

Building Embedded Fault-Tolerant Systems for Critical Applications: An Experimental Study

Paul Townend, Jie Xu and Malcolm Munro
Dept. of Computer Science, University of Durham, DH1 3LE, UK

Abstract: An increasing range of industries have a growing dependence on embedded software systems, many of which are safety-critical, real-time applications that require extremely high dependability. Two fundamental approaches – fault avoidance and fault tolerance – have been proposed to increase the overall dependability of such systems. However, the increased cost of using the fault tolerance approach may mean that this increase in dependability is not worth the extra expense involved. We describe an experiment undertaken in order to establish whether or not software redundancy (or the multi-version design method) can offer increased dependability over the traditional single-version development approach when given the same level of resources. The results of this and a subsequent follow-up study are then given. The analytic results from these experiments show that despite the poor quality of individual versions, the multi-version method results in a safer system than the single-version solution. It is evident that regarding the single-version method as a “seem-to-be” safer design decision for critical applications is not generally justifiable.

Key words: Detected and undetected failures, embedded fault-tolerant systems, multi-version programming, safety-critical applications, software development cost

1. INTRODUCTION

According to [SCI00], an embedded system is “*a data processing system which is ‘built-in’ or ‘embedded’ within a machine or a system. It partly or wholly controls the functionality and the operation of this machine. The data processing system and the enclosing system are dependent on each other in such a way that one cannot function without the other.*” In recent years, the number of embedded systems in usage has grown at an enormous rate; this has been especially evident with the rise of handheld computers, network computers and intelligent devices.

With this growth, the role of software dependability within such systems has become increasingly important. Software dependability is usually referred to as *the*

property of a software system such that reliance can justifiably be placed on the service it delivers [LAP95], and it subsumes the usual attributes of reliability, availability, safety, and security etc. These attributes have their own definitions. For example, software reliability is defined as the probability of *failure-free* software operation for a specified period of time in a specified environment, whilst software safety emphasizes the non-occurrence of *catastrophic* consequences on the environment of the software.

Of greatest concern is the dependability of embedded systems within the safety-critical domain, examples of which include factory control systems, heart pacemaker monitors, and radar systems. Systems such as these all have the scope of potentially disastrous failure, including loss of human life, property and the environment. Dependability is also important when considering the economic implications of developing embedded software, as the cost of replacing systems with faulty firmware can be huge, especially where software has been burned directly onto ROM. Many devices, such as those in consumer electronics, simply do not justify the cost of replacement, and so may instead pick up a reputation of poor quality – both in terms of product and company.

Given the clear need for dependability within such systems, an unacceptably high level of faults still occur; indeed, some commentators (e.g. [HAT97]) point out that although techniques that supposedly promote the goal of improved dependability have come and gone, the defect density of software has remained similar for more than fifteen years. It is therefore critical that methods be found for increasing the dependability of embedded software.

[KEL91] states that “...while complete fault removal in sequential software is difficult, it becomes effectively impossible in real-time, concurrent software. Current testing methodologies, while useful to detect and localize errors in sequential software are inadequate when applied to highly concurrent and asynchronous real-time software.” It can therefore be seen that traditional, single-version methods may not be advisable during the implementation and testing of many embedded systems. The *fault-tolerance* approach is based on the view that it is impossible to guarantee that no software design faults exist within a software system, and so seeks to allow errors to be detected and recovered without affecting the running of a system.

There are a number of different ways this can be achieved. Most approaches, such as recovery blocks [RAN95] and multi-version design [AVI77], are based on the use of functionally equivalent software components (i.e. software alternates or versions). **Recovery block systems** work on the principle of *acceptance testing*, whereby a primary alternate is executed, and then an acceptance test is evaluated to provide adjudication on the outcome of this primary alternate. If the acceptance test fails, then the system reverts to its previous state, and the next alternate is executed, and so on; if the final alternate is executed and fails the acceptance test, then the system fails. Unfortunately, there are situations in which backward error recovery is not appropriate; many embedded systems cannot simply “roll back” the system to a previous state. Also, recovery-block systems are less predictable, as it is impossible to know the duration of a given task should an alternate fail and other alternates be executed, and so may not be suitable for many real-time systems. Because of this, the most popular form of fault tolerance within such systems is based on the concurrent execution of multiple diverse components.

Building Embedded Fault-Tolerant Systems for Critical Applications: An Experimental Study

Multi-version design works on the principle of independently implementing n channels of a program, which are then executed in parallel with a single input (although conceptually, parallel execution is not necessary – channels may be executed separately). The outputs of these channels are then compared under a voting system, which forwards a single output based on the majority agreement [KNI86].

In principle, multi-version design provides a general way of allowing a software system to operate successfully in the presence of software design faults. Some researchers have concluded that the dependability of software developed using this approach increases dramatically; for example, [HAT97] concludes that a three-channel system, governed by majority polling, would have a dependability improvement ratio of 45:1 over a single version of the system. This is not a new finding; earlier papers, such as [AVI84] have also argued that multi-version produces highly dependable software. However, such massive increases in dependability have been drawn into question. [KNI90] argue that these gains in dependability are under the assumption that there are no correlated failures within two or more channels of the system – in other words, no faults will occur in the same place and produce the same results. Numerous studies have shown that this is simply not the case. [ECK85] has shown that even small probabilities of correlated faults can reduce the overall dependability of an multi-version system dramatically, and [LEV95] further argues that every experiment with the approach of using separate teams to write versions of the software has found that independently written software routines do not fail in a statistically independent way.

It appears to be the case that such massive dependability gains can only be assumed on a theoretical level; in real-world applications, the overall cost/dependability ratio of a multi-version system is likely to be much lower than theoretical models suggest. The question of cost therefore becomes increasingly important; if the increased cost of developing a multi-version system were to always result in an extremely dependable system then the increased development budget may be justified. However, if this increased budget does not lead to significant dependability gains, then the additional spending may not be justified.

Despite this, it is generally accepted that when each channel of such a system has high dependability, the overall system will be more dependable than an equivalent high-dependability single-version (i.e. fault avoidance) system [HAT97]. It is also generally accepted that when the budget and available resources are so limited that each channel of the multi-version system is of poor quality, the single version method will produce more dependable results based on the same budget. When the dependability of individual channels is in-between these two extremes, it is unknown which of the approaches will produce the most dependable system.

The question of which approach produces the more dependable system when given a fixed and limited budget has never previously been investigated. This is important, as the finite budgets that many organizations are faced with means that a large number of realistic applications fall between these two extremes. In such cases it is unclear which design method will achieve higher dependability; for example, the lack of current understanding is one of the major reasons why the Boeing Corporation made the decision not to use the multi-version approach in the Primary

Flight Control system in its 777 aircraft. Although it is dangerous to rely on the multi-version approach without theoretical and empirical support evidence, relying on the single-version approach just because it has been used traditionally for years is equally unjustifiable.

We therefore ask: *Given a limited budget (e.g. money, people and time) for a given application, which development method for building a reliable system should we choose in order to achieve the maximum possible level of software reliability?*

2. AN EXPERIMENTAL STUDY

At the University of Durham, we have begun to address this issue by performing an ongoing series of experiments to compare the two approaches when given a fixed development budget. We describe here our initial experiment for a realistic industrial application, detailed in [TOW01], and combine these results with the results of the follow-up study.

In [TOW01], both a multi-version system and single-version system were developed to control the simulation of a factory production cell, both using a fixed amount of development resources. Resources were defined as *the total means available to a company for increasing production or profit*. In the real world, this encompasses a number of different elements, such as system cost and employee wages. In this experiment, the resource measured was time; in other words, the accumulative resources (time) allocated to the development of the multi-version system was to be the same as that allocated to the single-version system.

The factory production cell application [LOT96] was chosen as it was necessary to implement an application that was both small enough to construct within the development budget allocated, and also complex enough to have the potential for faults to be present within the system code. It was also desirable for the application to be real-time and embedded, as such systems invariably involve high reliability and safety requirements, as well as high timing constraints. The production-cell simulation, detailed in figure 1, consists of two conveyor belts, one of which delivers the raw units (blanks) into the system, and one of which moves the blanks out of the system once they have been fully processed. The unit also consists of four separate workstations, each of which has its own number. Depending on the type of a workstation, it can either be switched on and off by the controller software, or is permanently on. Two cranes are mounted on a racking which prevents them from occupying the same horizontal position at the same time, and are used to transport blanks around the system. Each blank has its own bar-code, which identifies which workstations it needs to be placed in, and the minimum and maximum amounts of time that it can spend within each workstation. Blanks may be processed either in specific (preserved) order, or in any (non-preserved) order, depending on the instructions in the bar-code.

The software controllers were needed to control the operations of the simulation; they needed to allow the simulation to process up to two blanks at any one time, whilst ensuring that the blanks were processed correctly within the appropriate time constraints. It was also necessary to ensure that the system remained safe; for example, it was imperative to ensure that the two cranes never collided with each other, and that no blank was placed in a workstation that already contained a blank.

Building Embedded Fault-Tolerant Systems for Critical Applications: An Experimental Study

Also, the feed belt needed to be controlled by the software in order to ensure that no more than two blanks entered the system at any given time, and that none fell off the end of the belt. The simulation and the controller software communicated via a first-in-first-out pipe mechanism, with communications being sent as ASCII text.

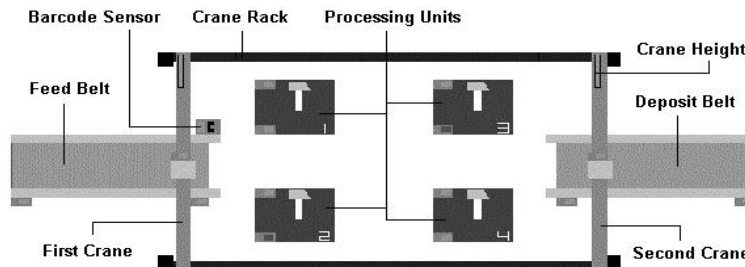


Figure 1 - The flexible production cell [LOT96]

Three programmers were used to develop the controller systems, with development separated into three equal time phases. The first time phase was used by each programmer to develop a working version of the controller software; at the end of this phase, the three programs were used as channels in the multi-version system. The second and third time phases were then used by each programmer to refine their respective versions and to exploit various methods and tools for improving the dependability of those versions. There was no differentiation between these two phases – they are merely noted in order to demonstrate that the additional time spent developing the multi-version systems was twice that of the original development time. These additional time phases resulted in each of the single-version systems having had the same amount of resources spent on them as the 3-version voter system. At the conclusion of this development process, an extensive testing plan was implemented in order to ascertain the dependability of the single-version systems and the 3-version system. An analysis was then performed to determine which of the methods had resulted in the most dependable software.

Unlike many other experiments of its type, this study did not make use of a pre-written requirements document; rather, the document’s production was considered to be part of the development process for which resources were allocated, and was produced jointly by the three programmers. Apart from this, there was no further interaction between programmers. In order to encourage diversity, each channel was developed using different development environments; these are shown in figure 2.

Channel	Operating System	Language
A	Sun Solaris 2.7	GNU C++
B	Red Hat Linux 6.0	GNU C++
C	Microsoft Windows NT 4.0	Java

Figure 2 - Operating system/language combinations for each channel

As with the Knight and Leveson experiment [KNI86], no single software engineering methodology was imposed on the programmers – each was allowed to develop their program using whatever method they saw fit.

3. TESTING AND RESULTS

There were only a finite number of possible states in which the simulation could be in (given that a maximum of two blanks could be processed by the systems at any one time) and these were specified as scenarios within the requirements document. Due to the real-time nature of the simulation, it was impossible to devise a series of tests that would cover all possible timing criteria, and so tests were formulated based upon these documented situations, consisting of a total of 440 tests per system. In all tests, timing constraints were set to random values.

3.1 The Multi-Version System

Of most concern when testing a multi-version system is the possibility of a common-mode failure that results in a multi-version system processing a task incorrectly; failures that result in the voter being unable to reach a consensus opinion and failures that cause the voter to perform an operation that results in the failure of a system as a whole are slightly less serious, as in either scenario, human operators can be alerted that the system is in a fail state.

The analysis of the three multi-version channels indicated that channel *A* and channel *C* suffered from unacceptably high rates of failure, with only channel *B* dependable enough to pass the acceptance test that was necessary before any channel can be entered into a dependable multi-version system. This is shown in figure 3.

As can be seen, the vast majority of failures were caused by channel *C*; this was largely due to the channel's inability to process blanks of non-preserved order, which automatically caused the channel to fail in 75% of all test inputs. Altogether, the 1320 tests performed on the three channels revealed a total of 480 faults. This means that for any given test on a single channel, there was a 36.36% probability of it resulting in a failure.

Channel	Probability of Failure
<i>A</i>	25.22%
<i>B</i>	1.59%
<i>C</i>	78.18%

Figure 3 - The probability of failure among the multi-version channels

However, at the conclusion of testing, only two common-mode failures had been discovered. This implies that even though two of the channels are unacceptably undependable, the overall multi-version system would only unknowingly forward an incorrect decision in 0.45% of tested possible situations. Figure 4 shows the overall distributed of faults in all tests.

The enforced diversity of operating system and platform resulted in diverse code; however, as the two common-mode failures suggest, the diversity of code does not necessarily lead to diverse distributions of faults. It is interesting to note that both common-mode failures were as a result of entirely unrelated faults within two of the channels; these faults resulted in the simulation moving into the same unrecoverable state, but one channel caused the system to fall into an infinite loop, whilst the other channel simply ceased to execute.

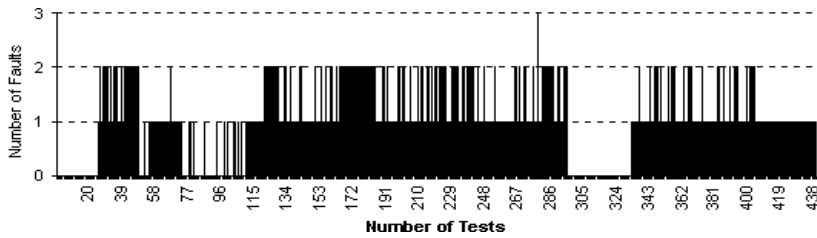


Figure 4 - The results of all tests performed on the multi-version channels

3.2 The Single-Version Systems

All three single-version systems exhibited increased levels of dependability over their corresponding multi-version channels, with the number of faults found in each system dropping significantly, as shown in figure 5.

Whilst the failure rates of channel A and channel C were still unacceptably high, both systems exhibited vastly improved dependability, with failure rates of 5.00% and 16.14% respectively, compared to the failure rates of 25.22% and 78.18% experienced with the multi-version systems on which they were based. Channel B also exhibited increased dependability, with its failure rate reduced to 0.91%, although all of the faults found in this channel were transient and so repeated testing may produce different statistics. Interestingly, the proportion of transient faults discovered while testing the systems remained much the same, as is shown in figure 6, and so whilst 55 repeatable faults were discovered among the single versions, the remaining faults appeared to be caused by timing anomalies with the Java simulation, and were therefore difficult to replicate with certainty.

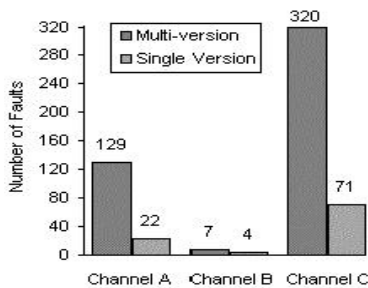


Figure 5 - The number of faults found during testing for each system

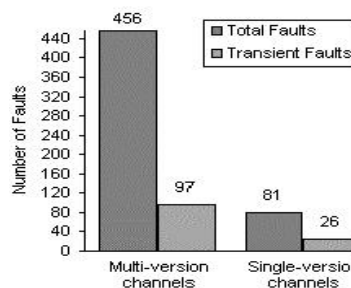


Figure 6 - The number of faults found during testing, and the number of them transient

The difficulty in pinpointing the cause of many of the transient faults perhaps underlines the difficulty in debugging real-time applications. One of the greatest areas of interest is that of how many undetected failures occurred in the single-version systems. There are two possible kinds of undetected failure; a blank can be left within the system instead of being extracted, or a blank can be processed through the system incorrectly. The occurrences of these are detailed in figure 7.

	Channel A	Channel B	Channel C
Blank in System	3	0	3
Blank incorrectly processed	3	4	20
Total Undetected failures	6	4	23
Undetected failure rate	1.36%	0.90%	5.23%

Figure 7 - Undetected failures in the single-version systems

As can be seen, *all three* single-version systems had an undetected failure rate that was greater than the 0.45% rate recorded in the multi-version system. Therefore, despite poor quality of components, the multi-version system still theoretically performed more dependably than either of the two poor quality channels in this aspect, and more importantly, would only unknowingly forward an incorrect decision in 0.45% of test inputs. The multi-version system therefore had a higher level of *safety* than the single-version system, although single-version system *B* was more dependable than the multi-version system in the value domain.

A potential problem with the system was its high granularity. In the specification, it was decided that voting would take place before each crane movement; due to the real-time nature of the simulation, this occurred almost constantly, which increased the scope for faults due to the consistent comparison problem. When viewed from the point of view of attempting to produce ultra-dependable software, this leads the experiment to come under the same criticisms as those made of the [KNI86] experiment by [AVI88a] – namely that the lack of rigor in the design paradigm resulted in many of the later weaknesses with the multi-version system.

3.3 The Improved Single-Version System

Although the initial experiment produced some interesting results, there were a number of areas of weakness that needed to be improved. The initial study used the second and third time phases to further test and enhance each individual single-version system; should no multi-version channel have been required, then each programmer would have had much more time to develop a rigorous and thorough specification, and the resulting single-version systems may have been more dependable. As it was, the necessity of completing the multi-version system on time meant that minimal work was applied to design. Related to this was the lack of a pre-written requirements documentation; although assigning the task of writing the requirements documentation in part of the development time was perhaps more realistic, the resulting documentation contained several faults and ambiguities.

In a follow-up study, the authors performed an experiment whereby a single-version controller system was developed independently of any multi-version channels, using an independent programmer, and based on a robust requirements document. The new system was allocated an identical period of development time, and on completion of this, a new series of tests were performed on this single-version system as well as the multi-version system developed in the initial experiment. In order to test the systems more accurately, tighter timing constraints were imposed upon all tests in order to highlight any weaknesses in each system. The results of these tests are shown in figure 8.

Building Embedded Fault-Tolerant Systems for Critical Applications: An Experimental Study

Single Version System:

No. of blanks	P x P Failures		NP x NP Failures		P x NP Failures		NP x P Failures		Totals	
	No.	%	No.	%	No.	%	No.	%	No.	%
1	0/64	0	0/64	0	-	-	-	-	0/128	0
2	20/144	13.89	17/144	11.81	13/144	9.03	17/144	11.81	70/576	12.15
									70/704	9.94

Multi-version System Channel A:

No. of blanks	P x P Failures		NP x NP Failures		P x NP Failures		NP x P Failures		Totals	
	No.	%	No.	%	No.	%	No.	%	No.	%
1	0/64	0	38/64	59.38	-	-	-	-	38/128	29.69
2	45/144	31.25	128/144	88.89	115/144	79.86	143/144	99.31	431/576	74.83
									469/704	66.62

Multi-version System Channel B:

No. of blanks	P x P Failures		NP x NP Failures		P x NP Failures		NP x P Failures		Totals	
	No.	%	No.	%	No.	%	No.	%	No.	%
1	0/64	0	2/64	3.13	-	-	-	-	2/128	1.56
2	14/144	9.72	17/144	11.81	19/144	13.19	25/144	17.36	75/576	13.02
									77/704	10.94

Multi-version System Channel C:

No. of blanks	P x P Failures		NP x NP Failures		P x NP Failures		NP x P Failures		Totals	
	No.	%	No.	%	No.	%	No.	%	No.	%
1	0/64	0	0/64	0	-	-	-	-	0/128	0
2	124/144	86.11	144/144	100	131/144	90.97	110/144	76.39	509/576	88.37
									509/704	72.30

Figure 8 – Results of testing the new single version system with multiple blanks (inputs)

As can be seen, the new single-version implementation has an overall failure rate of 9.94%. Although this is unrealistically high for a critical system, it compares favourably with the failure probabilities for each multi-version channel, and at first glance it would appear that the dependability (the reliability in particular) of the single-version system is much greater than that of the multi-version system. However, like the results of the initial experiment, once the safety attribute of dependability is investigated, the picture changes. When each system failure was analysed further, it was found that the new single-version system had an undetected failure rate of 3.27%. When the multi-version system was analyzed with the new test results, *no undetected failure was discovered*. This reinforces the initial experiment’s findings that although the multi-version system is less reliable, it is nonetheless safer.

4. CONCLUSIONS AND FUTURE EXPERIMENTS

It appears to be the case that for both our initial experiment and follow-up experiment, the traditional single-version approach produced more *reliable* systems in terms of overall numbers of faults and failures discovered, with the level of granularity and an ambiguous requirements document resulting in a less reliable multi-version system. However, the multi-version system resulted in fewer undetected failures occurring than in a single-version system, despite the poor quality of its individual channels. Most failures in one or two versions of the multi-

version system can be detected through result comparison. In these cases, the production cell can stop in a pre-defined safe state. Therefore, although the single-version systems produced fewer faults, the multi-version system could still be seen as being potentially the *safer* of the two approaches. It is evident that regarding the single-version method as a “seem-to-be” safer design decision for safety-critical applications is not generally justifiable, even for a given amount of resources.

Although the size of the software developed is still small compared to real-world applications (but certainly greater than that of [KNI86] and [AVI88b]), to develop more realistic software systems would be extremely difficult without commercial backing, and would be unjustified given that no previous experimental data of its type is available. Once we have gathered sufficient empirical evidence, we intend to seek out sources of data from the safety-critical industry, in order to validate our models further.

5. ACKNOWLEDGEMENTS

This work was partially supported by the EPSRC IBHIS and e-Demand projects.

6. REFERENCES

- [AVI77] A. Avizienis and L. Chen, “On the implementation of N -version programming for software fault-tolerance during execution,” in *Intl. Conf. Comput. Soft. & Appli.*, New York, pp.149-155, 1977.
- [AVI84] A. Avizienis and J.P.J. Kelly, “Fault tolerance by design diversity: concepts and experiments,” *IEEE Computer*, vol. 17, no. 8, pp. 67-80, 1984.
- [AVI88a] A. Avizienis and M.R. Lyu, “On the effectiveness of multi-version software in digital avionics,” in *AIAA/IEEE 8th Digital Avionics Systems Conference*, San Jose, pp. 422-427, Oct. 1988.
- [AVI88b] A. Avizienis et al., “In search of effective diversity: a six-language study of fault-tolerant flight control software,” in *18th Int. Symp. Fault-Tolerant Comput.*, pp.15-22, Tokyo, 1988.
- [ECK85] D.E. Eckhardt and L.D. Lee, “A theoretical basis for the analysis of multi-version software subject to coincident errors,” *IEEE Trans. Soft. Eng.*, vol. SE-11, no. 12, pp.1511-1517, 1985.
- [HAT97] L. Hatton, “ N -version design versus one good version,” *IEEE Software*, vol. 14, no. 6, pp.71-76, 1997.
- [KEL91] J.P.J. Kelly, T.I. McVittie, and W.I. Yamamoto, “Implementing design diversity to achieve fault tolerance,” *IEEE Software*, pp.61-71, 1991.
- [KNI86] J.C. Knight and N.G. Leveson, “An experimental evaluation of the assumption of independence in multi-version programming,” *IEEE Trans. Soft. Eng.*, vol. SE-12, no. 1, pp.96-109, 1986.
- [KNI90] J.C. Knight and N.G. Leveson, “A reply to the criticisms of the Knight and Leveson experiment,” *ACM Software Engineering Notes*, Jan. 1990.
- [LAD99] P. Ladkin et al, “Computer-related incidents with commercial aircraft,” <http://www.rvs.uni-bielefeld.de/publications/Incidents/>, 1999.
- [LAP95] J.-C. Laprie, “Dependability – its Attributes, impairments and means,” in *Predictably Dependable Computing Systems* (eds, B. Randell etc.), Springer-Verlag, pp.3-24, 1995.
- [LEV95] N.G. Leveson, *Safeware: system safety and computers*. Addison-Wesley-Longman, NY, 1995.
- [LOT96] A. Lötzbeyer and R. Mühlfeld, “Task description of a Flexible Production Cell with real time properties,” Internal FZI Tech. Report, Karlsruhe, ftp://ftp.fzi.de/pub/PROST/projects/korsys/task_descr_flex_2_2.ps.gz, 1996.
- [RAN95] B. Randell and J.Xu, “The evolution of the recovery block concept,” in *Software Fault Tolerance* (ed. M.R. Lyu), John Wiley & Sons, pp.1-22, 1995.
- [SCI00] http://www.scintilla.utwente.nl/shintabi/engels/thema_text.html
- [TOW01] P.Townend, J. Xu, M. Munro, “Building Dependable Software for Critical Applications: N -version design versus one good version,” in Proc. 6th IEEE Intl. Workshop on Object-Oriented Real-Time Dependable Systems, pp. 105-111, Rome, January 2001