Savings in overheads result from calling SID with fewer objects and only once, as compared with the existing situation.

Let us now focus on the miniature runtime system routine. It calls the original string of "work" routines, one after another, directly. It creates data arrays for their use by allocating parts of the single dummy object. It copies input objects into their correct places in the single dummy object before it calls any "work" routines. Later, it copies final results from the single "dummy" object array to the output objects.

It receives the size of each array via COMMON from the test routine. It decides on the position of these arrays in the single "dummy" object array. Integer variables are computed giving the first word of each array within the dummy object. Let these integer variables be MAT1, MAT2, MAT3,

MAT4, J1, J2, and JANNE. Let the single "dummy" object array be called SPACE. The input objects are copied into the right position in SPACE. Then the work routines are called as in the example below:

```
   :
   :
CALL ADDMAT (SPACE(MAT1),SPACE(MAT2),SPACE(J1))
CALL MULT (SPACE(J1),SPACE(MAT3),SPACE(JANNE))
CALL SUBMAT (SPACE(JANNE),SPACE(MAT1),SPACE(J2))
CALL KOLSUM (SPACE(J2),SPACE(MAT4))
   :
   :
```

In this example, we add MAT1 and MAT2 to give J1, which we multiply on the right by MAT3. The result is put into JANNE. MAT1 is subtracted from JANNE, with the result being J2. The columns of J2 are summed and the result is MAT4, which has a single row.

In this way we allocate a single array to serve as many arrays with variable lengths. The altered "work" routines find the dimensions of the matrices by calling on the new IDIMA and IDIMB routines (with altered names). The only alterations in the "work" routines are their names and the names of the routines they in turn call.

The next part of the suggestion is that IOR-routines (see earlier) be constructed to deliver data as required to the work routines. All this data is in core, and could be referred to directly but to do this we would have to rewrite the "work" routines and this would introduce unnecessary errors, so we do not do it like that.

In order to emulate the IUR routines, extra core buffers are needed. The single "dummy" object can be doubled in length to provide these (later we will see that it may as well be tripled at the outset), so that every new "array" created within it as in the example above has a "backup" array which could be used for example to refresh it, when emulating rewinding of a data object. Information regarding

the length of the single "dummy" object array would have to be passed via COMMON to the IOR routines. They would then be able to work on the basis that when asked to process an object beginning with address X, they could also use a core buffer with start address equal to X plus (say) half the length of the single dummy object (later on, it will be one third of the total length). I.e. when asked to process the array with name A, they could use A(1+length/2) and upwards as a suitable core buffer. It would be natural to keep a copy of each array here and bring down data to A(1) as requested by the "work" routine.

This may seem to imply a lot of unnecessary movement of data in core. In fact using Fortran these movements are unfortunately necessary, and for instance the existing DATSY routine IURW3 moves data around in core in addition to transferring it to and from disk.

If the implementation language allowed us to interfere with pointers to arrays in a suitable way it would no longer be necessary to move data around in core in this way. This is where for instance JOVIAL is superior to Fortran as an implementation language.

Note that this solution is really very simple. The IOR routines do not have to keep track of all the different arrays we have created in the "dummy" object array, via their addresses. The IOR routines are given the starting address of all these arrays implicitly by the "work" routines which call them. Hence the IOR routines can be written for a single array called for instance A as above, and will then work for all the new arrays created within the "dummy" object array.

Similarly, the "work" routines refer to the different arrays as A, B, C and so on as at present (they are not changed, remember). In the example given earlier, SPACE(MAT1) corresponds to the array A in ADDMAT, SPACE(MAT2) corresponds to the array B in ADDMAT, and SPACE(J1) corresponds to the array C in ADDMAT. ADDMAT then views its task as adding matrices A and B to give C.

However, the IOR routines do have to keep track of the status of each object -- i.e. whether the object is currently being read from or written to by the "work" routines, how many rows have been dealt with up to now, and so on. By tripling the size of the "dummy" object array such information can be stored there as well. It wastes a little space (of which we are not short according to our problem formulation) but is very simple to triple the length of the "dummy" object array and store all extra information including object dimensions for array A at A(1+2*length/3), with the previous core buffer mentioned earlier at A(1+length/3). The row of the data object currently available to the "work" routines begins at A(1), of course. The last one third of the array must be large enough to hold this status information for each object. This can be arranged by always allocating at least a certain minimum number of words for each object.

Information stored in the last one third of the "dummy" object array will be of two types -- dimension information used by the new IDIMA and IDIMB routines (with altered names), and status information showing the stage which reading and writing of each object has reached. Of course, this refers to an emulation of disk reads and writes using only core, which is why the extra copy of each object in core is needed.

Using the simplifications above, the IOR routines would be very simple and treat all data objects alike, as if in fact there were only one data object, although of course there are many.

Most commands in DATSY use only IURD3 and IURW3 among the IUR routines, so that emulation of IURD2, IURW2, IURD1 and IURW1 may not be necessary since not all commands will be involved in these combinations. However it would not be particularly difficult to emulate all these routines in any case.

As noted earlier, these suggestions are no longer relevant to MSG, but are included here as input to the general discussion about replacing or improving DATSY.