Software functional variability

Gill Ringland

Computer Analysts and Programmers Ltd, 14-15 Great James Street, London WC1N 3DY

A strategy of design for reliability in dedicated real-time systems is defined. Reliability of this type of system implies the ability to vary operating system functions and application program functions online: the software structure described uses a nucleus of fixed code to support a modifiable system. The conditions under which such modifications may be safely made are discussed, and the hardware and software requirements for the system outlined.

(Received October 1973)

This paper is concerned with a problem which is of vital importance to current and future realtime systems. The problem is how to maintain such a system, after the date it goes live. If the date of obsolescence can be postponed by designing the software appropriately, or by using the hardware with this aim in mind, then real money can be saved and the disenchantment of the user community reduced.

We do not have a solution 'at a stroke'. What this paper does contain is a suggested framework within which dedicated realtime systems can be constructed, so that they can in all probability be more easily modified without losing reliability.

The next section discusses why the problem of modifying realtime programs arises, and defines the term 'Functional variability'. The following section describes the design strategy and the software structure which is proposed to implement it. The use of the structure for failure handling and for varying the function of a realtime system is described, and the problems of sharing hardware and software for testing and live running are briefly discussed. Finally the implications of the strategy are considered in terms of hardware resources, software production, operational aspects of failure handling, and techniques for modification.

The use of this strategy for building a pilot system is discussed in a companion paper (Ringland *et al.*, 1975).

The nature of the problem

The problem may be briefly stated. Realtime systems, like batch systems, need to be modified after they go live. However the techniques developed to test and introduce modifications into batch systems cannot be applied directly to online systems. In this section we discuss the particular problems in maintaining reliability in a realtime environment. First, however, it should be made clear that this work is in a sense orthogonal to that aimed at proving the 'correctness' of programs (see, for instance, Elspas *et al.*, 1972). This work starts with the assumption that programs have bugs, and attempts to establish methods by which the effect of these may be contained—by fault tolerance, and by providing methods of correcting online software.

The number of such errors expected increases faster than the volume of code, because of problems in communication on large projects. On such projects, therefore, a greater proportion of the total development effort should be devoted to testing, compared with smaller projects. Realtime programs, however, pose an additional inherent problem, namely, that of the non-repeatability of errors (Fergus and Taylor, 1971). This combination of difficulties means that large realtime programs will almost certainly go live while containing errors.

At the same time, users of such programs expect that realtime systems should above all be dependable. Hence the need is apparent for methods to correct such initial software errors after the system has gone live. It is often not possible to delay the application processing while the corrections are introduced into the software. This means that, to improve the reliability of realtime systems, it is necessary to develop techniques for varying the software while sustaining system operations.

Reliability in a realtime environment

Reliability may be measured by the inverse of the frequency with which the user receives unsatisfactory service.

In batch processing applications, it is possible to increase reliability by repeating work which may have been faulty. The method relies on restarting a coherent batch of work from a checkpoint or a suitable dump (Wilkes, 1968).

In realtime applications, it is not possible to define a batch in the same way. The jobs are started in response to requests from the outside world, and they share access to data. Reliability in a realtime system is, therefore, provided by protecting the current data against loss or corruption, and by ensuring that the software as a whole can continue to function despite the presence of errors.

If an error is detected, a small unit of work, identifiable in user terms, should be failed. This may mean that some data is no longer valid, and should be recovered—for instance, by use of an audit trail. The achievement is that the software structure

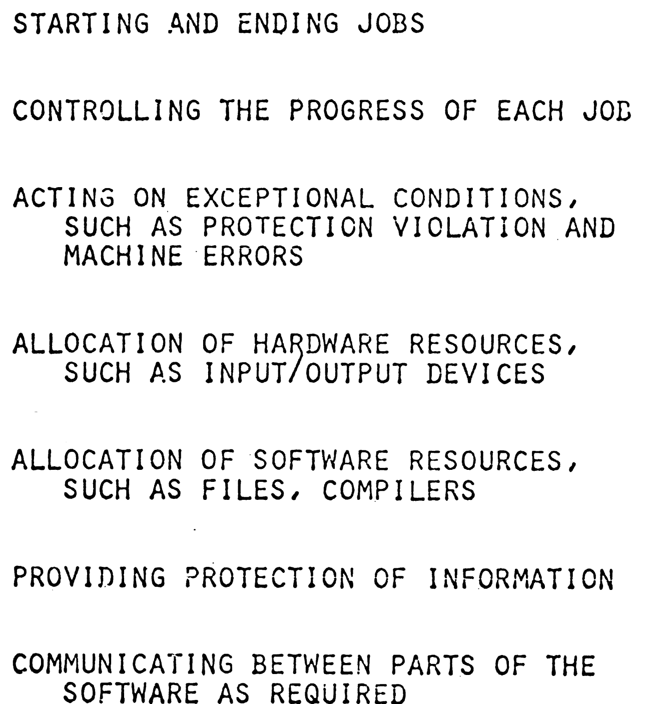

Fig. 1 Functions of an operating system

Downloaded from https://academic.oup.com/comjnl/article/18/4/312/347989 by guest on 19 April 2024

survives; or, in the rare cases when this is not possible, that the software will be restored to a well-defined state in a delay time which is acceptable to the users.

Such a system is said to be reliable.

Reliable operating systems

Operating systems can be described (Denning, 1971) as those parts of the total software that control the computer itself, as distinct from user programs that process the application. The functions of an operating system are summarised in **Fig. 1.** While the user programs are expected to change to reflect variations in the application, the operating system is often assumed to change only rarely, to correct faults found after the system has gone live.

Experience shows that faults in the operating system are often more difficult to locate, and more far-reaching in their effect, than faults in user programs. They are also difficult to correct for most configurations without causing a break in service. In addition, it is important to be able to extend the operating system to deal with new devices or to service changes in the user programs. Thus, to ensure the reliability of the system as a whole, it is necessary to develop techniques to vary the operating systems on-line.

Among the theoretical work on operating systems (e.g. ACM, 1969) little is concerned primarily with reliability. An important current idea which has, however, been adopted in the approach to the problem is that of a small nucleus of code. This nucleus has been defined to be itself invariant, and to provide the rules by which a variable structure can be built.

Functional variability

By this term we mean a methodology for achieving reliability in computer software by permitting online enhancement. The methodology may be used to add functions to the software structure as well as to maintain it: this is absolutely fundamental.

The idea of functional variability was first formulated about five years ago by d'Agapeyeff and Clark (1968; 1970). They outlined why maintenance techniques for on-line systems were at least as important as initial development techniques, and the differences between on-line and batch systems in this respect. Since then our work in the field of reliability has shown that it is not only necessary to correct faults to maintain reliability, but also to be able to handle extensions. It can happen, for instance, that a system reaches early obsolescence because many small extensions have caused a breakdown in the original structure. If these extensions can be accommodated within the structure of the existing software, the date of obselescence can be postponed.

The variations to be expected in a system during its lifetime can be of three types. The requirement may change in a way that implies more processing of the same type as before—for instance, a higher message density. This may require extra hardware to be added to the configuration. It may be required to add functions to the software or new hardware to the configuration. For additions of this type, testing facilities must be provided in a manner which decreases the probability of the live system being corrupted while testing is in progress. Although a different hardware configuration has been traditionally used for testing and for live running, BEA's CALC Cargo Allocation system successfully uses the same equipment for both functions. Thirdly, it may be required to extend the software or hardware in a way which alters basic features of the computer organisation. This type of change usually involves a rewrite of a large part of the system software.

A functionally variable system can accommodate changes of the first two types, and go some way towards accommodating changes of the last type.

The structure of the approach

The scope

A general strategy for creating functionally variable realtime systems does not exist and has not been attempted. We had two reasons for adopting a more particular approach. The first reason is that a theory divorced from practical application considerations was not regarded as suitable for solving real problems in a given environment: it can be more constructive to alleviate the more urgent problems with realistic constraints. Secondly, reliability is by its nature a relative rather than an absolute or general concept. That is, by introducing redundancy, validity checks and so on, the reliability is increased until it is acceptable to the users.

Functional variability was described above as a methodology for designing a reliable realtime system, which can be modified. This paper concentrates on a strategy for a class of applications and a model configuration. The application of the methodology to a particular application and hardware is described in a subsequent paper (Ringland and Trice, 1975).

The strategy has three aspects. The first is to recognise the unit of work in the system being designed. This unit will be recognisable in user terms, so that in the case of failure the unit(s) affected may easily be reported by the system. The second is to unify the techniques used for error recovery and for introducing changes into the system. If these are considered together when the system is designed, it is expected that simplifications will result. The third aspect is to define techniques for sharing common resources—code, devices, data—between a trusted or live system, and a test system. The response observed on the live system should ideally not be degraded by the existence and use of a test system: the advantage of sharing resources is the extent to which modifications can be tested in a realistic environment.

The class of applications can be categorised as those for which a large number of relatively low density peripherals interrogate or update a volatile database, and for which acceptable response times and down times are of the order of seconds. The nature and number of peripherals indicate variable input/output handling. Volatile databases need security precautions and recovery facilities totally different from those relevant for more static databases. The timing requirements imply that recovery procedures should be automatic, and that algorithms for defining which subset of the data is in main store are necessary. Minimum assumptions have been made about the hardware configuration to be used for an implementation. The strategy may be used on any configuration with fast backing store such as disc, with terminals, and with some form of storage protection. Some computers are provided with an operating system, which together with the hardware forms the effective machine for the user. In some cases, the operating system may be tailored to the requirements of the application by using middleware (Spooner, 1971). In other cases, the operating system might not be usable in this way. For machines supplied without operating systems, the functionally variable system would be designed directly on to the hardware.

The importance of software structure

The importance of structure in programming was first realised when high level languages were introduced to decouple the software from the hardware. For example, several languages had a lexicographic block-like structure which aided the construction of parameterisable procedures and functions.

More recently, it has been realised that the structure of the executed code is also very important and entirely distinct from its lexicographic order. For example, in realtime systems, unlike batch processing, the separate programs are not identifiable as separate units of execution. Instead, there exists an array of functions with many possible sequences of use. Thus, the realtime work of a system can be thought of as being the sum of a

Downloaded from https://academic.oup.com/comjnl/article/18/4/312/347989 by guest on 19 April 2024

number of independent 'routes' where a route is an execution path through the array of processes. Each route would correspond, in user terms, to a job. In the type of dedicated real-time system considered, this would correspond to an operator data entry and reply, which we will refer to as a transaction job.

The trend of modern operating systems is away from the monolithic structure which has been used in the past. This approach has its roots in the Atlas supervisor, as discussed by Spooner (1971). However, the cost of maintaining complex programs has been shown to increase with the disorder of the software (Belady *et al.*, 1972). Therefore, the design strategy of the functionally variable system is to use an ordered structure for the operating system as well as the user programs. This structure has advantages for maintenance and hence reliability, as well as variability, and is based on the pioneering work of Djikstra (1968).

The proposed structure

The software structure proposed divides into two parts. One part is relatively small, and fixed (the nucleus). It defines the construction rules and control paths for the second part. This second part defines the application system.

This split is different from the general operating system/ applications program split, which has been developed (historically) to deal with an 'average' work profile. For instance a job shop with a mix of compilation, testing and running, where all jobs are separate but may use facilities provided by the operating system, is often used to define the profile.

The F-V structure for dedicated systems assume that functions traditionally within the operating system as well as those in the application programs may be varied. The small, fixed part of the code does not contain information about the configuration: I/O is handled within the variable system. Similarly, contention for resources is handled within the application, or variable system. The functions of the fixed part of the system are to enforce the rules and extend the user facilities—for instance, passing control between different parts of the application system. It also, and very importantly, contains the basis of the error detection mechanism.

This basis is a pair of independent clocks which crosscheck each other. On quite general arguments it is easy to see that, if two independent clocks are serviced by execution of two separate software code sections which include a check that the other clock has interrupted since last entry, a failure in either can be detected. This means, unless each is positively reassured of the continued functioning of the other, a restart using a fresh copy of the core must be attempted. However it also means that, while the cross checks are functioning, the timer-based checks which are initiated by the clock routines are dependable. Also the housekeeping functions such as checkpointing, can be relied on to maintain the system integrity.

The application system covers both functions normally thought of as operating system functions, and those thought of as user or application functions. The nucleus also, as mentioned above, validates transfer of control between parts of the application system. To understand the rules it uses, we introduce the terms:

process, to mean a code section with well defined entry and exit points and function, and with associated data areas;

job base, to mean a collection of processes and other resources such as peripherals, which are compiled together and form a machine.

In most configurations, one job base would form 'the application system', with one database, a set of peripherals, and processes to service the messages entered into the system. The processes would include those performing 'system' activities, e.g. I/O, and 'application' activities, e.g. calculations of algorithms. This is referred to as the live job base. There would probably also be a job base which could be used for testing new devices or code modifications. This might use code, or share processes, or read data, which is part of the live job base. Other job bases could be used for compilation or unit tests. These are collectively referred to as test job bases.

Transaction jobs in the functionally variable system are short-lived. Most live jobs may access the application data, but the jobs are required to be as independent as possible to prevent errors spreading through the system. These somewhat conflicting requirements have been reconciled by structuring the job so that the application data may only be updated at the end of a job. It is thus possible to ensure that all the processing is satisfactory before the data is changed, and to limit the time for which a failure in the job could prejudice the integrity of the database.

The applications data area, containing the database, is thus protected against corruption. The methods used for granting access to items of application data ensure that all jobs see a consistent set of data items. Thus the recovery procedure, which combines a job base with the latest check point of the applications data area, leads to a reconstituted system containing only the effects of finished jobs.

The job bases form a hierarchy, the aim of which is to ensure that the live system is not degraded by the operation of the test system. Thus, jobs on the live job base use facilities from the nucleus—the fixed code. Test jobs can use facilities—such as the data entered by operators—of the live job base; the techniques used for this are discussed later, under 'sharing of resources'. The importance of this hierarchy for failure handling is discussed below.

Failure handling and recovery

It is necessary to consider how to:

(*a*) detect errors
(*b*) localise errors
(*c*) recover from errors.

The detection of errors in any collection of programs will to a large extent depend on validity and redundancy checks on data fields. In addition, it is necessary to detect loops inadvertently caused. In systems of co-operating processes in particular, it is necessary to provide a mechanism for detecting errors arising from improper co-ordination of the asynchronous threads. It is for these two latter cases that time-based checks are used.

Given that an error is detected, how can it best be localised? The transaction job described above, corresponding for instance to a message and answer, is the natural unit. By separating code and data throughout, it is possible to be confident that an erroneous job has not corrupted code, or data used by another job, except for two cases. The first is that of the application data area discussed above, the second is data, local to a process, which is nevertheless read by subsequent jobs using that process. Such data might be, for instance, the disc directory.

It is necessary to design the software so that the system can recover—albeit inefficiently—should this data be erroneous.

Recovery of job can in principle be manual, automatic, or not required. While automatic recovery is probably appropriate for jobs started in response to messages conveyed by noisy lines—the message has perhaps been corrupted in transit—it is less so for local operators engaged in complex interactions. In the latter case it is more probable that the operator has made an error, which he would not repeat. Jobs started in response to scans of input channels on a continuous basis, may be appropriately 'lost' in favour of the next scan.

There are, however, some situations in which it is not possible to attribute an error to a transaction job—for instance, if the code to scan the dispatcher queue containing several jobs finds that a housekeeping count differs from the number of jobs

Downloaded from https://academic.oup.com/comjnl/article/18/4/312/347989 by guest on 19 April 2024

actually seen. Then if this code is on a test job base, that job base is failed. If the code is on the live job base, this and any test job base in existence are failed. If the code is in the nucleus and this type of logic error is detected, the nucleus and all dependent job bases are failed.

The implications of a job base being failed are several. The first is that all jobs active on the job base are lost. If the job base is for testing, recovery action is left to the users to initiate. If, however, the live job base is failed, then to maintain reliability it is necessary to automatically reinstate the job base. This is done by reading from backing store a copy of the code of the job base, and combining it with a check point of the applications data area, to represent completed jobs. Then from the job numbers, the operators whose jobs have been lost are notified. If the check point is taken frequently, the number of jobs lost will obviously be lower although the overload will be higher. It is also important to note the importance of using the smallest unit which the operator can recognise, as a job.

The recovery mechanism is similar for errors detected in the nucleus: for reloading the nucleus, a section of code called the fallback code is used. This code section is sumchecked and stored in duplicate, so that occurrences of failure in fallback should be rare.

Varying the function

It may be required to vary the code or the database structure, either for maintenance purposes or to extend the function of the system. In this case the job base is the relevant unit.

The first step is to compose the job base, off-line, from the amended code and data descriptions. This activity consists of linking the compiled code segments and of testing the base for compatibility with the existing bases. The composed job base is labelled for easy user access later, and filed on backing store. For instance, if the main store required is more than is available, this can be established before any attempt to use the job base for live data.

The second step is to run the amended job base as a test job base, which may be used to perform compilations or module tests. It may also be run in parallel with the live job base, using some or all of the same input messages as the live job base.

Thirdly, when the amended job base has proved satisfactory

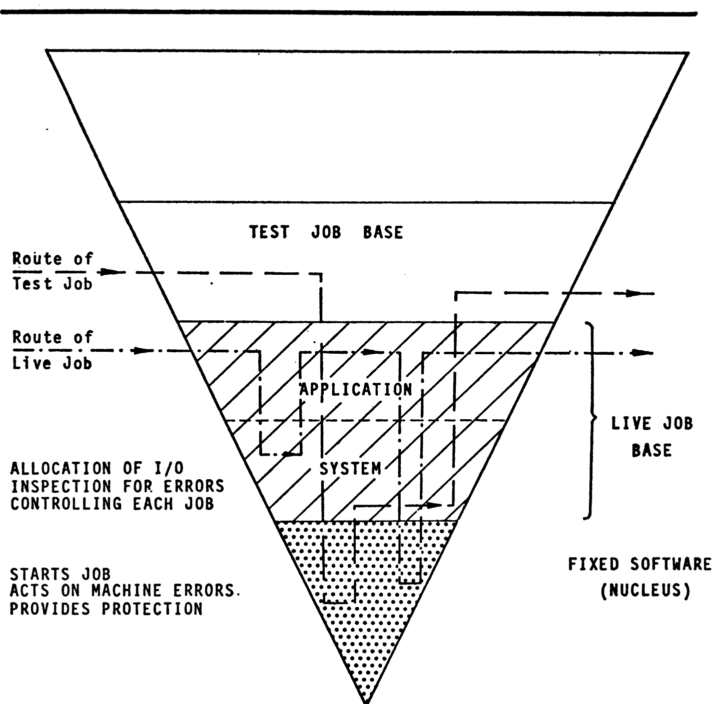

Fig. 2 Structure of the software

under test, it may be introduced as a new live job base. A terminal language is defined which enables the operator to fetch a labelled job base from backing store. This job base may take the place of the existing live job base in servicing data with a delay time of a few seconds. Moreover, there are no user jobs lost. This combination, that of small delay time and no job loss, is possible when the job is of typically short time scale.

This treatment highlights the difference between the structure proposed, for dedicated operating systems, and that employed in conventional operating systems which run independent user jobs. In our structure, the data is discarded if errors occur during processing and the code is kept. (Once the applications data area has been updated, and a checkpoint taken, the data is then 'permanent' in the same way the code is.) In conventional operating systems under similar conditions the code is deleted though the effects of the job—its data—will probably not be deleted.

Sharing resources

The test job bases need to be able to share resources which belong to the live system. For instance, to save main store space it may be required that the test job bases use either pure code, or processes, which are bound in the live job base. The sharing of code (where the queues and data areas of the test system are separate from those of the live job base) provides more security for the live base than the sharing of processes, and reduces the possibility that the live system's performance may be degraded through a malfunctioning test job base.

This method cannot be used for some types of device handlers. If the device is local to the job base, it is appropriate that the process to service it, is local to the job base. If it is desired to test a job base by routing input data to it from the live job base, this may be done by a ritual of requests from the test job base, avoiding the mixing of data from the two job bases on the same queue. The use of shared output is the case in which shared processes, rather than shared code, must be used. This is because co-ordination of requests from the bases sharing the output device must be performed by the process itself—to ensure, for instance, that a complete message is transmitted before another is accepted.

Another resource which is shared is the dispatcher or scheduler. The requirement that the test systems should not be able to degrade the live system leads to the adoption of a priority system, so that outstanding tasks for the live system lead to the adoption of a priority system, so that outstanding tasks for the live system are always scheduled first. The requirement also implies that a time slice system be used, so that the test system may be interrupted to deal with activities for the live system.

Implications of the approach

Hardware resources

The minimum configuration for a functionally variable system includes enough main store and processor power to allow for testing as well as for running the application. The nature of the technique used for varying the system—that of job base replacement—demands a fast form of backing store with access to files. Multiprocessors with access to a common area of main store are advantageous for reliability and efficient use of redundant hardware. The software could, however, be implemented on a single processor configuration if the lack of reliability implied were acceptable. We have not investigated cold start methods from standby machines, which would probably be needed if a single processor were used.

The size of the software will increase as functions are added and changes are made to the software. By reorganising the software, this growth in size can be kept in check. However, once this is done, fallback is no longer possible. There is therefore a straight tradeoff between recoverable changes and in

Downloaded from https://academic.oup.com/comjnl/article/18/4/312/347989 by guest on 19 April 2024

creating size. **Fig. 3** shows the effect on software size of changing requirements and reorganisation.

Software

The techniques used to achieve reliability have been chosen to minimise the overheads. The method of timer assurance, for instance, is designed to detect malfunction in an efficient way and enable corrective action to be taken before the software has deteriorated. Failure localisation and error immunisation rely on a formalisation of methods of passing control from one part of the software structure to another. These methods have an execution time overhead, but are unavoidable in a structure designed for reliability. The separation of current data and composed job base has been defined to speed the check-pointing process. The time spent in check-pointing will depend on the volume of the application data and the device used to store it: a drum being an order of magnitude faster than a disc, and the transfer time being the major single item.

Software variability has been achieved through the job base structure. This is an efficient structure for any dedicated application with a common database. Access rate to the database is a crucial factor in determining the response time of a system. The development of algorithms, for accessing the active subset of a database which is held in core, appears to have been neglected. Cheap mass storage devices may well, however, revolutionise this area.

The software could be written in any high level language which can be extended, by macros or procedures. The standard features of a compiler, assembler and linkage editor are necessary. The job base composer must additionally provide the labels used for fast access while replacing a job base during changeover or fall-back.

Streamlining of failure handling

The techniques for failure handling are dedicated to isolating errors to within one job, or 'event', for example by accessing common data only through a special handler.

The message passing mechanism is structured to prevent errors propagating from a faulty code section to an illegal destination. Since it is assumed that the job base as a whole is unaffected by errors localised to within a job, failed jobs may be repeated. This may be done manually by operators, automatically if from remote stations, or by abandoning the data received from continuous scan devices. The decision to repeat the job, or to reload the system if many user jobs were failing could thus be taken by the operator.

Downloaded from https://academic.oup.com/comjnl/article/18/4/312/347989 by guest on 19 April 2024

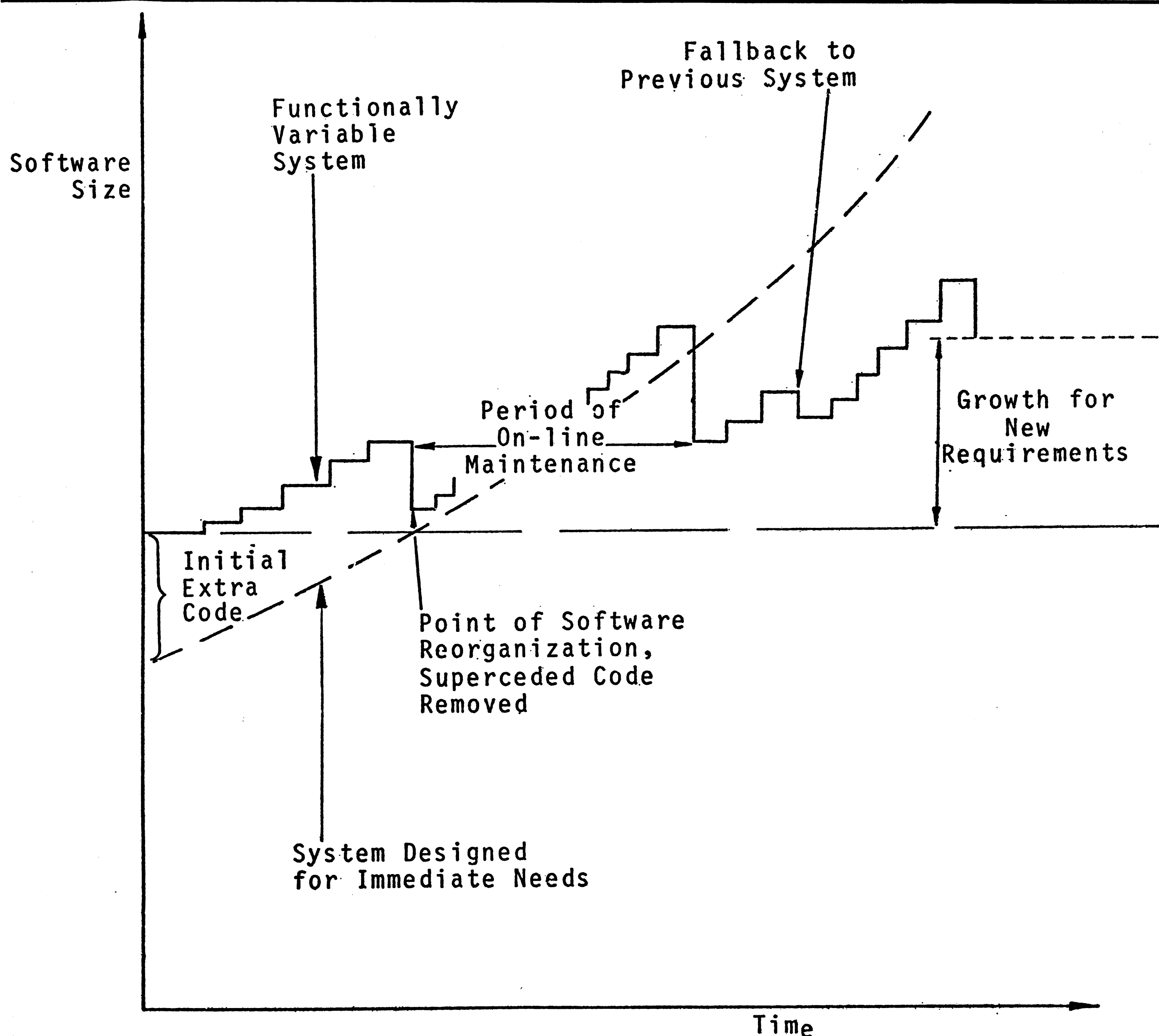

Fig. 3 Development by phases

There are two conditions which make it necessary to abort the live job base. Firstly, errors which cannot be attributed to a user job on the live job base must cause the job base to fail, since it is possible that the errors are due to code corruption. Secondly, the live job base must also be aborted if a job fails after it has gained access to the application data, because the data may have been corrupted by the job. In both cases, recovery is initiated.

The automatic recovery of the live job base after system loss corresponds to a warm start. Jobs current during system loss are cancelled. The delay time is of the order of seconds before new jobs are again accepted, operators being notified of jobs which have not been completed due to systems loss. Recovery may use this job base issue or possibly a previous issue.

Phased development

There are two ways in which the software structure may be varied. The technique is the same in both cases, namely, the new job base is introduced from backing store, but the implications are different. One method implies secure reversion to an old and trusted job base, the other precludes this. The methods together provide for phased development.

The conditions under which reversion is possible may be summarised as: the new job base must contain the code and data of the old, and must continue to update all data items which were updated by the old job base. It is anticipated that job base modifications of this type would be introduced to the system as a regular maintenance tool. The software structure grows by accretion when the function is varied in this manner.

It has been mentioned earlier that a general theory of functional variability had been rejected. The problem of growth by accretion provides an interesting illustration of the scope of the general theory. It is possible to show that in general nothing can ever be deleted from the software. Consider for instance a job base issue n, the last to require for application purposes a particular code segment. If we delete the code segment after j more issues, the applications data items which were updated by the segment (if any) will no longer have the same relation as on job base n to the real world. Then, if job base $(n + j)$, the latest, fails and a cascade of errors forces the system back to using job base n, the application data and code will be inconsistent. In particular cases, of course, it is possible to show that successful reversion will not be affected by deletion of given items. For instance, when all the devices of a particular type have been removed, the code to handle them is redundant. Thus, in this case the conclusions of the general theory need modifying for a particular situation.

At intervals, it will become convenient to make a discontinuous structural change to the software. This is essential if the configuration becomes no longer valid—for instance, the terminals are regrouped—or if the database is reorganised. (The latter case could, for instance, correspond to the addition of an airport to a network of air lanes, to adding a new branch to an online enquiry system for a group of shops, or to a new one-way system in a road control scheme.) It is anticipated that changes of this type will be less frequent than changes for which reversion is possible.

Conclusion

The problem may be stated as that of maintaining reliability of realtime control systems. The aim is to avoid breaks in service because of errors in the software, to enable the errors to be corrected while sustaining service, and to enable the software to be enhanced while the system is live.

The need to design an operating system, concentrating on factors for extreme reliability, has been apparent for some time. A pilot implementation is being performed to establish that the techniques defined combine to form a reliable system (Ringland and Trice, 1975), and further details of the strategy have been published as a CAP report (1972).

The approach is based on the adoption of a structure. Failures are localised to the smallest unit possible to prevent disruption of the rest of the structure. The structure used to recover the system after fatal errors is used to facilitate the variation of the function of the system.

Some of the structural elements in the software could be applied in the context of existing software suites. These are the job structure, the techniques for separating code and data, and those to increase the security of the application data. It is anticipated that the effectiveness of this type of analysis would be demonstrated by the increased reliability of modules implemented in this manner.

In software terms, the key to the response time of realtime systems is the handling of the large volume of volatile data describing the application. The data must be secure but accessible from many jobs. It is obvious that a directory system is necessary, to achieve the flexibility demanded by the application. But further analysis is needed of the problems associated with the application data in the context of the theory and practice of in-core databases.

The second paper in this series (Ringland and Trice, 1975) will describe the use of a Modular 1 to implement these ideas, and analyse the implications of using this structure to design a demonstration suite with Air Traffic Control-like properties.

Acknowledgements

The work described was commissioned by MOD(PE), and the the paper is published with the permission of MOD(PE). The author wishes to acknowledge helpful discussions with members of the staff of MOD(PE).

The author is writing as representative of the CAP team which included E. Hart, R. Jones, A Gough and R. Hill. The ideas expressed were evolved jointly: faults in exposition are the author's.

This paper summarises the CAP report 1972, published as MOD(PE) Report JF/A/0183.

Downloaded from https://academic.oup.com/comjnl/article/18/4/312/347989 by guest on 19 April 2024

References

ACM (1969). *Proceedings of the 2nd Symposium on Operating Systems Principles.*
BELADY, L., *et al.* (1972). *System growth dynamics*, Imperial College preprint.
CAP (1972). Software functional variability: A methodology for reliability in *Computer Systems.*
CLARK, K. (1970). *Middleware*, CAP Report.
d'AGAPEYEFF, A., and CLARK, K. (1968). *Functional Variability in large on-line systems*, CAP Report.
DENNING, P. J. (1971). Third Generation Computer Systems, *Computing Surveys*, Vol. 3, pp. 176-216.
DJIKSTRA, E. W. (1968). Structure of the Multiprogramming System, *CACM*, Vol. 11, pp. 341-346.
ELSPAS, B., *et al.* (1972). An assessment of techniques for proving program correctness, *ACM Computing Surveys*, Vol. 4, pp. 97-147.
FERGUS, P. J. B., and TAYLOR, J. M. (1971). High Integrity Systems, UKAEA. *Report HL71/5799.*
RINGLAND, G. M., and TRICE, A. R. (to be published). A pilot implementation of software functional variability.
SPOONER, C. R. (1971). A Software Architecture for the 70's, Part 1, *Software Practice and Experience*, Vol. 1, pp. 5-38.
WILKES, M. V. (1968). *Time Sharing Computer Systems*, McDonald.