A DEVELOPMENT OF HAZARD ANALYSIS TO AID SOFTWARE DESIGN

J. A. McDermid and D. J. Pumfrey,
Dependable Computing Systems Centre,
Department of Computer Science,
University of York,
Heslington,
York YO1 5DD, U.K.

Abstract *— This paper describes a technique for software safety analysis which has been developed with the specific aim of feeding into and guiding design development. The method draws on techniques from the chemical industries' Hazard and Operability (HAZOP) analysis, combining this with work on software failure classification to provide a structured approach to identifying the hazardous failure modes of new software.*

I. INTRODUCTION

Software safety analysis is a focus of much current research, and many methods have been proposed. However, most of these methods suffer from two problems; they are difficult to apply at all stages of software development, and tend to produce large, intractable sets of results from which it is difficult to extract useful design guidance.

As part of a study to assess the capabilities of various software safety analysis methods, we proposed the following set of desirable properties:

1. The method should be capable of being applied at all stages of the system lifecycle from initial design through to validated implementation.
2. The method should not involve an excessive increase in the work required at early stages of the design. Ideally, it should allow the system designers to identify quickly which areas of the design are most critical, and concentrate further analysis work on those areas.
3. The analysis should help drive design development through the comparison of alternatives and the refinement of specifications.
4. It should be possible to have a high degree of confidence that thorough application of the method will lead to consideration of all credible failure modes.
5. The analysis should be in a form which allows the design to be checked and approved incrementally, permitting closer integration of design / implementation and verification / validation activities.
6. The results of the analysis should be in a form which is suitable for inclusion in a safety case.

To assess applicability to all stages of the design and implementation lifecycle, we considered whether a method contained features to support both *inductive* and *deductive* safety assessment techniques — typified by Failure Modes and Effects Analysis (FMEA) and Fault Tree Analysis respectively — and also two additional classes of analysis identified by [1]. These additional classes are *descriptive* and *exploratory* analysis, which describe the cases where the causes and effects of failures are both known (descriptive), and both unknown (exploratory).

At the earliest stages of design of new software, nothing is known about its failure modes, and knowledge of the effects of failure will generally be limited to a high level preliminary hazard analysis. However, this is the stage at which it is easiest and most cost effective to take measures to improve the safety characteristics of a system, so exploratory analysis is particularly important.

We concluded that no software safety analysis technique proposed so far adequately addressed this requirement for a *structured* exploratory analysis of the safety properties of a completely new software system. This paper presents a method which we are developing to support such an analysis. Section II briefly presents the background and important concepts of our method, which is then described in sections III and IV, and illustrated by a small example in section V. Conclusions and an outline of our proposed directions for further developments of this work are presented in section VI.

II. SELECTION OF METHODS

The development of a completely new analysis technique, with the attendant problems of convincing potential users of its value and overcoming resistance to completely unfamiliar procedures and notations, was considered undesirable. Instead, we undertook a survey of a range of safety analysis techniques from various industries, and considered their suitability for adaptation to software development.

We concluded that Hazard and Operability Studies (HAZOP) [2], [3], widely used in the chemical, nuclear and food processing industries, has features which made it an interesting starting point for further work.

The HAZOP system of *imaginative anticipation* of hazards was developed to provide, for chemical plants, precisely the type of structured analysis feeding in to the development of a new design which we considered lacking in the software domain. A HAZOP study attempts to identify previously unconsidered failure modes by suggesting hypothetical faults for review and, where necessary, this is followed by the suggestion of means of overcoming identified hazards.

HAZOP has a number of features which distinguish it from other analysis methods such as FMEA, and our work has

concentrated on two of these features which convey particular benefits. These are the concept of *flow-based analysis*, and the use of *guide words*.

Flow-based analysis means that, in the context of a chemical plant, the first focus of analysis is the properties and behaviour of the flows in the pipelines connecting the major components (storage tanks, reactor vessels, pumps etc.) of the plant. We concluded that this concept could usefully and effectively be applied to software, if a suitable design model was used, by considering information flows between components. This approach has several potential benefits over analyses based on consideration of the function of each component:

- In many design methods, the interfaces between parts of the system are defined before the component implementation is finalised. A suitable analysis of these interfaces could provide a useful input to the later stages of design elaboration.
- The intended behaviour of an interface is likely to be simpler both to specify and understand than that of an active component.
- In general, the failure modes conceivable for interfaces are more restricted than those of active components. This may help to contain complexity and limit the size of the results.

It is important to note that the term *flow* as used in this paper is not specifically *data* or *control* flow, since both must be considered as part of the analysis.

For each flow, a set of *guide words* is used to prompt consideration of hypothetical failures, known as *deviations*, from the intended characteristics and behaviour. If a hypothetical failure suggested by the guide words can be shown to have both conceivable cause(s) and hazardous consequences, it is a *meaningful* failure mode, and consideration must be given to measures which can be taken to remove its causes or limit its effects. These guide words provide the structure of the analysis and, if suitable guide words are selected and correctly applied, can give confidence in the coverage achieved.

A critical feature of the process industries' HAZOP is that it is a team activity, and great emphasis is placed on the selection of appropriate team members. Their diversity of knowledge and experience helps to ensure a thorough investigation of all properties of the system, and the interdisciplinary approach helps to prevent one person or group solving a problem in a way which will create new problems in other areas. We believe that a similar team approach is appropriate for software analysis and development, but the composition of the team will depend on the organisation performing the study, and this paper does not address this issue.

The decision to use HAZOP as the basis for our work favours the use of design notations which employ a structural model of the system, i.e. one which partitions the system into independent processes or modules and defines the interfaces (however implemented) between them. Whilst we accept that a structural model alone is not sufficient to fully specify a system, and must be supported by other models (e.g. a state-based model), it is appropriate for the first stages of design of most systems, and can be applied from a very high (context) level down to a relatively detailed level. Thus, even if the method we develop is not, of itself, sufficient for a total safety analysis of a system, it should at least be suitable for identifying those parts of the system where other more detailed safety analysis techniques must be used.

Although we believe the techniques we are developing could readily be applied to a wide range of methods and notations, it was necessary to select one notation as the basis for our initial effort at designing a "software HAZOP". We selected MASCOT 3 [4], since MASCOT's communication *paths* correspond closely to our concept of information flows.

The principal components of a MASCOT design decomposition are *activities* — fundamental processing elements, conceptually executed in parallel — and *Intercommunication Data Areas* (*IDAs*) — passive components which encapsulate the mechanisms through which the activities communicate and share data. The fault transformations possible within a passive IDA are much more restricted than those possible within an activity, providing good fault containment properties in the implementation, and an effective basis for analysis.

A further attraction of MASCOT is the strong mapping between a MASCOT design and the eventual implementation. The structure of the code is developed directly from a textual representation of the design diagrams, supplemented by definitions of data types and the executable code. Thus we can have confidence that analysis results at the architectural design level will remain valid for the implementation.

III. Method outline

We first briefly describe other approaches to HAZOP based software safety analysis in order to identify limitations which we intend our method to overcome.

A. Other approaches

A number of recent papers [5], [6], [7] have suggested adaptations of HAZOP to the software environment.

Burns and Pitblado [5] propose three separate studies for programmable systems which control or monitor plant or machinery:

1. An initial "conventional" HAZOP studying the plant to be controlled, using the guide words and method outlined above.
2. A more detailed Programmable Electronic System (P.E.S.) HAZOP study of the computer or Programmable Logic Controller (PLC) systems controlling a plant, considering deviations in *SIGNALS* and *ACTIONS*, using the guide words *NO*, *MORE*, *LESS* and *WRONG*.
3. A human factors HAZOP.

Of these, the P.E.S. HAZOP is closest to our intended application, but the paper implies that this study is intended to be

conducted at the level of the external interfaces of the system.

J. V. Earthy's short paper [7] presents little more than an overview of some possibilities for adapting HAZOP techniques to software. Again, the recommendation appears to be to apply the analysis at the level of interfaces, in this case between processor, storage devices and peripherals. At a lower level, data flow diagrams are identified as a suitable model for analysis, subject to verification that they represent the system as built, but this overview does not suggest details of a method or propose guide words.

Cambridge Consultants' modification of HAZOP described in Chudleigh's paper [6] most closely matches the method we are developing. Data flow diagrams are used as a basis for the analysis, a table of guide words and the parameters to which they apply is presented, and a brief description is given of the manner in which they are applied. However, the method does not follow the principles of the process industries' HAZOP method in that, although deviations in the input data flows are considered, the processes (components) themselves are analysed to determine possible deviations of the process outputs. Mention is also made of the review of the analysis of data flows entering and leaving the diagram from and to a higher level of the hierarchy.

The analysis of activities as well as data flows seems to complicate the analysis unnecessarily; any meaningful failure of a process must eventually manifest itself as a deviation at an output of that process and, provided the analysis of the output flows is thorough, will be considered when possible causes are sought for that deviation. A consequence of this complication is that, although the set of guide words presented is relatively large, it is difficult to be confident that they provide complete coverage of all credible failure modes. In developing our own method and selecting the guide words we have attempted to address these criticisms.

B. Our method

It is important to state that the intention of this analysis method is only to assess the safety or otherwise of the *application* software. It assumes that the operating environment (i.e. the MASCOT run-time system) is error free, and correctly enforces the fundamental MASCOT principles such as independence of processes, and inter-process communication via IDAs.

Although we recognise the potential benefits to software system development of a team approach to analysis and design review, it is hard to define the environment in which the method is applied, and our work so far has concentrated on defining the procedure and guide words.

The basic unit for analysis of a software design is a single MASCOT drawing representing a **system** or **subsystem**. This system or subsystem consists of **components** — i.e. MASCOT *activities* and *IDAs* and *external devices* — connected by **information flows**. These flows may be MASCOT *paths*, or *device-server interfaces*. Separate tables are produced for each MASCOT diagram, with the MASCOT hierarchy defining the relationship between the tables. The process begins with the top-level (context) MASCOT diagram of the system.

The major steps of the method are:

1. The **flows** in the diagram are identified and consistently **labeled**.
2. The design is reviewed to ensure that the *intended* operation is clear. At this stage, various **context information** is recorded. This is mainly a textual form of the information contained in the MASCOT notation, and could be supplied automatically by a suitable tool. However, *path protocols*, which describe the communication and synchronisation models of each flow, are not shown in MASCOT, and must be added.
3. A table of **guide words** is constructed, as described in section IV
4. The appropriate set of guide words is considered for each flow. Each guide word may suggest one or more **hypothetical failure modes**, which are recorded.
5. The **potential causes** of each identified fault are determined.
   This stage is a *deductive* analysis (i.e. similar to Fault Tree analysis), searching for possible causes of the hypothetical failure in the component where the flow originates.
6. The **effects** of each hypothetical fault are considered and recorded. Where necessary, the effects of the hypothetical fault in combination with **normal states**, **normal events** or other faults occurring simultaneously are also considered.
   This step is an *inductive* analysis (i.e. similar to FMEA) of the effects of the hypothetical fault on the destination component of the flow.
7. The set of hypothetical faults is reduced to a set of *meaningful* faults by discarding those for which the potential causes are acceptably improbable, and those for which no hazardous effects have been identified.
   An important feature of the method is that a *justification* must be given whenever a hypothetical failure mode is discarded. In most cases this will be a simple statement, but where the decision is difficult it may be necessary to supply a more complete argument.
8. For each meaningful failure mode identified, alternative strategies are suggested for removing its causes or limiting its effects. These may take the form of design modifications or a set of requirements which must be satisfied by lower-level design elaboration to achieve acceptable system-level safety properties.
   The final step is selection of one of these strategies to pursue, and recording a justification for the selection.

When the design of the current level of decomposition is satisfactory, the first-cut design for the next level of decomposition is produced, taking account of any new requirements derived from the analysis, and the process begins again at the new level.

C. Application of the method

One of the most obvious over-simplifications in this outline method is that it assumes that system development will always proceed top-down, and that the role of the analysis is simply to refine the specification of lower level subsystems. Once defined these specifications will only be changed if requirements change at a higher level, or it proves impossible to meet the specification, prompting a re-design. This is quite clearly at odds both with the reality of system development and with our stated aim of producing a method appropriate to all stages of a more integrated system lifecycle.

The top-down development model is attractive because of its simplicity; it consists of a sequence of steps which are simply repeated at successively more detailed levels until there is nothing to be gained from further decomposition. It is well suited to an environment where many people are involved in the development of a system, particularly if parts of the system are to be subcontracted to other departments or companies.

In practice, however, rigid adherence to this model is too inflexible — there are many factors which can lead to a system being developed in quite different ways. For example, a new system may be re-using parts of an existing system, for which no safety analysis has been carried out. Alternatively, the system may be based on a library of standard low-level routines, e.g. hardware interfaces or common functions, which have well-known properties. Another common approach is for parts of the system which are seen as difficult in some way to be either prototyped or fully implemented first, and the rest of the system built around these core parts. The integrated method must be sufficiently flexible to accommodate all of these scenarios.

To provide acceptable support for these different patterns of development, we have defined a second role for the method, namely as a means of recording design assumptions affecting safety. This is based on the observation that at all stages of design, up to and even including actual coding, the implementor(s) of a system are making implicit assumptions about the way that other parts of the system will function. The only change required to the outline given above is that, instead of specifying what is required of subsystems at a given level, the analyst records the anticipated failure properties of each subsystem, based on an assumed implementation, and then uses these to determine whether the current level of decomposition will meet the properties which were assumed when it was defined at a higher level.

This approach attempts to take advantage of the way in which system implementors work. It is far less rigid than the system of progressive specification refinement and, as such, is probably better suited to the lower levels of system design and implementation, or to small systems which will be developed by one person or a small team. Its biggest disadvantage is that it is extremely difficult to concisely express what may be a very complex set of assumptions about the expected implementation of a system or component. One of our aims in our research is to find an appropriate balance between complexity and utility in representing assumptions.

These two models of development are not incompatible. Indeed, even if a progressive refinement approach is adopted rigorously, the analysts should, in specifying the requirements for a subsystem, consider possible implementations of that subsystem (i.e. make assumptions) and set requirements which are believed to be reasonable and attainable.

IV. Selection of guide words

Since the set of guide words used for HAZOP analysis in the process industries has been developed and refined over a considerable period of time, the interpretation of each guide word in a given situation is well understood, and there is a high degree of confidence that systematic application of the complete set of guide words, by a suitably qualified team of people, will result in a complete analysis of all the important failure modes of the plant.

Traditional analysis techniques have concentrated on identifying rather than classifying failures. However, a considerable amount of research has been carried out into the classification of software failures, and this provides the basis for proposing a means of developing sets of guide words with a high degree of confidence in their completeness.

Recent publications in this area include [8] and [9]. The categorisations proposed by both papers are similar, although that proposed by Bondavalli and Simoncini [9] is of more interest to this work, since it explicitly examines the detectability of faults, an important property when considering strategies for handling failures.

The categorisations they propose are based on consideration of a service — usefully analogous to our model of an information flow. The provision of a service is specified in terms of two parameters; the *value* associated with it, and the *time* at which this value is presented. The value domain is divided into four categories; *correct*, *subtle incorrect*, *coarse incorrect* and *omission.* The distinction between subtle and coarse incorrectness is that subtly incorrect values cannot be detected. The time domain, similarly, is divided into four; *correctly timed*, *early, late* and *infinitely late.* Bondavalli and Simoncini's summary of the possible combinations of time and value faults, and how they may be detected, is reproduced in table I.

The distinction between an omission in the value domain and infinite lateness is assumed to be made by a *perfect observer*, who has knowledge of the internal state of the system providing the service. In practice, as the table shows, the two are indistinguishable to a user of the service, and can only be detected by their timing behaviour.

A problem with this scheme is that there is no combination of time and value categories which reasonably accommodates the case of a faulty system which emits some sort of output when a correctly functioning system would not have emitted any output at all. It is interesting that Bondavalli and Simoncini reject the *Byzantine* fault class proposed by

TABLE I : FAULT CLASSES AND DETECTABILITY

Time	Value			
	Correctly Valued	**Subtle Incorrect**	**Coarse Incorrect**	**Omission**
Correctly timed	Correct service	Undetectable failure	Detection on value syntax or semantics	Detection at T_{inf}
Early	Detection on time	Detection on time	Detection on value syntax or semantics and/or time	Detection at T_{inf}
Late	Detection on time	Detection on time	Detection on value syntax or semantics and/or time	Detection at T_{inf}
Infinitely late	Detection at T_{inf}	Detection at T_{inf}	Detection at T_{inf}	Detection at T_{inf}

Shrivastava and Ezhilchelvan, [8] which expressly includes events such as completely unexpected output. It is desirable for analysis purposes to include this case specifically — we will use the term *commission*.

We therefore consider that a complete set of suitable failure classes is:

Service provision : OMISSION
COMMISSION
Service timing : EARLY
LATE
Service value : COARSE INCORRECT
SUBTLE INCORRECT

These words represent a similar level of abstraction to the guide words used for process HAZOP analysis, and could be employed directly. We believe, however, that there are be significant advantages in attempting to develop a more precise set of guide words. In particular, this might make it easier to produce a formal definition of each word, which will ultimately be useful in automating analysis.

The most obvious way to attempt the definition of more precise guide words is to consider the interpretation of each failure class when applied to particular data types. At present, we know of no other work which has attempted to do this. The meaning of the *value* failure classes can clearly be refined by considering the data type to which they are applied. However, knowledge of the data type of a flow alone is insufficient, since there are many different path protocols (i.e. communication models) possible in MASCOT and it is these which determine the timing and service provision characteristics of a flow.

Our proposal is that, once the data type and path protocol of each flow in a MASCOT diagram have been established, guide words should be defined by considering the interpretation of each failure class in the context of every combination of type and protocol. Clearly, where the same types and protocols are used in many diagrams, this need only be done once. For some combinations of data type and path protocol, consideration of one fault class may result in the definition of more than one guide word. The guide words thus defined are recorded in a table, and the appropriate set selected as each flow is analysed.

V. EXAMPLE

To illustrate the application of the method, consider the following example. A prototype full-authority electronic throttle controller is to be added to the engine management system (E.M.S.) of a development vehicle. The mechanical linkage between the accelerator pedal and the throttle plate will be replaced by pedal position sensors and a computer controlled actuator. The requirements for the new system are:

- The accelerator pedal will have two independent position sensors. In addition, there will be two independent switches which will be *opened* when the accelerator pedal is released and reaches the top of its travel, so a wiring or switch failure will appear to signal a closed throttle.
- The throttle body will have a single actuator controlled by the E.M.S., which will directly control the position of the throttle plate. If the actuator is not energised, a mechanical spring and damper system will close the throttle plate.
- The throttle control software will receive data from other vehicle systems and E.M.S. subsystems. This data will be assembled by a data monitor process, and supplied to the throttle control software in a single record to ensure that it is consistent.
- The software shall have two independent channels, each of which will receive a copy of the data record from the data monitor process, the signal from one of the accelerator position sensors and one of the end stop switches, and independently calculate the required throttle plate angle.
- Each channel shall incorporate self-test routines to detect sensor failures and internal errors.

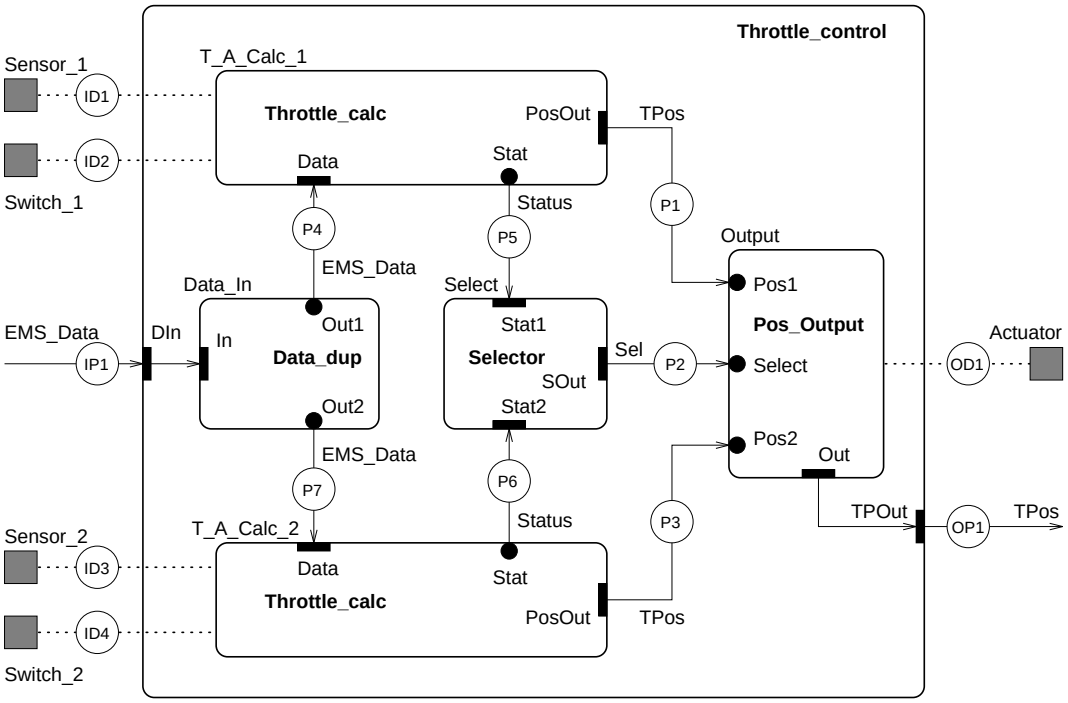

Fig. 1. Initial proposal for the Throttle Control System decomposition

- The position of the throttle plate shall be updated every 25 ms. It shall normally be controlled by the output of one channel, unless its self-test routines signal an error, in which case the other channel shall be used. If both channels fail, the actuator will be released (de-energised) to allow the mechanical system to close the throttle.
- The software shall return a record to the E.M.S. containing the commanded throttle position, and a flag which shall be set if the controller has failed.

The initial design proposal is to partition the software into five subsystems — the two calculation channels, a process which duplicates the data received from the E.M.S., a process which monitors the calculation channel's status and determines which is to be used, and the output routine containing the actuator interface. Since the actuator update rate is relatively high, all the data flows into the output routine are to be *pools* — a destructive read / non-destructive write protocol which can be implemented to give completely asynchronous access. Figure 1 represents this design.

This design is subjected to analysis. The flow labels are shown in small circles on the flows in the diagram, and table II shows the guide words derived by applying the failure classifications to the combinations of path protocol and data type used in this design. Some comments should be made on this table:

1. The categorisation *early* is shown as not applicable to the *pool* protocol, since there is no way in which this can be detected by or affect the activity which is reading from the pool.
2. The *complex* data type represents a data structure which contains several (possibly related) values, but is always passed as a single item.

Table III shows a fragment of the analysis output — the column headed **M?** records whether a hypothetical failure mode has been identified as *meaningful*. This section has been selected because it shows a critical failure mode; consideration of the guide word OLD DATA applied to flow P2 reveals that there is a potential race condition, which could result in the output of a defective channel being used to control the throttle. This arises from the separation of the status and result outputs of each channel. If a scheduling failure should occur, it is possible that the error status output by a channel which has detected an internal failure may not propagate through the *Select* subsystem in time to prevent an erroneous result being read by the *Output* process. Since the *Select* subsystem cannot be guaranteed to complete before *Output* is scheduled, this failure mode is plausible, and a redesign is required to remove it.

The solution in this case is trivial — the functionality of the *Select* subsystem is sufficiently small that it can be incorporated into the *Output* subsystem. The calculated throttle position and status output from each channel are combined into a single record, and a sequence counter is added, ensuring that *Output* can also detect a channel which has stopped updating its output. This revised design is shown in figure 2, and part of the analysis of the new combined data path is shown in table IV.

TABLE II : TABLE OF GUIDE WORDS FOR THE THROTTLE CONTROLLER EXAMPLE

Flow		Failure Categorisation					
		Service Provision		Timing		Value	
Protocol	**Data Type**	**Omission**	**Commission**	**Early**	**Late**	**Subtle**	**Coarse**
Device Input	Boolean	NO READ	UNWANTED READ	EARLY	LATE	STUCK AT 0 STUCK AT 1	N/A
	Value	NO READ	UNWANTED READ	EARLY	LATE	INCORRECT IN RANGE	OUT OF RANGE
Device Output	Value	NO WRITE	UNWANTED WRITE	EARLY	LATE	INCORRECT	N/A
Pool	Enumerated	NO UPDATE	UNWANTED UPDATE	N/A	OLD DATA	INCORRECT	N/A
	Value	NO UPDATE	UNWANTED UPDATE	N/A	OLD DATA	INCORRECT IN RANGE	OUT OF RANGE
	Complex	NO UPDATE	UNWANTED UPDATE	N/A	OLD DATA	INCORRECT	INCONSISTENT
Signal	Boolean	NO DATA	EXTRA DATA	EARLY	LATE	STUCK AT 0 STUCK AT 1	N/A
	Complex	NO DATA	EXTRA DATA	EARLY	LATE	INCORRECT	INCONSISTENT

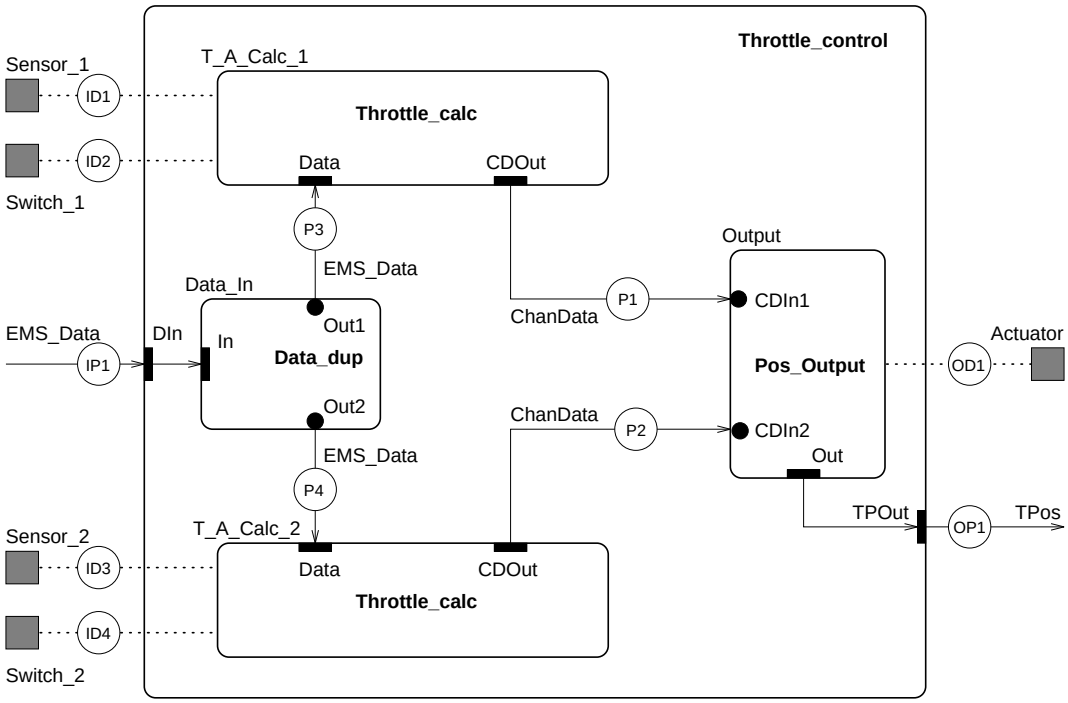

Fig. 2. Revised proposal for the Throttle Control System decomposition

TABLE III : A FRAGMENT OF THE ANALYSIS OF THE INITIAL THROTTLE CONTROLLER DESIGN

Flow I.D. number : P2	**MASCOT type** : Sel	**Path protocol** : Pool	**Basic type** : Enumerated			
Guide Word	**Deviation**	**Causes**	**Co-effectors**	**Effects**	**M?**	**Justification / Design Proposals**
NO UPDATE	Channel status never available to Output	Complete scheduling or communication failure	Any	No output to throttle plate. Failure flagged to EMS	NO	Vehicle will not start, since inhibited by failure flag.
UNWANTED UPDATE	Channel status updated unexpectedly	Scheduling failure	Any	None significant	NO	No significant effects
OLD DATA	Channel status not updated by Select before read by Output	Scheduling affected by any process exceeding time allocation	Sensor or channel failure	Defective channel may have control of throttle	YES	Critical race condition inherent in design, since data and status are passed separately from channels to output. Recommend redesign.
⋮	⋮	⋮	⋮	⋮	⋮	⋮

TABLE IV : A FRAGMENT OF THE ANALYSIS OF THE REVISED THROTTLE CONTROLLER DESIGN

Flow I.D. number : P1	**MASCOT type** : ChanData	**Path protocol** : Pool	**Basic type** : Complex			
Guide Word	**Deviation**	**Causes**	**Co-effectors**	**Effects**	**M?**	**Justification / Design Proposals**
NO UPDATE	Channel data never available to Output	Complete scheduling or communication failure	T_A_Calc_2 failed	Both channels failed, so throttle actuator released	NO	Specified action on complete software failure achieved.
			Any other	Control assumed by T_A_Calc_2	NO	No hazardous consequences.
UNWANTED UPDATE	Channel data updated unexpectedly	Scheduling failure	Any	None significant	NO	No significant effects
OLD DATA	Old channel data read by Output	Disrupted scheduling	Any	Old data detected on sequence number. Effects as for NO UPDATE	NO	No hazardous effects.
⋮	⋮	⋮	⋮	⋮	⋮	⋮

VI. Conclusions

This paper has described the essential principles of a software safety analysis method based on the application of a set of guide words to suggest hypothetical failures in the flows between components of a system.

The method has been applied experimentally to a real system of moderate size, both by its developers and by an independent assessor. The results of this study were encouraging, demonstrating the applicability of the method at early stages of system development and its ability to provide useful input to later stages of design elaboration. The case-study itself was of an aerospace system, but the example above is technically extremely similar and representative of the type of results produced.

The work involved in analysing the case-study was not excessive, and with suitable tool support would be further reduced. We are now investigating the development of a prototype tool to support a larger trial application in an industrial environment. The information such a tool must manage has been identified, and we are currently attempting to define a minimal essential functionality.

As a long term goal, we hope that tools can be developed which provide advanced support such as consistency checking both within a single diagram and between the analyses at different levels of the hierarchy, although this will require the development of a formal notation or structured language representation of failure modes, causes and effects and the precise meaning of guide words.

The method as described above applies to a smaller portion of the systems lifecycle than our ideal, and extension into additional phases must be considered. In particular, we have not yet included verification and validation activities in a study, although we believe that this style of analysis may allow these activities to commence at an earlier stage of development than is commonly achieved.

Although MASCOT has been used for most of the work to date, our hope is that the method can be adapted with relatively little effort to work with other design notations such as HOOD. In common with most other safety analysis techniques, our method is based primarily on the consideration of *events*, and we would like to investigate the potential for developing a similar technique for a *state* based design approach such as StateCharts.

Acknowledgements

This work was supported by British Aerospace under the activities of the BAe Dependable Computing Systems Centre at the University of York.

References

[1] P. Fenelon, J. A. McDermid, M. Nicholson, and D. J. Pumfrey, "Towards integrated safety analysis and design", *ACM Applied Computing Review*, Aug. 1994, (To appear).

[2] CISHEC, *A Guide to Hazard and Operability Studies*, The Chemical Industry Safety and Health Council of the Chemical Industries Association Ltd., 1977.

[3] T. Kletz, *Hazop and Hazan: Identifying and Assessing Process Industry Hazards*, Institution of Chemical Engineers, third edition, 1992.

[4] JIMCOM, *The Official Handbook of Mascot, Version 3.1*, Joint IECCA and MUF Committee on Mascot, June 1987.

[5] D. J. Burns and R. M. Pitblado, "A modified HAZOP methodology for safety critical system assessment", in *Directions in Safety-critical Systems: Proceedings of the Safety-critical Systems Symposium, Bristol 1993*, F. Redmill and T. Anderson, Eds. Feb. 1993, pp. 232–245, Springer-Verlag.

[6] M. Chudleigh, "Hazard analysis using HAZOP: A case study", in *Safecomp '93: Proceedings of the 12th International Conference on Computer Safety, Reliability and Security, Poznań-Kiekrz, Poland*, J. Górski, Ed., Oct. 1993, pp. 99–108.

[7] J. V. Earthy, "Hazard and operability studies as an approach to software safety assessment", in *I.E.E. Computing and Control Division Colloquium on Hazard Analysis*. Nov. 1992, Institution of Electrical Engineers, Digest No.: 1992/198.

[8] P. D. Ezhilchelvan and S. K. Shrivastava, "A classification of faults in systems", University of Newcastle upon Tyne, 1989.

[9] A. Bondavalli and L. Simoncini, "Failure classification with respect to detection", in *First Year Report, Task B: Specification and Design for Dependability, Volume 2*. ESPRIT BRA Project 3092: Predictably Dependable Computing Systems, May 1990.