Effect of Parallel Planning on System Reliability of Real-Time Expert Systems

Ing-Ray Chen, Member IEEE
National Cheng Kung University, Tainan

Key Words — **Real-time, expert system, parallel planning, software reliability model, system reliability.**

Summary & Conclusions — **Real-time expert systems (RTXS) are expert systems embedded in process-control systems which must plan & execute control strategies in response to external events within a real-time constraint. This paper presents a method for estimating the reliability of uni-processor & multi-processor RTXS. The paper discusses why there are intrinsic faults in RTXS programs that must be considered in their reliability modeling. Then, it shows that for uni-processor RTXS, no single planning algorithm can avoid all types of intrinsic faults. Finally, it presents a multi-processor architecture with parallel planning with the objective of reducing intrinsic faults of RTXS and improving the embedded system reliability. A robot control system illustrates the method.**

1. EMBEDDED REAL-TIME EXPERT SYSTEM

Acronyms[1]

CF	correctable fault
CPU	central processing unit (of a computer)
DTD	difficult to decide (for failures)[2]
DV	deadline violation (for failures)
ML	maximum likelihood
XS	expert system
RTXS	real-time XS
T&D	testing & debugging.

Notation

$R_{sys}(t)$	system reliability
$R_{soft}(t)$	software reliability
$R_{hard}(t)$	hardware reliability
λ_0	initial value of λ_1
θ	failure rate decay parameter for λ_1
λ_1	failure rate due to software CF
λ_2, λ_3	failure rate of [DTD, DV] failures
T	length of the T&D phase
t_i	CPU time at which CF i is observed and removed during the T&D phase
n	number of CF detected during the T&D phase
H_t	$(t_1, t_2, \ldots, t_n)$: failure history due to CF collected during the T&D phase
r	number of DTD failures detected during the T&D phase
f_i	failure level of DTD failure i detected during the T&D phase.
H_f	$(f_1, f_2, \ldots, f_r)$: DTD failure history collected during the T&D phase
q	number of DV failures detected during the T&D phase
β_0	$1/\theta$
β_1	$\lambda_0 \cdot \theta$
$\hat{}$	implies: ML point estimate
λ_h	hardware failure rate
w	number of hardware failures detected during the T&D phase
t_r	real-time deadline.

Other, standard notation is given in "Information for Readers & Authors" at the rear of each issue.

An embedded RTXS, *eg*, in factory automation, aircraft control, or robot control systems [13] must provide timely control functions in response to external sensor events. Each sensor event represents a mission assignment for which the embedded RTXS must plan a control strategy, and subsequently invoke the underlying hardware to execute the strategy within a t_r corresponding to the time interval between two successive sensor events. For a hard RTXS, failing to plan & execute a response within t_r can be catastrophic. Therefore, it is important to assess the system reliability before putting the system into operation.

An embedded RTXS affects system reliability in several ways. The most prominent is the software reliability of the embedded RTXS. Software failures are caused by design faults. Formal methods have been proposed to validate & verify RTXS designs [7, 18] to produce bug-free RTXS programs. However, even if the program can be proved to be completely devoid of bugs, there are characteristics of RTXS programs which must be considered explicitly in the software reliability model, *eg*,

1. Correctness/response-time tradeoff. In planning, there is often a strong s-correlation between time spent in formulating a strategy and the possibility that the strategy is correct. For example, a deep (detailed) knowledge rule base usually requires more time to execute than a shallow (summarized) knowledge rule base to produce an output for the same input but the output is more likely to be appropriate because of the use of deep reasoning. In rule-base XS, when there are many rules which can be fired simultaneously, firing more rules along many search paths can usually result in a better solution than firing only a few rules along a single search path. Therefore, as more rules are selected to fire, the more likely that the selected strategy is appropriate.

[1]The singular & plural of an acronym are always spelled the same.
[2]In the appendix, the DTD events are treated with *fuzzy & possibility* theory [22].

2. DTD correctness criterion. The correctness of the output of some XS programs is a DTD rather than a binary quantity, in the sense that one might not be able to state categorically whether the output is correct or not. For example, an XS program for controlling an automated factory might assemble acceptable products but perhaps not of the best quality or not in the best way. This problem is compounded by the fact that minor deviations from the optimum point are usually not easily discernible.

3. Intrinsic faults. Some of the fundamental techniques used in XS programs do not work successfully all the time. For example, the planning algorithm used for selecting rules to fire terminates only when some newly generated data meet the termination condition. Similarly, even the best heuristics might fail for certain cases. These faults are well known but cannot be eliminated easily. ◄

XS programs also directly affect the hardware reliability of the embedded system. Consider strategy #1 formulated by an XS program to execute two hardware components sequentially *vs* strategy #2 formulated to execute three similar components in parallel, to accomplish a mission. The hardware reliability of the mission is the product of the hardware reliabilities of the planning & execution phases. Strategy #2, although providing a higher execution-phase hardware reliability than strategy #1, might require the system to spend more planning time to explore extra solution paths before a solution is formulated and, consequently, might reduce the planning-phase hardware reliability. Therefore, strategy #2 might not provide a better hardware system reliability when compared with strategy #1. In RTXS, this tradeoff is compounded by the fact that if an XS program spends too much time in the planning phase trying to formulate the best solution, then there might not be enough time left for the execution phase even though the formulated solution is otherwise the best.

This paper advocates two approaches reminiscent to black-box and white-box testing strategies in software engineering to assess the reliability of RTXS, considering that an embedded RTXS might contain not only the conventional CF (design faults or program bugs) but also intrinsic non-CF associated with heuristic search or reasoning techniques which can cause the real-time deadline to be violated (due to excessive search) or produce control decisions that are not entirely acceptable (due to insufficient search).

The black-box approach is often used when there is no knowledge regarding the design & implementation details of the system and is based on probability modeling. It can yield a closed-form solution for the reliability of a RTXS as a function of time, or as a function of the number of missions encountered by the RTXS during its life [4], in accordance with a failure criterion [1] specified by the application. Under the black-box approach, the end product is a theory and a method with which the reliability of a RTXS can be calculated based on the failure data collected during testing.

The white-box approach is used when the design details are known. A reliability model for RTXS was developed in [5] based on the white-box approach using hierarchical modeling in combination with simulation and Petri net techniques. For a RTXS being modeled, it can yield a response-time distribution and a failure-level distribution (for partially correct outputs) which can be used to predict the system reliability according to the failure criterion specified by the application. ◄

Both the black-box and white-box approaches are extensions to [2] which considers that the reliability of an embedded system is the probability that an embedded XS can successfully formulate & execute strategies under real-time constraints.

This paper concerns the black-box approach. In [4], under the black-box approach, the software reliability issues of real-time artificial intelligence software running on uni-processor systems were studied. A modified Musa-Okumoto software reliability model [15] was used to consider an XS that might contain not only design & coding faults, but also intrinsic faults which cannot be removed even after the XS has been tested & debugged for a long time. This paper extends that work to assess the system reliability of uni-processor & multi-processor RTXS. An embedded-system reliability is approximated as:

$$R_{sys}(t) = R_{soft}(t) \cdot R_{hard}(t). \quad (1)$$

Eq (1) is an approximation by assuming mutual s-independence of software & hardware states, and does not consider common faults which affect both of them. $R_{sys}(t)$ is interpreted in a statistical sense over all possible events which the system might encounter during $[0,t]$.

This paper shows that no planning algorithm running on uni-processor systems can avoid all types of intrinsic faults of RTXS programs. Then, a multi-processor architecture with parallel planning is proposed to improve system reliability. A method for assessing the resulting system reliability due to parallel planning is given and the reliability improvement over a uni-processor system is illustrated with a real-time robot control system.

This work can be extended to analyze the effect of other parallel algorithms on the system reliability of parallel XS, *eg*,

- distributed algorithms with rule partitioning [3],
- blackboard-based algorithms [9],
- parallel Rete or Treat matching algorithms [6, 8, 14],
- parallel rule firing algorithms [11].

2. RELIABILITY OF UNI-PROCESSOR RTXS

For uni-processor systems, an embedded RTXS conceivably can: a) employ a fixed planning algorithm, or b) plan a quick, non-optimal strategy at the beginning of a mission assignment using a non-optimal planning algorithm and then use the remaining time to improve the quality of the strategy by switching to an optimal planning algorithm. Since the assessment method in this paper is based on the black-box approach, it can be generically applied to both cases. Assumption #1 is used for ease of presentation.

Assumptions

1. The system uses a fixed planning algorithm.
2. After the RTXS is designed & coded, it enters a T&D phase in which it is tested until a failure is encountered. If the fault causing the failure is a CF then it is removed from the program.
3. The 3 processes for $\lambda_1(t)$, λ_2, λ_3 are mutually s-independent. ◄

2.1 Software Reliability Assessment

To assess the software reliability of a RTXS program, the logarithmic Poisson execution time software reliability growth model [15] is modified to account for the intrinsic faults which are not removable but can exist in RTXS programs.

The general assumptions of software reliability growth models include: a) the correction of a fault does not introduce any new faults into the program and therefore the software reliability of the program grows as more faults are removed; and b) inputs to the program are selected randomly and s-independently from the input domain according to the operational distribution [17]. The logarithmic Poisson execution time model [15] assumes that failures occur as a nonhomogeneous Poisson process and that the failure rate decreases exponentially per failure experienced, *ie*, the first few failures yield large decrements in failure rate while later failures result in much smaller decrements. In sections 2 & 3, the Musa & Okumoto model [15] is used to estimate the failure rate of CF in RTXS programs in the operational phase.

For uni-processor systems, $R_{\text{soft}}(t)$ of a RTXS program can be estimated by splitting the failure process into 3 s-independent Poisson processes:

- $\lambda_1(t)$ is decreasing, reflecting the growth in the software reliability as a result of CF which are detected & removed during T&D;
- λ_2 is constant and due to intrinsic faults which cause DTD non-removable failures;
- λ_3 is constant and due to intrinsic faults which cause DV.

During the software T&D phase, an embedded XS program is tested with the anticipated operational profile [17] until a failure is encountered. By anticipated operational profile, we mean a set of test cases (or mission assignments) generated according to the probabilities with which they are anticipated to occur during the operational phase. If the failure is caused by a CF (program bug) then the fault causing the failure is located and removed from the program. Otherwise, the program is not modified because the failure is caused by a DTD or DV fault which is not removable. As a result of CF which are removed from the program during the T&D period, the program software reliability increases. Based on the Musa-Okumoto model [15], the failure rate of CF in the operational phase is estimated by:

$$\lambda_1 = \lambda_0/[\lambda_0 \cdot \theta \cdot \mathrm{T} + 1]. \tag{2}$$

λ_0 & θ are estimated by ML based on H_t [16].

Notation

$\psi_i(x)$ $[1 + x \cdot t_i]^{-1}$
$\phi_i(x)$ $[\ln(1 + x \cdot t_i)]^{-1}$.

$$\hat{\beta}_0 = n \cdot \phi_n(\hat{\beta}_1),$$

$$\left[\hat{\beta}_1^{-1} \cdot \sum_{i=1}^{n} \psi_i(\hat{\beta}_1)\right] - n \cdot t_n \cdot \psi_n(\hat{\beta}_1) \cdot \phi_n(\hat{\beta}_1) = 0.$$

The upper & lower limits of an approximate $(1-\alpha)$ s-confidence interval for β_1 are:

$$\hat{\beta}_1 \pm \kappa_{1-\frac{1}{2}\alpha}/I(\hat{\beta}_1)$$

$\kappa_{1-\frac{1}{2}\alpha}$ = appropriate s-normal deviate

$$I^2(x) = \tfrac{1}{2}(n/x) \cdot \phi_n(x) \cdot [4t_n \cdot \psi_n(x) - 1 + \psi_n^2(x) - 2x \cdot t_n^2 \cdot (1 + \phi_n(x)) \cdot \psi_n^2(x)]$$

The s-confidence intervals for β_0, λ_0, θ and consequently for λ_1 in (2) can be established by the substitution principle since they are all strictly monotonic functions in the permissible range of $\hat{\beta}_1$. Also,

$$\hat{\lambda_2} = r/\mathrm{T},$$

$$\hat{\lambda_3} = q/T.$$

For some real-time applications, a mission assignment is successfully accomplished as long as a strategy is planned and executed within a real-time constraint (*eg*, a bomb placement mission before the bomb explodes). For such applications, the notion of DTD failures does not exist and λ_2 is zero; for other applications, see the appendix.

$$R_{\text{soft}}(t) = \exp(-(\lambda_1 + \lambda_3) \cdot t) \cdot \sum_{n=0}^{\infty} \text{poim}(n;\ \lambda_2 \cdot t) \cdot G^{(n)}(1), \tag{3}$$

$G^{(n)}(\cdot)$ is defined in the appendix.

2.2 Hardware Reliability Assessment

Collecting hardware failure data by physically executing a plan formulated by the XS planning program and observing if the hardware execution fails during the testing period is impractical, because the hardware system fails rarely and it would require a very long testing period (*eg*, life-testing) to collect the hardware failure data. One approach is to consider the hardware system of a mission as having completely failed (various missions use different hardware components and structure functions as determined by software) if the predicted hardware reliability of the mission is less than a hardware reliability requirement. This approach allows the embedded system to be

tested without physically executing the plan formulated by the planning program. The hardware reliability can be considered as 1 between mission assignments.

Based on this approach, hardware failure data can be collected easily during the testing phase on a mission by mission basis.

Example A

1. Test case j causes the system to formulate a strategy with a given planning time;
2. The failure rate of the digital hardware is constant.

The hardware reliability in the planning phase for test case j is:

$$R_{\text{hard}}(j; \text{planning phase}) = \exp(-\lambda_p \cdot t_{p:j}).$$

Notation

$t_{p:j}$ planning time for test case j
λ_p failure rate of the digital system on which the XS is run. ◄

The planning phase is followed by an execution phase in which the plan is executed by the underlying hardware actuators. The hardware reliability of the execution phase of mission j can be estimated by computing the hardware reliability of the formulated strategy using common hardware reliability assessment techniques [12]. ◄

Example B

A robot system has a strategy to move its hand, arm, and leg actuators simultaneously to reach for an object such that as long as two actuators remain alive, the mission j can be accomplished. The hardware reliability of the execution phase is that of a 2-out-of-3 *parallel* system for the duration of mission j. This reliability can be computed easily since the component reliabilities of the hand, arm, and leg are usually known.

On the other hand, if the strategy is to move the leg actuator first and subsequently the hand actuator after the leg motion has stopped, then the hardware reliability of the execution phase is that of a *series* system connecting the leg and hand components, which again can also be computed easily. ◄

Hardware failures experienced during the T&D period can be caused by intrinsic faults of XS programs which determine the execution time and structure function of the hardware system through planning. Then, analogous to the approach used to estimate λ_2 in section 2.1:

$$\hat{\lambda_h} = w/T, \; R_{\text{hard}}(t) = \exp(-\lambda_h \cdot t). \tag{4}$$

Combining this result with $R_{\text{soft}}(t)$ in (3), the system reliability of an embedded XS can be calculated as a function of the operational time t.

2.3 *Example: A Robot XS Given*

1. A robot RTXS is embedded in a hard real-time environment.
2. The RTXS runs on a uni-processor system having a constant hardware failure rate $\lambda_p = 1.0 \cdot 10^{-7}$ failures/sec.
3. A planned strategy is executed by invoking the robot's mechanical components which have a constant failure rate of $\lambda_e = 5.0 \cdot 10^{-7}$ failures/sec.
4. The function of RTXS is to formulate a path connecting a start node to a goal node of a random graph to be determined in real-time. The robot is required to plan a path and subsequently execute the path within the execution time (traveling time from the start location to the end location) equal to the sum of the edge-costs along the solution path.
5. A mission incurs t_p planning time and t_e execution time.
6. Hardware unreliability requirement $= 10^{-6}$. ◄

The hardware system reliability for that single mission is:

$$\exp(-(\lambda_p \cdot t_p + \lambda_e \cdot t_e)),$$

which is compared with the hardware reliability requirement to determine whether that mission has caused a hardware failure on the uni-processor system.

This approach allows the robot system to be tested without physically executing the path planned by the embedded XS planning program to avoid the impractical problems associated with life-testing.

The input graphs are weighted graphs (with costs on edges) of 50-100 nodes not known beforehand. They are randomly generated during the T&D phase based on the anticipated operational profile. The t_r is 3 minutes. For each input graph (corresponding to a mission assignment) thus generated, if the solution planned by the robot is not optimal, then the system has either partially failed or experienced a DTD failure, because there is higher risk of accidents (*eg*, on bumpy roads). The DTD failure level in a mission assignment is proportional to the difference in cost between the optimal solution path and the solution path found by the robot program. That is, the failure level of that mission is:

$$(C - C_{\text{opt}})/C_{\text{opt}},$$

Notation

C, C_{opt} mission cost of the [planning algorithm, optimal] solution, $C > C_{\text{opt}}$.

This is recorded as one of the f_i in H_f during the testing phase so as to estimate the a, b parameters of the Beta(a, b) distribution (see appendix). If the sum of DTD failures encountered exceeds 1, the system has completely failed.

There are two options in designing an XS planning algorithm:

1. Use an optimal search algorithm such as A^* with lower-bound estimates [21] as the underlying planning algorithm to avoid DTD failures. A^* operates by ranking all partially explored solution paths by the sum of cost accumulated so far (g) and a lower-bound estimate of the cost remaining (h), and always expanding the solution path with the minimum $f = g + h$

value among all paths until the goal node is found. Because the estimates of the cost remaining can be underestimated (*eg*, the linear distance between the frontier node of a solution path and the goal node can be used as the estimate of the remaining cost), the solution path found is optimal [20, 21], thus ensuring that no DTD failure happens during the robot's operational time. Nevertheless, a potential difficulty with A* is that the robot might not be able to plan & execute a solution within t_r, due to excessive planning.

2. Avoid DV failures by using a planning algorithm that can formulate a solution path more quickly, although possibly less optimally. One such algorithm (a variation of A^*) is A^*_ϵ [19]. In addition to maintaining an OPEN list keeping all partially explored solution paths (or all rule instances) sorted in ascending order of costs, A^*_ϵ also maintains a FOCAL list keeping a few partially explored solution paths sorted in ascending order of the remaining planning effort required to find a solution (minimum number of nodes that must be visited before the goal node can be found). A^*_ϵ always expands the solution path at the front of the FOCAL list, rather than the front of the OPEN list, until the goal node is found, so that it will only expand a solution path that never deviates by more than $100\epsilon\%$ in cost from the best solution path but promises to offer the fastest planning time. For example, $\epsilon = 0.1$ means that the cost of the solution found by $A^*_{\epsilon=0.1}$ can only be worse than that found by A^* by 10%.

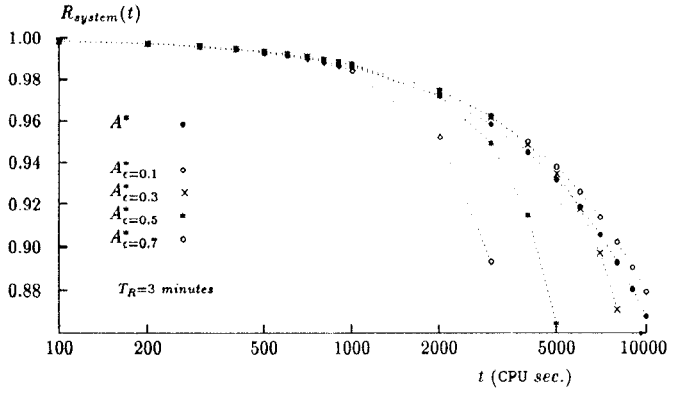

Figure 1. Robot-Program Reliability for Uni-Processor Systems Running A^* & A^*_ϵ

Figure 1 shows the robot system reliability running A^* & A^*_ϵ as a function of the operational time. Each system version (running A^* or A^*_ϵ) was tested for $\approx$57 CPU hours using the same test profile; software & hardware failure data were collected to compute: 1) point estimates of the failure rates, and 2) system reliability based on (1), (3), (4). In general a robot system running A^*_ϵ provides better reliability than a system running A^* when ϵ is small and the operational time is not too long (*eg*, less than 1 day of continuous operation), because under this situation, DV failures will more likely cause system failures than DTD failures. However, as ϵ becomes larger or the anticipated operational time becomes longer, the advantage of A^*_ϵ over A^* disappears quickly because then DTD failures dominate DV failures. Neither planning algorithm can provide the robot system with the best reliability for all situations.

3. RELIABILITY OF MULTI-PROCESSOR RTXS

Section 2 showed that a uni-processor RTXS sometimes can benefit from using a non-optimal planning algorithm because it can plan a strategy more quickly and thus can avoid DV failures. On the other hand, the system sometimes can benefit from using an optimal planning algorithm because it can always formulate an optimal solution and thus can eliminate DTD failures. However, no single planning algorithm can avoid both DTD & DV failures. A multi-processor system with parallel planning can avoid both types of intrinsic faults. Under this architecture, a central scheduler:

- distributes (the same) mission assignments to processors,
- collects strategies formulated from all assigned processors,
- selects the best strategy to execute within t_r.

A timeout mechanism can be used to ignore delaying results. Using the robot XS as an example, a dual-processor system can be used to run A^* & $A^*_{\epsilon=0.1}$ simultaneously on two processors. The scheduler first can receive the strategy planned by the processor running $A^*_{\epsilon=0.1}$ because $A^*_{\epsilon=0.1}$ requires less planning time. Then, based on the information returned by the strategy planned by $A^*_{\epsilon=0.1}$, the scheduler estimates the maximum length of the time interval within which it can wait for a response from the processor running A^* without a DV if the strategy planned by $A^*_{\epsilon=0.1}$ is to be executed at a later time. If within that time interval, the processor running A^* can return a planned strategy, then that strategy is executed to avoid DTD failures; otherwise, the strategy planned by the processor running $A^*_{\epsilon=0.1}$ is executed to avoid DV failures. Therefore, with parallel planning both DTD & DV failures can be appreciably reduced and the system reliability can be greatly improved. In this design, the scheduler should be fault-tolerant because if it fails, the system fails.

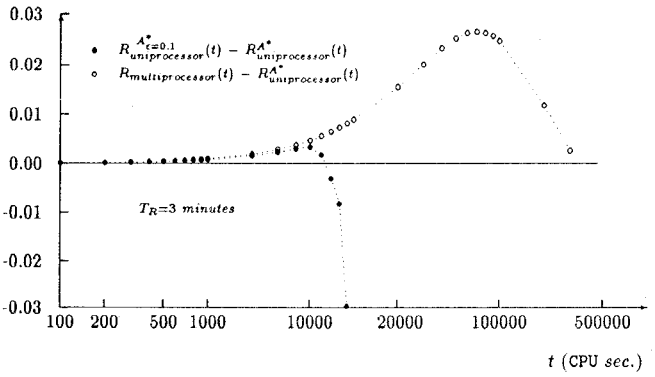

Figure 2. Reliability Difference Between Systems With & Without Parallel Planning

Figure 2 shows the difference in reliability between a uni-processor system running A^* only and:

1. a uni-processor system running $A^*_{\epsilon=0.1}$ only;
2. a dual-processor system running A^* & $A^*_{\epsilon=0.1}$ simultaneously on two processors.

The differences are shown in:

$R^{A_{\epsilon=0.1}^*}_{\text{uniprocessor}}(t) - R^{A^*}_{\text{uniprocessor}}(t)$, for case #1

$R_{\text{multiprocessor}}(t) - R^{A^*}_{\text{uniprocessor}}(t)$, for case #2.

The software failure data for the dual-processor system were manually generated on a mission by mission basis by comparing the two sets of output data obtained from the two uni-processor systems operating under A^* & $A^*_{\epsilon=0.1}$. For example, if a mission (test graph) requires $A^*_{\epsilon=0.1}$ to spend 1.5 minutes to find a solution path which has a failure level of 0.1 and requires 1 minute to execute (which we know by calculation), then we check whether A^* can find a solution in 2 minutes for the same mission to meet $t_r=3$ minutes. If the answer is yes, then the path found by A^* is selected to execute in order to avoid a DTD failure by $A^*_{\epsilon=0.1}$; otherwise, the path found by $A^*_{\epsilon=0.1}$ is executed to avoid a DV failure by A^*. In either case, no DV failure is observed for that mission. Of course, where even $A^*_{\epsilon=0.1}$ cannot find a solution within the time constraint, then the system has a DV failure. The hardware failure data of the dual-processor system are collected using a similar procedure. The calculation of the hardware reliability of the dual-processor system itself is included in the calculation of the hardware reliability of the planning phase as each mission is considered, and thus is already included in the calculation of the overall $R_{\text{hard}}(t)$.

After the failure data of the dual-processor robot system are collected this way, the same set of equations, (1), (3), (4), were used to compute the system reliability, considering the case in which the speed of a single CPU in the uni-processor and dual-processor systems is the same; the dual-processor system contains two CPUs.

Figure 2 illustrates the effect of parallel planning on RTXS reliability. The multi-processor design with parallel planning can improve the system reliability over a uni-processor system running either A^* or A^*_ϵ because both DTD & DV failures can be reduced. Even for a long operation time, the dual-processor design can maintain at least the same reliability as that of a uni-processor system running A^* because of infrequent occurrences of DTD failures.

ACKNOWLEDGMENT

This work is supported in part by the National Science Council of R.O.C. under grant NSC-85-2213-E-006-069 and NSC-86-2745-E-006-020.

APPENDIX

This appendix treats the DTD failures of λ_2 as *fuzzy* [22] failures. The system can be considered as having experienced a fuzzy failure if the solution planned is not the best solution for accomplishing a mission even though the real-time constraint is satisfied, *eg*, a decision to shut down a system when there is no need to do so. For this latter case, we can model the fuzzy output of XS programs with a distribution of fuzzy failures over [0,1], with 0 denoting a benign (no) failure and 1 denoting a definite failure. Specifically, fuzzy failures can be assumed to have a distribution G. The system can be considered as having failed if the sum of the fuzzy failures encountered exceeds 1; this is the accumulation criterion, and other fuzzy failure criteria are in [1]. Then, the software reliability of the system during the operation phase is given in (3).

$G^{(n)}(1) = n$-fold convolution of $G(1) =$

$$\begin{cases} 1, \text{ if } n=0 \\ G(1), \text{ if } n=1 \\ \int_0^1 G^{(n-1)}(1-y)\, dG(y), \; n > 1. \end{cases}$$

To use (3), one more set of failure data is needed to estimate the G, *viz*, H_f. A reasonable model for G is the Beta(a, b) distribution [10] with pdf:

$$g(x) = \begin{cases} [\Gamma(a+b)/[\Gamma(a)\cdot\Gamma(b)]]\cdot x^{a-1}\cdot(1-x)^{b-1}, \\ \quad \text{if } 0 \le x \le 1 \\ 0, \text{ otherwise.} \end{cases}$$

The $\hat{a}$, $\hat{b}$ can be calculated using H_f.

REFERENCES

[1] F.B. Bastani, I.R. Chen, T.W. Tsao, "Reliability of systems with a fuzzy failure criterion", *Proc. Ann. Reliability & Maintainability Symp*, 1994, pp 442-448.

[2] I.R. Chen, F.B. Bastani, "Effect of artificial intelligence planning-procedures on system reliability", *IEEE Trans. Reliability*, vol 40, 1991 Aug, pp 364-369.

[3] I.R. Chen, B. Poole, "Performance of rule grouping on a real-time expert system architecture", *IEEE Trans. Knowledge & Data Engineering*, vol 6, 1994 Dec, pp 883-891.

[4] I.R. Chen, F.B. Bastani, T.W. Tsao, "On the reliability of AI planning software in real-time applications", *IEEE Trans. Knowledge & Data Engineering*, vol 7, 1995 Feb, pp 4-13.

[5] I.R. Chen, T.W. Tsao, "A reliability model for real-time rule-based expert systems", *IEEE Trans. Reliability*, vol 44, 1995 Mar, pp 54-62.

[6] C.L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem", *Artificial Intelligence*, vol 19, 1982, pp 17-37.

[7] R.F. Gamble, G.-C. Roman, H.C. Cunningham, "Applying formal verification methods to rule-based programs", *Int'l J. Expert Systems*, vol 7, num 3, 1994, pp 203-237.

[8] A. Gupta, *Parallelism in Production Systems*, 1987; Morgan Kaufman.

[9] B. Hayes-Roth, "A blackboard architecture of control", *Artificial Intelligence*, vol 26, 1985, pp 251-321.

[10] P.G. Hoel, S.C. Port, C.J. Stone, *Introduction to Probability Theory*, 1971; Houghton Mifflin.

[11] T. Ishida, "Parallel rule firing in production systems", *IEEE Trans. Knowledge & Data Engineering*, vol 3, 1991 Mar, pp.11-17.

[12] B.W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, 1989; Addison Wesley.
[13] T.J. Laffey, P.A. Cox, J.L. Schmidt, *et al*, "Real-time knowledge based systems", *AI Magazine*, 1988 Spring, pp 27-45.
[14] D.P Miranker, B.J. Lofaso, "The organization and performance of a TREAT-based production system compiler", *IEEE Trans. Knowledge & Data Engineering*, vol 3, 1991 Mar, pp 3-10.
[15] J.D. Musa, K. Okumoto, "A logarithmic Poisson execution time model for software reliability measurement", *Proc. 7th Int'l Conf. Software Engineering*, 1984 Mar, pp 230-237.
[16] J.D. Musa, A. Iannino, K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, 1987, pp 303-351; McGraw-Hill.
[17] J.D. Musa, "Operational profiles in software reliability engineering", *IEEE Software*, vol 10, 1993 Mar, pp 14-32.
[18] T.J. O'Leary, M. Goul, K.E. Moffitt, A.E. Radwan, "Validating expert systems", *IEEE Expert*, 1990 Jun, pp 51-58.
[19] J. Pearl, *Heuristics*, 1984; Addison-Wesley.
[20] E. Rich, *Artificial Intelligence* (2nd ed), 1991; McGraw-Hill.
[21] P.H. Winston, *Artificial Intelligence* (3rd ed), 1992; Addison-Wesley.
[22] L.A. Zadeh, "Fuzzy sets and information granularity", *Advances in Fuzzy Set Theory & Application* (M.M. Gupta, R.D. Ragade, R.R. Yager, *Eds*), 1979; North-Holland.

AUTHOR

Dr. Ing-Ray Chen; Inst. of Information Eng'g; Nat'l Cheng Kung Univ; No. 1, University Road; Tainan, TAIWAN - R.O.C.
Internet (e-mail): irchen@iie.ncku.edu.tw

Ing-Ray Chen (M'90) received the BS from the National Taiwan University, and the MS & PhD in Computer Science from the University of Houston. He is a Professor of Computer-Science & Information-Engineering at the National Cheng Kung University. Prior to joining Cheng Kung, he was an Associate Professor of Computer & Information Science at the University of Mississippi. His research interests are in reliability & performance analysis, and real-time intelligent systems. Dr. Chen is a member of the IEEE and ACM.

Manuscript TR95-144 received 1995 September 15; revised 1996 November 20

Responsible editor: R.J. Loomis Jr.

Publisher Item Identifier S 0018-9529(97)02336-1 ◄TR►

Bayes Inference for S-Shaped Software-Reliability Growth Models

(*Continued from page* 80)

Dr. Tae Young Yang; Dep't of Mathematics; Myongji Univ; Yongin, Kyonggi, Rep. of KOREA.
Internet(e-mail): tyang@wh.myongji.ac.kr

Tae Young Yang is an Assistant Professor at the Myongji University. He received a BS (1985) in Mathematics from Korea University, a MS (1987) in Statistics from the University of Vermont, and a PhD (1994) in Statistics from the University of Connecticut. He has done research in software reliability and Bayes inference for stochastic point processes. He has been a visiting scholar at Stanford University.

Manuscript TR95-138 received 1995 September 6; revised 1996 December 9

Responsible editor: R.A. Evans

Publisher Item Identifier S 0018-9529(97)03033-9 ◄TR►