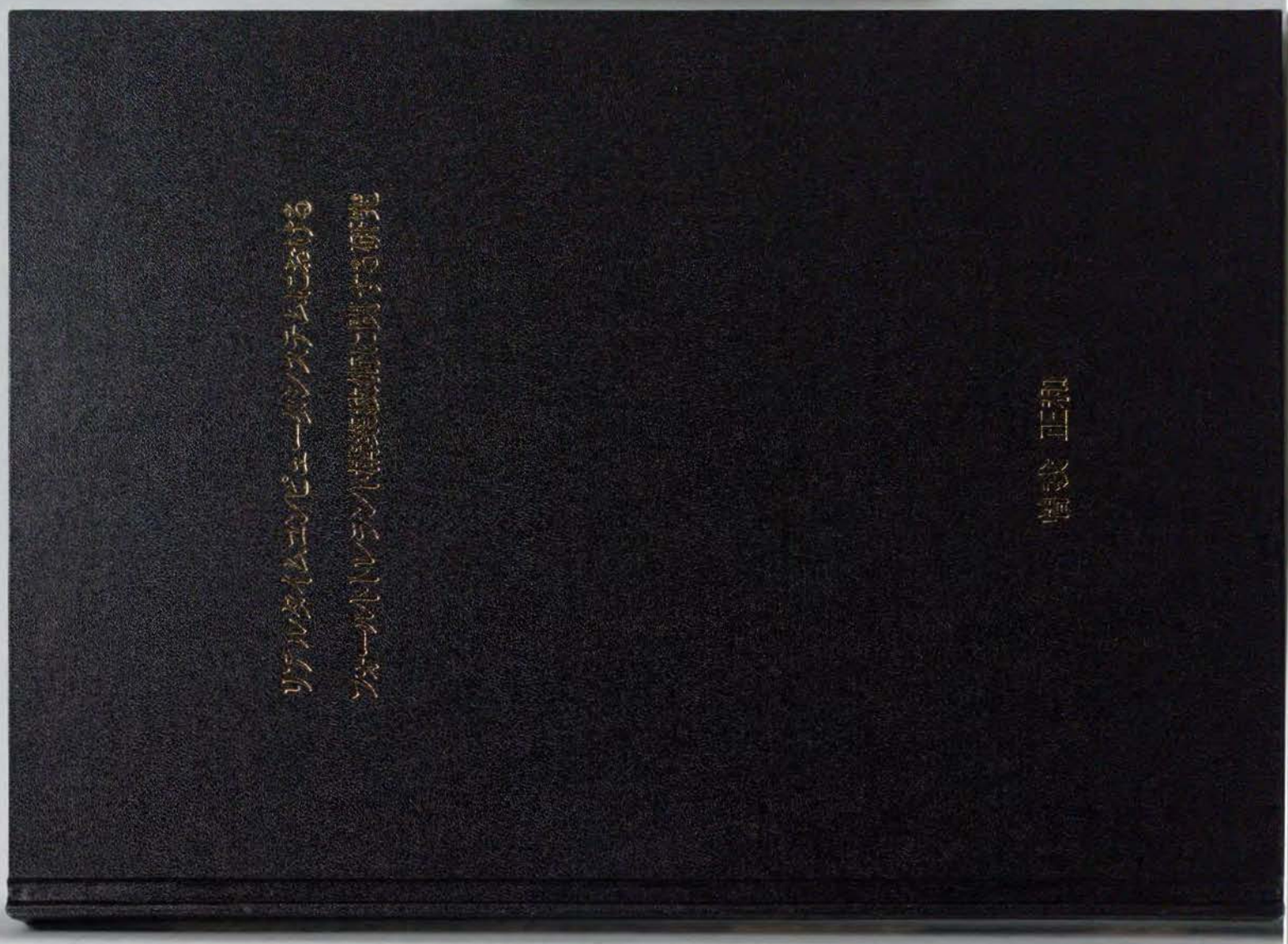

Kodak Color Control Patches
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
Kodak Gray Scale
C
Y
M
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19

リアルタイムコンピュータシステムにおける
フォールトトレラント構築技術に関する研究

曽我 正和

目次

第1章

序　論

第 1 章

序論

1.1　研究の背景及び目的

半世紀前に、高速に弾道計算をする機械としてコンピュータは誕生した。真空管を主要回路素子とするＥＮＩＡＣ (1946) である。このコンピュータは、期待通りの計算能力を発揮したが、一方で、ＭＴＢＦも短く、約１時間であった。以来、よく知られているように、コンピュータは、その回路素子の面では、トランジスタ、ＩＣ、ＬＳＩ、と長足の進歩を遂げ、現在では、ワンチップマイクロプロセッサの姿となっている。

応用面では、弾道計算に始まり、この種の科学技術計算用途、続いて企業の経理計算、オンライン在庫管理、預金管理、鉄鋼プラント、電力プラント、化学プラント等におけるリアルタイム制御、通信回線の蓄積交換、ＣＡＤ、ＯＡ、家電品制御、個人携帯用品に至っている。

このように、現在のコンピュータはスーパーコンピュータから家電品の片隅に埋め込まれたマイクロプロセッサに至る迄、大小さまざまの姿で、個人の生活、企業活動、社会基盤に深くかかわっている。

コンピュータが各種の姿で、各種の用途に、深く入り込むにつれて、それが一旦故障したときの影響もさまざまである。個人携帯端末が故障した場合の影響は、一般にはその個人の便宜が妨げられる程度であり、それ程深刻な影響はないであろうし、代替手段の選択にも、自由度や時間的余裕が適当にあるのが普通と考えられる。

しかし、他方で、極めて深刻で、広範な影響を及ぼす場合も多々ある。以下に幾

つかの代表的システム事例と一般に予想し得る不具合状態を幾つか示す。

表 1.1: 代表的なシステム事例と故障事の不具合事例

システム事例	不具合状態事例
・オンラインバンキングシステム	預金手形業務停止
・列車運行システム	列車停止
・国際メッセージスイッチシステム	商社国際通信機能停止
・空港管制システム	離着陸航空機誘導停止

上例では、不具合状態の一例としてシステムが停止した場合を示しているが、システム設計者の立場から考えると、システムが停止するのは、真の最悪事態ではない。最悪事態は、システムが一見動いているように見えて、実は誤った出力を出したときである。このときの方が、不渡り手形が出たり、死傷事故が起ったりする危険が大きい。身近な実例として、平成５年７月に常磐線踏切で列車が近づきつつあるときに遮断機が上がり、死傷事故が発生した。この例ではマイクロプロセッサが関与したかどうかまだ解明されてはいないが、システムとしては誤った出力を出してしまった事は明らかである。

フォールトトレラントシステムは、上例のような重要使命を担うシステムにおいて、システムのどこかに部分的障害が発生しても、システム全体としては正常運転を続行させたいとするニーズから生まれたシステムである。

正常運転という意味は、予め定められたサービスを提供しつづける、との意味であり、当然ながら前掲の例のようにシステムとして不都合な出力を出してはいけない。これは単に停止しないで運転を続けるという意味ではない。勿論、停止も又、不都合な状態ではあるが、その一つのモードである。

本研究の目的は、上記のような背景の下に、フォールトトレラントシステムを構築するための技術を明確化し、システムとして構築するための設計技法を確立するこ

とである。

1.2 フォールトトレラントシステム発展の経緯

フォールトトレラントシステムは、その中枢となるコンピュータの発展、とりわけ回路素子の発展に伴って、考え方、実現の仕方が変わってきている。本節では、過去の事例をベースに、フォールトトレラントシステムの考え方、要素技術、そしてシステム構築の発展の経緯に簡単に触れる。

1.2.1 エラー検知技術

フォールトトレラントシステムを実現するためには、大きく分けて２つの技術が必要になる。１つはフォールト又はフォールトの結果起きるエラーの検知技術であり、他の１つは冗長構成技術である。

エラー検知技術の原点は、1834 年に D.Lardner が述べたとされる「計算の過程で発生するエラーに対しては、最も確かで有効なチェックは、同じ計算を分離して独立した計算機で行なわせることであり、そしてそれらの計算機が異なる方法で計算すれば、このチェックはいっそう決定的となる。」との言葉に遡ることができる [1]。この指摘は含蓄に富んでおり、今日においてもなお新鮮である。ただし、これをそのまま実施することは、大きなコスト負担を伴うため、時代によって許されうるコスト負担の範囲内に抑えるべく、色々と代替手段を探してきたのが過去経緯である。

(1) パリティチェック

ENIAC に遅れること僅か５年、1951 年に初の本格的商用機 UNIVAC-1 において早くもパリティチェックが用いられた。このコンピュータは 5000 本の真空管を使っており、主に国勢調査の集計に用いられた。フォールトトレラントと言うよりも、演算結果が正確であってほしいとの要請による。なおパリティチェックは初期のシリアル演算式コンピュータにおいては、経済的に実現できるので一旦普及したが、その後パラレル演算式コンピュータになると中核の算術演算／論理演算ユニット部分において実現が非常に複雑となり、コスト上、性能上の負担に耐えきれずに姿を消してしま

い、周辺のインタフェース部とか、バスデータ部に残るのみである。

(2) エラー検知／訂正符号

1950 年に Hamming によってエラー検知／訂正符号が提案された [2]。その後 1960 年にＩＢＭ Stretch にて主メモリへの採用が開始され、1970 年にはＩＢＭ 3330 磁気ディスクにおいてバースト誤り検知／訂正符号が採用され始めた。これらの冗長符号は、高々２割程度の冗長ビットの付加と、比較的簡単な一式の符号生成／チェック回路の付加とにより実現できるので、主メモリ、外部メモリ、データ伝送回路には大いに普及し、今日でも常識的に使われている。

しかしながら、やはりパラレル算術演算／論理演算を中核とする CPU においては、至るところでデータビット構成が複雑に変化するため、符号生成とチェックをその都度やるにはあまりに性能低下の負担が大きすぎて採用されてはいない。

(3) 二重照合チェック

算術演算／論理演算ユニット或は制御回路については、性能上の低下を最も少なく抑えられる方式が並列二重照合チェックである。しかし、コスト上の負担が大きかったため、初期の UNIVAC-1 で一部用いられたあと採用例がなく、MSI 素子による CPU が出現して部分的に採用した例が出てきた [3]。その後、LSI 素子を経てワンチップマイクロプロセッサとなった現在では、チップまるごと並列二重照合する事例が随所に見られる。ストラタス社の XA-2000 （ペア＆スペア） (1982) が商用機としての代表的な例である。

(4) 三重多数決チェック

1960 年に Tripple Modular Redundancy (TMR) として、三重多数決方式が実現された [4]。単一エラーの検知の機能だけを見れば、二重照合チェックの場合と変わらないが、多数決の場合はその場でエラーモジュールを特定することができる。

1970 年代後半から 1980 年代にかけては、ワンチップマイクロプロセッサの出現にともない、幾つかの TMR に関する改良が報告され [5] [6]、1990 年にはタンデム社

Integrity 82、1992 年には日立 FT6000 等の商用機が出現している。

(5) その他のエラーチェック

1964 年に出たＩＢＭ 360 では、メモリ保護チェック、不正命令チェック、および特権命令チェックを行なっている [7]。これらは、バウンダリチェックとでも呼ぶべきエラー検知事例であり、ソフトウェアバグによるエラー、そして一部のハードウェアフォールトによるエラーを検知する。

また、ペリオディカルヘルスチェックは、短い診断ソフトウェア（またはファームウェア）を一定周期、またはアイドルタイム中に流すやり方であり、比較的初期から広く採用されていた [8]。

(6) ソフトウェアのバグの検知

初期のコンピュータは、ハードウェアの信頼性の低さのため、ハードウェアのフォールトが注目を集めていたが、素子の集積度の向上に伴ってハードウェアの信頼性が向上し、他方でソフトウェアの量の拡大と共にソフトウェアバグが増大してきた。1987 年のベルのレポート [9] では、電子交換システムのシステムダウンの約１／３がソフトウェアに起因するとされている。近年、我々の身のまわりに一般的に見られる業務処理システムでも、稼働開始後１年未満の時期とか新規にソフトウェアを改版したあとの半年程度とかではソフトウェアのバグによるシステムダウンが過半数を占めるとの実感がある。従って、フォールトトレラントシステムを考える際には、フォールトの対象としてソフトウェアのバグを看過し得なくなっている。

まず、ソフトウェアバグの生起頻度を適当な分布関数で近似しようとする試みがなされている [10] [11]。しかし、現実のバグはなかなか統計的な扱いに乗らず、まだ模索フェーズにある。1.2.1(5) で述べた IBM 360 やそれ以降の汎用コンピュータで採用されているバウンダリチェックは、ソフトウェアバグに起因するエラーの検知に有効であることは間違いないが、関所のような機能であるから、エラー現象が関所を通らない或るエリアの中で閉じている限りは検知にかからないという弱点がある。

筆者は、ソフトウェアバグの検知の問題は、バグの数とか発生間隔とかの現象に

目を向ける以前に、潜在しているバグが顕在化するときのメカニズムとその加速手法を追求すべきと考えており、第2章にて論ずる。

(7) 相互監視方式

1980年にタンデム社が発表したNonStopシステムは、疎結合された複数のコンピュータが相互に"I'm alive"信号を交信し合い、健全走行していることを監視し合うシステムである[12]。信号を一定期間出さなかったコンピュータは異常と認められる。タンデムは、このシステムはソフトウェアバグに対しても有効である、としている。ということは、ソフトウェアバグが顕在化して、ソフトウェア走行に何らかの不具合が発生したとき、"I'm alive"信号は途絶えなければならない。他方で、正常時には、ソフトウェアが如何なるループに入ろうが定期的に信号を発生しなければならない。このように、バグセンシティブかつタイムクリティカルに信号を発生することは、ソフトウェア作成上特別な工夫が必要と考えられるので、市販ソフトウェアや既成アプリケーションソフトウェアをそのままで組み込むことは不可能と思われる。

1.2.2 冗長構成技術

以上でエラー検知技術の発展経緯を概観したが、フォールトトレラントシステムを構築するためのもう一つの大きな要素技術は冗長構成技術である。以下に、簡単に発展の経緯を見る。

(1) 冗長符号

1.2.1 (2)に述べたエラー訂正符号を用いた主メモリ装置やディスク装置は、自己の内部に冗長ビットを持っており、それ自身で冗長構成を形成していると見ることができる。これらは、エラー検知と同時にそのエラーを訂正する能力を持つ。従って、装置の切替えは何ら必要としない。こういう冗長メカニズムを静的冗長とも呼ぶ。

他方、システムの中心となるCPUについては、エラー訂正符号の適用は性能面、コスト面から困難であり、従ってエラーを起こしたCPUはそれが固定的なエラーならば、もはや運転を継続することはできない。従って、予めこの事態に備えて用意されていた予備CPU或は冗長CPUが何らかの方法で運転を引き継ぐことになる。

このやり方に幾つかの方法があり、過去の事例を概観する。技術的には第2章以降で論ずる。

(2) 機能ユニットの二重化

1964年ベルから発表された電子交換機NO1.ESS.では、バス、CPU、メモリ等の機能単位に二重化（または多重化）がなされていた[13]。CPUでは、二重照合によりエラー検知を行ない、検知したときは、いづれがエラーであるか判定するための診断プログラムを流し、そのあとシステム再構成を行なう。この方式は、その後あまり普及しなかったが、1982年ストラタスのペア＆スペアがこの流れの延長線上に再登場した。ただし、ペア＆スペアは四重化と見るべきである。（後述）

(3) 機能ユニットの三重化多数決

1967年Saturnの誘導制御系に、三重化CPUが使用されていた。1台が故障しても多数決により正常な残り2台が特定され、そのまま運転を継続できる。その後、1990年にタンデムからIntegrity S2、1992年に日立からFT6000が商用機として発表された。本論文の第4章では、この方式での新しいシステムを提案している。

(4) 二重系システム

1964年、国鉄は座席予約システムMARS101を実用開始した[14]。このシステムは二重系で、照合チェックを行ない、エラーが発見されたときは、いづれかがエラーかすぐには判定できないため、一旦両系ダウンした。エラーが瞬時エラー（後述）の場合は、再び二重系で立ち上げるが、片系の固定エラーが判明したときは、残りの片系で一重運転を行なった。このときはエラー検知能力が相当低下したと推定されるが、詳細は明らかでない。

1968年、同じく国鉄で貨車ヤード自動化システムYACが稼働開始した。このシステムのエラー検知は二重系照合を使わず、単系独立に検知する工夫をしており、片系エラー時の残りの片系の運転は全く正常に行える。

YACに関する詳細は、本論文第3章、第4章にて論ずる。

(5) 二重系+スペア システム

1972年、国鉄で新幹線運行制御システム COMTRACの運用が開始された[15]。このシステムの運用系は前節のMARSと同じく二重系で照合チェックをしている。不一致でエラー検知すると、各系にて診断プログラムが動き、自己の正常性を確認する。その後スペア系がエラー系に代替されてシステムに投入され、システムは二重系として再開する。

(6) 疎結合マルチプロセッサシステム

前節で述べたタンデムのNonStopシステム(1980)は、疎結合マルチプロセッサシステムであり、疎結合された複数台CPUとそれらをつなぐ複数のバス、および2台のCPUからデュアルアクセスされるディスク群からなる。正常な各CPUは別々の負荷を持っている。あるCPUが異常と判定されると、そのCPUの業務は隣接CPUに引き継がれる。そのために、隣接CPU同志は常時業務の区切りで進行経過情報を交換し合っている。この情報と、デュアルアクセスディスクにある最新のファイル情報により業務が引き継がれる。

(7) デュアル×デュアル プロセッサシステム

前節で触れたストラタスのペア&スペア システム(1982)がこのデュアル×デュアル方式である。クロックレベルで同期走行する2個のCPUチップ(デュアル)を1個のモジュールとし、これと対(ペア)になる同様の別の1個のモジュールがあり、都合2個のモジュール、4個のCPUチップがクロックレベルで同期走行、同一作業を行なう。このペアを単位として密結合マルチプロセッサが構築されている。(4×nプロセッサ、nの最大は6)

モジュール内でクロックレベルで比較照合がなされており、エラーが検知されるとそのモジュールは即バスへの出力を遮断される。しかし、対になっているもう1個のモジュールが同じ出力をバスへ出し続けているからシステム全体では何らの中断もない。LSIが発達し、CPUが1個のチップになってしまったメリットを最も贅沢に反映させたシステム形態である。

1.2.3 今後のフォールトトレラントシステム

ネットワークに接続した複数のサーバが、物理的には分散した姿のままで、システム的には相補的にカバーし合って全体としてフォールトトレラントシステムとして動く形を目指し、研究開発が行なわれている[16][17][18][19]。真のフォールトトレラントまで行きつくにはまだ遠く、さしあたりHigh Abailabilityということを追求している。本論文の第5章では、筆者等が行ないつつあるアプローチを論ずる。

1.3 研究の特徴

本研究で主たる研究対象としている技術分野、システム、および研究全体の前提条件、考え方等について述べる。

1.3.1 フォールトトレラントシステムをどう考えるか

フォールトトレラントシステムは、システム全体の信頼度を大きく向上させるための一つのアプローチである。別のアプローチもある。例えば、部品を徹底的に高信頼化する方法もある。システム設計者の立場に立てば、どれか一つを選択するのではなく、その時代ごとの技術レベルとコストとの按分によりトータルでの最適解を求めて、ウェイトバランスを考える。フォールトトレラントシステムは、常にかかるトレードオフの中で検討され、考察され、選択されてきた。従って、実用に供されたシステムは、いづれもその時代におけるコストバランスの結果であり、理論的に100%完璧な産物ではなく、至るところに妥協のある産物である。

フォールトトレラントシステムは、その名称の由来どおり、内部に故障が発生してもシステム全体としては、その故障に耐えて正常な運転を続行しうるシステムである。発想としては、高信頼度システムを実現するに際し、「部品レベルの高信頼化はそこそこにあきらめ、部品に故障はつきものだから、これをシステム技術で救う。」という考え方である。現実に高信頼度運転を命題とするシステムの構築を設計する立

場になって考えてみると、「部品に故障はつきもの」というあたりをどの程度のニュアンスで考えるかが重要なトレードオフのポイントになる。本論文で論じている研究の基本スタンスは、現実の諸経験を踏まえ、

- まず部品の固有信頼度向上があり、それを補完する意味でフォールトトレラント技術を考える。

とのスタンスである。

換言すれば、フォールトトレラントシステム構築技術は、

- 固有信頼度の低いユニットをシステムとして救うための技術

ではなく

- 良好な固有信頼度のユニットを前提とし、システムとして更に完全を期す技術

である。

従って、以下に述べる中で、フォールトの発生について、

- 単一障害である。
- 低頻度である。

ことを前提として論じている。

理論的には、複数障害が同時に発生する確率も零ではないので、これを回避対象としていないフォールトトレラント技術は、理論的に完璧な１００％稼動を保証する技術ではない。フォールトトレラント技術は確率論的に十分実用可能な高信頼稼動を目指す技術である。

1.3.2 研究対象システム

フォールトトレラントコンピュータシステムの実用上の形態は、

(1) オンライントランザクション処理

(2) リアルタイム制御

に大きく２分される。

コンピュータ中核部については両者に差異はないが、(1) はファイル装置の障害時の記録内容の一貫性、保全性に最大の注意を払っており、(2) は現場機器とそのインタフェースコントローラの障害時のシステム安全性に最大の注意を払っている。

本研究は、後者のリアルタイム制御システムを対象としている。リアルタイム制御システムおよびリアルタイム制御コンピュータの一般的特徴を以下に示す。

(1) システムの制御対象は主として物理的、化学的現象である。

(2) 制御システムの出力は人手を介さず制御対象に直結している。

(3) 制御対象の状態をセンスし、それに対応する制御出力をアウトプットする。

(4) システム不具合時にはシステムの安全性が最優先される。

(5) 連続運転に対する要求が高い。

(6) 予定外のシステム停止に対する許容時間巾は極めて短い。

(7) 業務の種類が固定しており、長期間変わらない。

(8) 運転員は訓練を受け、固定される。

なお、本研究で対象としているシステムは、以上の特徴の他に

(9) システム不具合時には、修理員による点検、修理が可能である。

という地上系システムを念頭においている。

1.3.3 研究対象技術

本研究は、前節に述べたリアルタイム制御システムを対象とし、その高信頼度化を達成するためのフォールトトレラント構築上のシステム技術を対象とし論じてい

る。具体的には、フォールトの検知、フォールトの回避をシステムとして実現するための技術を対象とする。その内容については、第２章以降で論ずる。

1.3.1 で述べたように、高信頼度システムを実現するアプローチとしては、システムレベルでのフォールトトレラントシステム技術が唯一のものではない。部品の信頼度を追求するのも一つのアプローチであり、部品のスクリーニング、部品のテスト、部品の耐環境設計、部品／回路レベルでのフォールトトレラント、等々の技術が関連する。本研究は、良好な固有信頼度部品を使うことを前提としているものの、かかる部品、回路レベルの技術は研究の対象外としている。

フォールトの対象として、どういう類のフォールトを考えるか。これは現実のフォールトトレラントシステムの開発目標を検討するときには、システムの任務の分析と並んで最初に決めねばならない重要な課題である。そのシステムでよく起こる恐れのあるフォールト、そのシステムにとって致命傷となるフォールトについては、重要な対象とせざるを得ない。本研究では、1.3.2 で述べたように、リアルタイム制御システムを対象としているので、フォールトの対象もそれに沿って考えている。焦点は、ソフトウェアバグをどう考えるかにある。1.2.1 で概観した如く、我々の身辺にある業務システムやＯＡシステムではハードウェアフォールトよりも、ソフトウェアバグによるダウンの方が多いという実感がある。とりわけ、システムの使用形態に変更や追加が多い場合や不特定ユーザが参加する場合、或いはオープンシステムで新しい市販ソフトウェアが頻繁に投入される場合、等々ではいつまでたってもソフトウェアバグが収束しない。しかし、一般にリアルタイム制御システムにおいては、システムの任務と運転形態は長期間固定され、特定の運転員がきめられた手順で操作を行い、本稼動開始前に十分な試使用運転期間をとるのでソフトウェアバグは、比較的初期フェーズに収束する。

従って、本研究では、ソフトウェアバグについては試使用運転中に収束を加速する技術について第３章で論ずるが、稼働中のフォールトとしてはハードウェア故障を主たる対象とし、それに準じる形で考える。

現実には、リアルタイム制御システムといえども、稼働開始後にもソフトウェアバグが残存する危険は内包しているので、ソフトウェアバグに対しても無防備であってよいわけはなく、一通りの対策は持つべきであり、本研究でもそのレベルでの考察は行なっている。

1.3.4　実システム適用事例

本研究では、実際にリアルタイム制御システムのフォールトトレラント化のために適用し、長期間の稼働実績を残した技術を事例として交えつつ論ずる。事例に取り上げるのは、次の２件である。

（1）　実際に大規模リアルタイム制御系として実用され、１４年間稼動して効果が実証されたシステム構築技術を論ずる。

筆者がメーカ側の主任設計者として開発し、納入した国鉄東北線郡山操車場の自動化システム“ＹＡＣ”(Yard Automatic Control system) は、昭和４３年１０月のダイヤ大改正と共に実稼動を開始し、昭和５７年９月まで１４年間連続して１日の休みもなく稼動したシステムである。（昭和４５年科学技術庁長官賞をプロジェクトマネージャ故鳩村和也が受賞）。

このシステムは、クリティカルリアルタイム制御系として、国内で始めて、実際に二重系コンピュータシステムによる制御を行い、実際に高信頼稼動を記録した。

（2）　約２００台の出荷実績をもつリアルタイムコンピュータ MELCOM350 -30 シリーズのＣＰＵにおいて採用し、有効であった技術を論ずる。

筆者が主任設計者として開発したリアルタイムコンピュータ MELCOM350-30(昭和 41 年）及び、その後継機 350-30F （昭和 45 年）はそのＣＰＵ内に当時としては、先端をゆくエラー自動記録メカニズムを持っていた。これは昭和４０年に輸入された IBM360 でも実施していることは判っていたが、350-30 は実現方法も含めて発表した。本論文では、350-30 で考案し採用したエラーの即時記録メカニズムについて述べ、その後、実稼動記録が残されている 350-30F についての効果を評価する。

1.4 論文の構成

本論文は、全7章と付録からなる。

まず、本章（序論）では、背景となるフォールトトレラントシステムのニーズと過去の主な事例および研究対象を述べた。

第2章においては、フォールトトレラントシステムを構築するために必要となる技術と、その考え方、および解決すべき課題を提示する。

第3章においては、リアルタイムコンピュータシステムにおけるエラー検知、エラーステータス記録、リトライについて特徴を明らかにし、実際に稼働したリアルタイムコンピュータシステムにおいて得られた成果を分析する。

リアルタイムコンピュータシステムにおけるエラー検知方式としては、一般に採用されているメモリ、CPU、バス、等におけるエラーチェックの他に、巨大な量的ウェイトを占めるプロセスIO部分と現場機器のエラーチェックが問題であり、これらを包括的にソフトウェアによりチェックすることが有効であることを示す。具体例として、筆者が開発した郡山YACシステムにおけるプロセスIO妥当性チェック、及びプロセスIOレスポンスチェックに対する適用結果から、以下の特長が得られることを示す。

（1） コンピュータのフォールト、及び現場機器のフォールトの両方を包括的にチェックできる。

（2） 余分なハードウェアを必要としないため、経済的に実現できる。

（3） 単一系内で実現できる。

また、以下のような弱点も明らかにする。

（1） 障害部位を特定する位置分解能が低い。

（2） 障害検知の時間分解能が粗い。

しかし、実際のプロセスIOの信号ルートのシンプルさや、制御対象の時定数との相対関係から、実用的には障害にならないことを示し、一般のリアルタイム制御システムに有効であることを示す。

リアルタイムコンピュータにおけるエラーステータス記録は、フォールトトレラント構築上、フォールトユニットの切離しのための情報として必須であるとともに、瞬時故障を起こし、その後潜ってしまった潜在故障部位の推定のために有効であることを示す。瞬時故障の部位の推定には、エラー時点での即時記録が必要であり、その経済的な実現方式として、MELCOM-350/30で筆者が開発したエラー記録機構を示す。実績として同じく筆者が開発したMELCOM-350/30Fの4システムの11年間の稼働データとボード交換データとから、潜在故障の推定が有効に働いていることを示す。

リアルタイムコンピュータシステムのリトライについては、一般的な瞬時故障対策の他に、現場機器やリレー接点の動作不良に対しても実際的な対策として有効であることを示す。

第4章においては、フォールトユニットの切離しと、冗長ユニットによる引継ぎについて述べる。また、連続運転を要求されるリアルタイムコンピュータシステムでは、プロセスIOユニットの修理後の再投入にあたって、最終動作確認のために、プラントシミュレータなどのテスト装置を備えておかねばならないケースがあることを示す。

フォールトユニットの切離しについては、フォールトを起こしていない別装置により切離し制御を行なうこと、論理回路上の切離しだけでなく、電気的に切離す必要を述べる。また、切離すユニットの大きさについては、コンピュータ実装方式の時代的相違、及び標準製品流用か専用製品かの設計スタンスの相違によることを示す。

冗長ユニットによる引継ぎについては、コンピューター式を単位とするときは、立上げ時間を要するため、リアルタイムシステムでは引継ぎに先だって運転準備しつ

つ待機しておらねばならぬことを示す。

具体的な事例として、筆者等が開発した郡山 YAC のホットスタンバイ系をとりあげ、そこでかかる二重系の構築技術として、単独系としての独立性、すなわち単独系内でのエラーの自己検知ができること、及び単独系内で修理後の最終動作確認ができることが必須技術であることを示す。

また、郡山 YAC の初期 8 カ月の稼働実績を示し、これが予期せざる送電系統の二重故障に見舞われたものの、この外部要因を除外してコンピュータシステムとして眺めれば、ほぼ期待通りの二重系の効果を発揮していることを示す。

ただし、現実の二重系では、或種のフォールト（例えば人間による誤操作）が起きると両系とも同時にダウンすることがあり、これが単純な二重系モデルの MTBF の理論値と現実値とが大きくかけ離れる原因となっている。そこで、かかる状態遷移が起こりうることを含めたマルコフ過程モデルで MTBF 理論値を算出すると、現実値とフィットすることを示す。

第 4 章後半部では、現在のトレンドに沿った新しいフォールトトレラントシステムを提案する。すなわちハードウェアとしては市販マイクロプロセッサ、オペレーティングシステムとしては市販オペレーティングシステムの環境下において、フォールトトレラントコンピュータのあるべき姿を論じ、一つの解を提案する。

この提案は、三重多数決方式をベースとし準正常状態という新しい概念の提起と、予防引継ぎ方式という、従来のホットスタンバイでもなくコールドスタンバイでもない、新しい緩やかな引継ぎ方式の提案とを含んでいる。

準正常状態は、あるユニットの内部的な健全性が若干低下した状態を定義づけたものであり、そのユニットは健全性が低下していることを外部に知らせるとともに、運転は全く正常通り続行する。具体的には、トリプル・モジュラ・リダンダンシ（TMR）のユニットにおいて、1 個のモジュールが故障し、2 個が正常に動いている状態が典型的な事例である。

予防引継ぎ方式は、TMR を単位とするマルチプロセッサ構成のシステム（タイトカップル、ルーズカップル、いづれでも可能）において、1 つの TMR プロセッサが準正常状態になったとき、正常運転を続行しつつも他のプロセッサに対し予防的に

信号を送り、自己の待ち行列テーブルにある業務を引きとってもらう方式である。

準正常状態の詳しい定義付けをおこない、予防引継ぎ方式の具体的な構成事例と動作方式とを提示し、この引継ぎ方式がオペレーティングシステムに動作中のタスクの引継ぎを要求しないために市販オペレーティングシステムを用いることが可能であることを示す。また、この予防引継ぎ方式によるマルチプロセッサフォールトトレラントコンピュータが、現存のフォールトトレラントコンピュータと比較して、信頼度、コスト、性能で優位にあることを示す。

第 5 章では、リアルタイムシステムも将来の姿としては、高信頼化分散処理システムの姿になると考え、分散システムにおける高信頼化へのアプローチを論ずる。すなわち、分散オブジェクト管理技術に基づく高信頼化について検討する。ここでは、分散環境でのオブジェクトの内部構成と管理方式について、従来の内部構成におけるサービスインタフェースの他に新たに管理インタフェースを持つことを提案している。管理インタフェースを持たせることにより、オブジェクトの静的な情報の管理だけでなく、動的な状態の管理も統一的に扱えることになる。

実際に、この内部構成をベースに、RODS という分散オブジェクト指向環境を構築した。この環境の特徴は、オブジェクトを位置透過、及び複製透過に管理することができる点であり、しかも管理インタフェースによって、オブジェクトの活性／非活性の状況、また負荷の状態を環境が管理することができる点である。

この RODS の環境を、実際にリアルタイム制御システムのプロトタイプである分散アプリケーションに適用し、クリティカルなリアルタイム性を要求されない部分については十分適用可能であることを検証する。しかし、リアルタイム性が高くクリティカルな応用については、まだオブジェクト管理のオーバヘッドが高いため、これからの課題となっている。

第 6 章では、本論文における成果を要約する。

第2章

フォールトトレラントシステム構築技術の概要

第 2 章

フォールトトレラントシステム構築技術の概要

2.1 はじめに

本章においては、フォールトトレラントシステムを構築するにあたって、何を考えねばならぬか、それを解決する技術はどういうものか、具体的な研究課題のポイントはどこにあるのかを示す。勿論、具体的な個々の実システムの構築にあたるときは、個々のシステム特有の条件について細かく分析する必要があるが、本章ではその中で一般的と考えられるもの、とりわけリアルタイム制御システムを対象としたときに一般的と考えられるものをとりあげて示す。

2.1.1 対象とするフォールトについて

システムが正常に運転しているということは、予め定められたサービスを量、質、精度、時間限度等、各面から見て満足に実施しているということである。システムが上記サービスを果たせなくなることを、システムフェイリャ（又はシステムダウン）と呼ぶこととする。システムフェイリャになった時、その起因となった何らかの原因が存在する。それをフォールト（又は障害）と呼ぶ。フォールトは、システムフェイリャの直前に発生する場合もあれば、長い間潜在していたのち顕在化する場合もある。システムフェイリャの原因となるものをすべてフォールトと考えると、フォールトには色々なタイプがあり、それらは 2.1.2 のように分類される。又、フォールトが発生したあと、システムフェイリャに至る過程は、2.1.3 のように分析される。

2.1.2 フォールトの分類

フォールトが稼働中のシステムで発生（又は顕在化）したとき、そのフォールトが発生した直接的一次的原因がある。その原因別にフォールトを分類すると下記の表2.1のようになる。

表 2.1: フォールトの原因別分類

	分類	具体例
1	ハードウェア故障	半導体素子の絶縁膜劣化 接続線の接触不良
2	ハードウェア動作裕度不足	電源電圧変動時誤動作 環境温度上昇時誤動作
3	ハードウェア論理不備	稀な動作条件時誤動作 （論理設計のバグ）
4	ソフトウェア論理不備	稀な操作環境時誤動作 （仕様設計の見落し プログラミングのバグ）
5	システム環境変動	落雷による電源電圧の変動
6	オペレータ誤操作	操作ボタンの押し間違い

フォールトが発生（又は顕在化）したあと、その持続時間によって表2.2のように区分けして考える。

これらのフォールトについては、発生原因に照らし、システム稼働期間には極力発生しないように、一般的な事前の予防策があり、表2.3に示す。

本研究では、表2.1のフォールトに対し、表2.3に示す一般的予防策および実際の経験をもとに次のように取り組む。

(1) ハードウェア故障

エージングは強くやり過ぎると破壊テストになってしまうので、その効果には

表 2.2: フォールトの接続時間分類

	分類	具体例
1	固定故障	半導体絶縁膜破壊 接続線の断線 磁気ヘッドのクラッシュ （不可逆的現象）
2	瞬時故障	エレクトロマイグレーション （ウィスカがショートすると 電流で溶断し復旧） 稀な動作条件でのみ顕在化するバグ

自ずと限界がある。又、稼働の途中から進行はじめる欠陥も多い。従って、ハードウェア故障はリアルタイム制御システムの稼働中のフォールトの主成分として扱う。

(2) ハードウェア動作裕度不足

このフォールトは事前のパラメトリックテストを慎重に行なえば、ほぼ完全に事前除去が可能である。よって、稼働開始後には大きな関心は払わない。

(3) ハードウェア論理不備

このフォールトは、事前のシステムテストの充実により、殆どの原因（バグ）を除去できる。一部は潜在したまま残留し、稼働中の特殊な動作組合せが重なったとき顕在化する。（システムテストカバレッジの洩れ）この場合、論理バグといえども現象的には瞬時ハードウェア故障の場合と同様に出現するので、それと同様に扱える。クロック同期で完璧に同じ動作、同じ環境の冗長系においては、冗長系にも同時に出現するが、クロック同期でないシステムレベルの冗長系では、マクロに同じ動作をしていても、ミクロには環境や走行タイミングがずれているので、同時に出現することはない。（ハードウェア故障と同じに扱える。）原因を究明し、バグを訂正してしまえば、そのあと再現することはないので、その点はハードウェア故障とは異なる。

表 2.3: フォールトの事前予防策

	分類	一般的な予防策
1	ハードウェア故障	素子レベル、ユニットレベルでの加速エージング（潜在欠陥を強制的に顕在化させる）
2	ハードウェア動作裕度不足	適当な裕度がとれる迄、設計パラメータを変更する
3	ハードウェア論理不備	論理動作テストスーツを充実せしめる
4	ソフトウェア論理不備	事前のシステム総合テストの充実（機能テストカバレッジ、組合せテストカバレッジ エンドユーザテスト、試使用テスト）
5	システム環境変動	環境シミュレーションテスト
6	オペレータ誤操作	操作統一化、簡便化、慣熟

(4) ソフトウェア論理不備

ハードウェア論理不備と類似の原因により、類似の現象を呈し、類似の対策を必要とする。ただし、ソフトウェアバグのほうが、一般に数が多く、かつ一部のものは潜在したままで、顕在化しにくい。従って、事前の除去に一層の工夫を要する。

潜在バグは何故潜在しているのか。バグが潜んでいる場所（アドレス）をプログラムが走行しないからである。或いは又、走行したとしても、その時の周囲の環境がバグを顕在化せしめる条件にセットされていないからである。（具体的には、その時のレジスタの値や、フラグの状況、直前に走った別のプログラムがどんなステータスを残しているか、或いは並行して走っている入出力チャネルのバストラフィックの状況、等々）しからばバグを積極的に顕在化させるためにはどうしたらよいか。初歩的な方法は、テストスーツを拡充し、全プログラムの全ブランチルートを隅なく辿ることである。（ルートカバレッジを100% にする。）しかし、現実には、これで見つかるバグは初歩的なバグである。しつこいバグは、複雑な条件を組み合わせてやらないと顕在化しない。しからば、どういう指標をもってテストカバレッジとすべきか。筆者は、ルートではなく、

全機能を全条件でテストすることである

と考える。ファンクションカバレッジと称するか、技術的にはコンビネーションカバレッジと称するべきか。これは、言うは易くて行なうは難く、天文学的なテストスーツを必要とすることであろう。これは、恐らく人間が直接書くことは不可能であり、コンピュータに自動発生させるしかない。

本研究で対象としているシステムはリアルタイム制御システムであり、その場合、一般にシステム総合動作可能となった後、実システムにて数カ月間の試使用運転を行なうのが普通である。この運転の目的は多々あるが、主目的はソフトウェアのバグの摘出、それも特殊なケースでの動作仕様見落し等の上流レベルのバグの摘出にある。この期間に上記テストスーツも実行可能である。かかる試使用運転を行なったあと、本稼働に入ると、リアルタイム制御システムの場合は任務がほぼ固定され、操作員も一定のメンバに固定され、不特定多数のユーザが直接システムにアクセスすることはないので、ソフトウェアバグの頻度は小さい。少数のしつこいバグが潜在するのみである。これらのバグが顕在化したときの現象は、大体において瞬時ハードウェア故障の場合に類似している。完全なクロックレベルの同期系でない限り、冗長系で同時に起こることもない。（タイミングのずれ）従って、ソフトウェアバグについても、ハードウェア瞬時故障の延長として取り扱ってゆく。リアルタイム制御用コンピュータとしては、むしろ事前のバグ除去を加速する機能をもつべきであり、第3章で論ずる。

(5) システム環境変動

システムが運転される環境下で起こりうる変動を洗いだし、事前にシミュレーションテストを行って弱点を修正するか、対応設備の増強を行なう。よって、本研究の対象フォールトとは考えない。

(6) オペレータ誤操作

オペレータは、システムの総合監視、操作を行なうので、一重系になっている例が多く、フォールトトレラント技術を導入することは困難である。すなわち、人間系も含めて二重系にすると混乱が起こり、マイナス効果も考えられる。一般にオペレータ誤操作に対しては、誤操作しにくいエルゴノミクス設計とか、2回操作によって有効化する等のマンマシンインタフェースの改善、或はトレーニング等により対処する。よって、本研究ではトレラント設計の主たる対象フォールトとは考えない。

現実には、数少ないが誤操作により二重系の両系ダウンも観測されているので、それがシステム信頼度に与える定量的評価は行なう。

2.1.3 フォールト発生後の現象

稼働中のリアルタイム制御システムの中枢コンピュータの中で、フォールトが発生したあと、一般にはどういう現象になるか。まず、ハードウェア固定故障のケースを図2.1に示す。

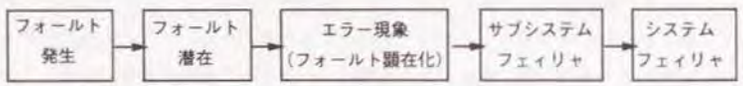


図 2.1: フォールト発生後の現象

次に、瞬時ハードウェア故障の場合、それがまだ潜在している間に自然復旧すれば、システムは何事もなく正常運転を続ける。しかし、コンピュータ動作が一旦何らかの影響を受ければ（即ちフォールトが顕在化しエラーが発生すれば）、そのあとは仮にフォールトそのものが自然復旧したとしても、コンピュータ動作にはどこかにエラーの痕跡が残り、一般的には時間の遅速はあるものの、サブシステムフェイリャへと進む。前節（3）に示したハードウェア論理バグの場合は、そのコンピュータが開発されたときからフォールト（バグ）は潜在しており、一般にそれらは事前に除去しそこなった深い潜在バグなので、たまたま運転中に顕在化する条件が揃った時点でコンピュータは不正な動きをする。この状況は、ハードウェア瞬時故障に同じである。前節（4）に示したソフトウェア論理バグの場合も、このソフトウェアが開発された時点からフォールト（バグ）は潜在しており、事前の除去活動をくぐり抜けた深い潜在バグである。よって、たまたま運転中に顕在化するステータス条件が揃った時点で、プログラムは設計者が予期しなかった動作を実行する。この場合、ハードウェアレベルの故障やバグではないので、ハードウェアレベルでのエラーではなく、ソフトウェアの意味レベルでの矛盾動作である。しかし、設計者が予期していない動作を行なう点では、瞬時故障の場合と同じである。この時点で、フォールト（バグ）は顕在化したことになる。そのあとの動きは、ハードウェア瞬時故障の場合に同じである。

2.2 システムフェイリャ

システムフェイリャは、一般論ではシステムが与えられた使命、或は定められたサービスを実行できなくなった状態である。システムが、独立性のある幾つかのサブシステムで構成されているときは、システムフェイリャになる前に、まずサブシステムのフェイリャが発生する。

具体的なシステムでは、個々に、そのシステムの使命の範囲やグレードを定義し、想定されるフォールトにより如何なる不具合状態が発生し、如何なる被害が発生しうるのかを分析した上で、システムフェイリャの定義をせねばならない。リアルタイム制御系では一般的にはフェイリャのグレードは下記の3レベルに分けて考えられる。

- 危険サイドフェイリャ（暴走型）
- 安全サイドフェイリャ（停止型）

- 部分的フェイリャ（サービス削減、フェイルソフト）

これら3レベルの分類は、現実には個々の具体的なシステムに依存するところが大である。例えば、地上交通システムでは、停止が安全サイドフェイリャであるが、航空機ではそうではない。本研究では、個々のシステムに順次立ち入っての追求はやらないので、フェイリャは一律にフェイリャとして扱っている。物理現象としてフェイリャが起こっていても、その持続時間がシステムの総合動作からみれば、ごく短時間で復旧すれば、マクロなシステム使命の観点からは許容しうる事例が多い。この種の狭義のフェイリャは真の意味でのフェイリャではない、として以下では論ずる。

2.3 フォールトトレラントシステム構築の一般的手順

フォールトトレラントシステムを構築する場合には、一般に次の手順を踏む。

（1）システム運転仕様の確定

このステップは、フォールトトレラントシステムに限らず、一般的なシステムでも構築のスタートポイントであり、以下の事項を明確化する。

- 対象システムの範囲
- システムの使命の範囲（ピークロード、拡張予備、互換性等）
- 運転条件　（タイムスケジュール、設置環境、保守条件等）
- 信頼性目標　（稼働率、ＭＴＢＦ、ＭＴＴＲ等）
- 評価方式　（シミュレーション、モデル解析計算等）

（2）フォールトトレラント仕様の明確化

フォールトトレラントシステムにおいては、上記の信頼性目標を次のように堀下げて分析し、明確化する必要がある。

- 対象とするフォールト，対象としないフォールトへのシステムとしての対処の考え方.
- 想定しうるシステムフェイリャと被害程度
- 稼働率、ＭＴＢＦ、ＭＴＴＲ、許容リペアタイム限度目標

（3）エラー検知の方式

対象とするフォールトの種類に対応した検知方式をコストとのトレードオフによって決める必要がある。主メモリ系、外部メモリ系、データ伝送系では、冗長ビットによる検知が常識的であるが、磁気ディスクにおいてはヘッドクラッシュ、データ伝送系ではデータ線以外の故障等をも当然フォールトの対象として考えるべきであり、これらに対しては冗長ビットよりもレスポンスチェックを対応させるべきである。

共通的な課題は、CPU のエラー検知方式である。1.2.1 で述べたように、過去に各種の事例があるが、要求性能、要求信頼度、許容コストとのトレードオフにより決めることとなる。

リアルタイム制御システムの場合は、これらの他に、プロセス IO と称する入出力インタフェース部がある。この部分は、制御対象プラントの現場機器との入出力制御／データ授受を行なう部分である。量的に巨大であると共に、エラーがプラント現場に直接的に波及するので、要求信頼度と許容コストとのトレードオフに一層の注意を要する。

（4）エラー記録の方式

固定的故障を対象にする限りは、エラー記録の必要性は小さい。実際にはハードウェア故障の8割以上は瞬時故障である [20]。この他に、ハードウェア論理、ソフトウェア論理の深い潜在バグが顕在化したときの現象も一瞬の特殊複合条件のときのみ出現し、あとは再現しないためにハードウェア瞬時故障と同様の振舞いをすることが殆んどである。かかるフォールトの場合、復元はするものの、そのままでは根本原因

が除去されないので、繰り返し発生する。これを防ぐためには、一瞬顕在化したのちに潜在してしまったフォールトを摘出修理せねばならない。これを行なうために、一瞬顕在化したときのエラーの状況ならびにコンピュータの走行状況を記録し、これを手掛かりとして原因を推定する必要がある。

リアルタイム制御システムでは、一般にシステムストップ許容時間が極めて短いので、システムの復元を優先する。従って、エラー検出時の迅速な記録が必須となる。又、2.1.2(4)に示したように、ソフトウェアバグに対しては、システム試使用段階における除去が非常に重要であり、そのためにもエラー記録は重要である。

（5）システムフェィリャ回避方式

冗長符号によりエラー検知した場合は、同時にエラー訂正（フェィリャ回避）が実施され、記録がそのあと行なわれる。CPUのフォールトでエラーを検知し、ステータスを記録したときは、そのあとシステムフェィリャ回避の処置へ移る。以下では、CPUフォールトの場合について考える。

このとき、システムはシステムフェィリャへ向かって進行途中の危険な状態にある。システムは、直ちにフェィリャ回避のための処置に移る。一般的には、次の手順を踏む。

1. リトライ

2. エラーサブシステムの切離し

3. 冗長サブシステムによる引継ぎ

リトライは、ハードウェアによるエラー記録、ＯＳによるエラー記録に引き続いて、同じくＯＳによる最短のチェックポイントへ戻ってステータスを設定し、再走行を試みるものである。ハードウェア瞬時故障、リレー接点や入出力機器の一時故障、ハードウェア論理／ソフトウェア論理の深い潜在バグが顕在化したときにリトライを試みると、２回目（或いはｎ回目）の走行時にはエラーが出現せず正常に動作することが多い。これが成功すると、システムとしては見かけ上は冗長符号によりエラーを

訂正したのと同様に、エラーをマスクしたことになる。考慮すべきポイントは、リアルタイム制御対象の時定数に比較して、リトライに要する時間が十分に短いかどうかの点と、エラーを検知したとき、常にリトライを行なうのか、それともエラーの種類によってはリトライを最初からバイパスするのかという点である。

エラーを検知し、リトライでも不成功となったサブシステムに対して、システム全体の視点からは、正常運転続行は不可能であると判断すると共に、一般にはシステム内に存在していると有害な出力を出す危険があると判断する。そこで、エラーサブシステムをシステムから切離す処置が必要となる。このとき、考慮すべきポイントは、当該サブシステムは、もはや故障状態に陥っているから、切離し動作の制御を任せられないということである。

切離しのあと、何らかの形で予め用意されていた冗長サブシステムが業務を引継ぐ。引継いで、即刻正常な制御出力を出せるのか、若干の準備遅延時間を要するのかについては冗長系をどういう方式で待機させておくのかに依存する。個々の具体的システムにおいては、制御対象の時定数に基づくプラント連続運転のための許容引継ぎ時間から待機方式を決めることとなる。表2.4に代表的な方式を示す。

リアルタイム制御システムにおいては、一般に表2.4の2.照合型ホットスタンバイと3.独立型コールドスタンバイは引継ぎ時間が許容時間を上回る恐れがあり、1.独立型ホットスタンバイ、又は4.多数決方式が選ばれる。独立型ホットスタンバイの場合は、サブシステムとして独立にエラー検知を行なう必要があり、注意を要する。この件は、具体例を交え、第３章にて論ずる。

（6）修理方式

冗長系を持つシステムの全体のMTBFは、故障した片系を如何に速く修理するかにより影響を受ける。（第４章にて論じる）従って、システム設計の段階において、単系のMTTRを短縮するための方式検討を行なう必要がある。修理に要する時間を分析すると、下記のように分類される。

1. 原因究明　　（含、保守員呼出し時間）

表 2.4: 代表的な待機方式

	方式名	待機引継ぎ方式	引継ぎ時間
1	独立型 ホットスタンバイ	常時、運転系、冗長系が同一運転を行なっている。出力は、運転系が与えている。固定故障検知の後冗長系が切り替わって出力を与える。	出力ラインのスイッチ時間のみ。
2	照合型 ホットスタンバイ	常時、運転系、冗長系は同一運転を行なう。出力は、照合比較チェックされてから与えられる。照合エラー検知し、固定故障と判定した後、各個に診断プログラムを走らせる。正常系を特定したあと、片系で再開する場合と、別に待機させてあった3台目の予備機を投入して照合型ホットスタンバイとして再開する場合とがある。	診断プログラムによる判断時間。 （片系再開） 又は、 診断プログラム＋予備機立上がり時間。 （照合系再開）
3	独立型 コールドスタンバイ	常時、冗長系は別業務運転をしているか、又は停止している。運転系が固定故障検知すると、切離され、替わって冗長系が該当業務の引継ぎのための準備に入り、準備完了後運転再開する。	冗長系の立上がり時間。
4	多数決方式	常時、3台が同一運転を行なう。出力は、多数決比較されて与えられる。1台の固定故障を検知すると、多数決による残り2台を正常と判断し、そのまま2台によって照合型スタンバイ運転へと移る。	零。

2. 部品交換、修理　（含、交換用品入手時間）

3. 正常動作確認、および運転準備立上げ

本研究で対象としている地上設置型のリアルタイム制御システムでは、一般に保守員常駐、交換用品常備であり、具体システム設計時に検討すべきポイントは、原因究明のため予め備えるべき手段（オンラインエラーステータス記録方式、オフライン診断方式）、交換ユニットの単位、常備すべきユニットの量、正常動作の確認のための手段である。とりわけ、リアルタイム制御システム特有のポイントは、ハードウェアの量的に膨大なプロセスＩＯ部分の故障時の原因究明と修理後の正常動作確認を時間効率良く行なう手段である。本件は、第4章にて論ずる。

システム設計時点にて検討しておくべき修理方式のもう一つ別の側面は、予防保守という点である。予防保守は、更に次の2つに大別される。

1. 定期的予防保守

2. 瞬時故障の記録に基づく推定保守

定期的予防保守は、一般の計算機システムにおいても大規模なシステムでは古くから実施されているものである。計画的に一定インタバル（1回／日～1回／週）にて一定の保守時間（30分～1時間）を設定し、CPUの健康診断、ＩＯ機器類の清掃点検、注油等を行なう。リアルタイム制御システムでは、年中無休連続運転のものがあり、この場合は、オンライン動作中のアイドルタイムにパトロール的に診断プログラムを走らせる。この種の予防保守の目的は、成長しつつある半欠陥を完全欠陥になる前に検知しようということである。そのためには、適度の加速試験を行なう必要があり、電圧変動の下で診断プログラムを走らせるのが一つの例である。

瞬時故障の記録に基づく推定保守は、一般の計算機システムでは定期的予防保守ほどには認知されていない。しかし、リアルタイム制御システム、とりわけ年中無休の電力プラントシステムにおいては、非常に有効な手段である。これを行なうには、設計時点において、エラーステータス記録に不足がないように配慮する必要がある。

本件は、第3章にて具体事例を交えて論ずる。

具体的なシステムにおいては、当然ながらこの他に全体系がフェイリャになった場合の緊急処置、フェイルセイフ設計、全体系の修理方式（単系と必ずしも同じではない）を決めねばならない。ただし、これら全体系の問題は、個々の具体システムの特性に大きく依存するので本研究の対象外とする。

（7）評価

以上の諸項目を検討した上で、システム全体として目標とする信頼度を達成する見込みがあるのか否か、設計の締めくくりとして定量的評価をする必要がある。そのための評価技術がフォールトトレラントシステム構築のための一つの要素技術である。評価の方法は以下の3つがある。

1. 解析的手法

2. シミュレーション

3. 実験的評価

設計段階で、一般的に使われるのは解析的手法である。コンピュータシステムの構成は、本節（5）の表2.4で示した冗長系の待機引継ぎ方式のどれを選ぶかによりほぼ決まる。そこで、システム全体の中で直列ユニット、並列ユニットの構成が判るので、これをもとに基本的な運用モデルを作ることができる。又、リアルタイム制御システムの稼働フェーズは、事前の試使用フェーズのあとであるから、故障率対経過時間のバスタブカーブの底にあたる期間、すなわち故障率λ＝一定値の偶発故障期を仮定してよい。すなわち、信頼度関数は、指数関数 $R(t) = e^{-\lambda t}$ を数学モデルとして仮定し、これをベースとして、各種の冗長構成時の信頼度やMTBFを近似計算することができる。リアルタイム制御システムは、一般に修理、再投入を前提とするシステムであり、この場合のMTBF算出にはマルコフ過程モデルを用いるのが便利である。注意すべきポイントは、リアルタイム制御システムで代表的な独立型ホットスタンバイシステムにおいても、或種のフォールトの場合は、即システムフェイ

リャとなってしまう例が存在することである。かかる点に、設計時点より注意を払い、マルコフ遷移モデルとしては、一挙に両系フェイリャとなる遷移を考慮しておくべきである。本件は、カバレッジコンセプトとしても提案されており[21]、第4章で具体事例を交えて論ずる。

シミュレーションと実験的評価は、大規模なシステムで、より正確な評価を得たい場合に使用されている。本研究では、これらの評価手法そのものは研究の対象とはしていない。ただし、第4章と第5章において、具体的な事例を解析的手法により評価している。

2.4 解決すべき課題

前節までにリアルタイム制御システムにおいて、フォールトトレラントなシステム構築を行なうにあたり、一般的に考えねばならぬ事項とそれを解決するためのシステム技術を概観し、解決すべき課題を論じてきたが、ここでそれらをとりまとめて示すと次のようになる。なお、個々の具体システムで個々に解決すべき課題は除く。

(1) エラー検知

- 主対象とするフォールトの分析と範囲
- CPUとプロセスIOのエラー検知方式
- 二重照合チェックする場合はコスト負担、正常系の特定方式、一重系運転を行なうか否か、またその時のエラー検知方式
- 単独系で独立チェックする場合はその検知方式

(2) エラー記録

- エラー検知時点でのCPUステータスの正確な記録の手段

(3) リトライ

- リトライ有効範囲の確認、実効性の確認

(4) 切離し

- フォールトユニットの確実な切離し
- 切離し、引継ぎの大きさの単位

(5) 引継ぎ

- 許容しうる引継ぎ時間内でコスト、性能上ベストな方式
- 有効性の検証、評価
- 両系同時フェイリャ現象の特定と対策

第3章、第4章で順次これらの課題について、具体事例をあわせて論ずる。

第3章

リアルタイムコンピュータシステムにおける エラー検知、記録、リトライ方式

第 3 章

リアルタイムコンピュータシステムにおけるエラー検知、記録、リトライ方式

3.1 はじめに

動作中のリアルタイムコンピュータシステムにおいて、何らかのフォールト（障害）が発生すると、まず部分的に予定外の動作を惹起し、引き続きシステム全体が予定外の動作を行う。従って、部分的な予定外動作をエラー現象と認識し、検知することが、フォールトトレラントシステム構築の第一歩であり、最も基本的な技術である。このとき、フォールトが発生してからのち、極力短時間内に、理想的にはワンクロック内に、エラーとして検知することが望ましい。どれだけ短時間に検知できるかは、時間分解能と呼ばれる。又、エラーを検知したとき、その波及範囲が狭く限定されている方が望ましい。どれだけ限定した状態で検知できるかは、位置分解能と呼ばれる。

時間分解能の許容限界は、当然ながら全体システムが不具合動作を起す前に検知し、システムとして防止処置をとることができる程度の短さ、ということになる。

位置分解能については、障害部位の特定、切離し、交換、のための情報を提供するものであり、システム全体の信頼性と保守コストに大きな影響を及ぼす。しかし一般には、エラーの検知メカニズムは、エラー現象に注目して検知しているので、二次的にこれを位置情報に置換する。

以下、本章では、エラー検知、記録、リトライについて、リアルタイム制御システムとして現場機器の誤動作防止に焦点を合わせた観点から論ずる。

3.2 エラー検知

システムフェイリャをひき起こす原因となる事象をすべてフォールトと考えると、第2章表2.1に示したように、ソフトウェア論理不備（バグ）やオペレータ誤操作等を含め6種類のものがある。2.1.2節で論じたように、これらのうちでリアルタイム制御システムの実稼働中にも或頻度で起きると考えねばならぬフォールトは次の4つである。

(1) ハードウェア故障

(2) ハードウェアの論理上の深い潜在バグ

(3) ソフトウェアの論理上の深い潜在バグ

(4) オペレータ誤操作

但し、（4）オペレータ誤操作については、関心を払いつつも、対策については対象外とする。すなわち、一般的には人間の判断による操作介入を最高プライオリティとし、システム暴走を止める最後の手段の一つにしているシステムが大多数である中で、ロウレベルにあるコンピュータシステム側がハイレベルにある人間に対して、如何に誤操作であることを検知するのか、検知しようとすること自体が矛盾ではないのかと考えられる。この問題は知識工学の面、人間工学の面からのアプローチが必要である。具体的なシステム構築の場では、各個別システムごとに、マンマシンインタフェースの改良とか、オフラインでの訓練とか、運転員の勤務体系とかを検討することとなる。

本節では、以下で（1）ハードウェア故障、（2）ハードウェアの深いバグ、（3）ソフトウェアの深いバグの3種類のフォールトを対象とし論ずる。

3.2.1 ハードウェア故障

CPU内でハードウェア故障がどこか1箇所で発生したものとする。故障としては、例えばトランジスタ素子の絶縁膜破壊とか、エレクトロマイグレーションによる

アルミニウム配線層の断裂とかである。これらは、故障の物理的な現象であり、これをコンピュータハードウェアとして眺めると、どこかのゲート或いはフリップフロップが“1”又は“0”に不正に膠着（スタック）する現象となる。CPUが走行している途中で、かかる故障が発生すると、どこかでCPUの動作は意図された動作と異なる動作をする。この異常動作を本研究ではエラーと呼ぶ。一般に物理的な故障が発生した瞬間から、CPUがエラーを起こすまでに若干の時間がある。この時間、故障は潜在している。潜在時間の長短は、CPU動作走行中にその故障箇所がいつCPU動作に関与するかによって決まる。ハードウェア故障の場合の潜在時間は、長かろうと短かろうとシステムとしてはあまり問題にならない。それがエラーを起こさなければ、システムとしては何も知り得ないし、問題も起こらない。故障がCPUに対して、何らかの論理動作上の予定外の影響を与えた瞬間からあとが問題となる。

物理的な故障が論理的な異常動作（エラー）となってCPUに影響を与えたとき、結果としてCPUは如何なるエラーを起こすのか。エラーがまずどこに出現し、どう波及するかの個々の振舞いは、故障が起こった位置と、そのときCPU上で走行しているプログラムの組合せによって決まる。結果的には、天文学的な組合せ数の振舞いがありうるが、これを極めて大雑把に分析し、図3.1に示す。

現象としては、矢印に沿って大まかには上から下へ落ちてくるが、その速度は非常にバラツキがある。速い場合はクロックレベルのオーダで次々と進行するが、遅い場合は、例えば稀にしか影響を与えないレジスタの特定の1ビットが反転したようなときを考えると、プログラムがそのレジスタに関与するまではエラーの波及は停止する。

3.2.2 ハードウェア故障時のエラー検知

エラー検知の目的は、図3.1の最下部に示すシステム動作異常となる前に、CPUが異常状態になったことを知ることにある。従って、検知の機能は時間的制約を受けている。又、CPUでは、故障の場所とプログラムの組合せによりエラーの起こり方が極めて多様であるが、検知の機能としては、これらを洩れなく捕捉することを期待される。

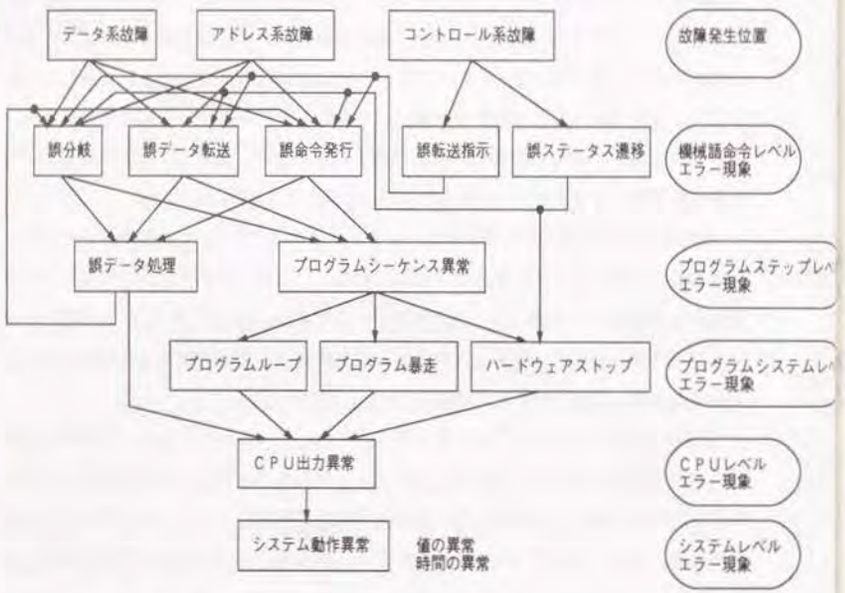


図 3.1: ハードウェア故障からシステム出力異常への波及フロー

時間的制約を考えると、エラーが出現したら極力早い段階で検知することが望ましい。これは、図3.1の上部、すなわち機械語命令レベルでエラー検知網を張ることを意味する。この場合は、算術演算器、論理演算器、各レジスタ、カウンタ、コーダ／デコーダ、状態遷移制御回路網等の個々に二重化し、比較照合チェックを行なう必要がある。何故なら、個々の小さなユニットごとに個々独立に論理動作を行なうため、メモリ系や伝送系の如く、同じルールで一定の冗長符合をチェックすることが成立しないからである。このレベルで、二重化してゆくことは、ハードウェアの量の面では約3倍に増加し（二重化で約2倍、照合回路を含めて3倍）、速度の面では単純に設計すると約半分に低下する（個々に比較に要する時間だけクロックを長くする）。更に、単体の故障率が3倍に増大する。これらのマイナス要因が大きいため、現実にはこの方式は採用されていない。

CPU のエラー検知として広く実用されている代表的な方式は、３０年前も現在もCPU 全体を一括して二重化（又は三重化）し、その出力信号だけを照合チェックする方式である。この場合、図3.1の下部すなわち CPU レベルでのチェックとなる。この場合は、システム動作異常へとつながる直前に位置するから時間的には余裕が少ない。但し、内部の機械語レベルで細かく二重化照合する場合に比べると、最後の出力信号のみに絞った照合であるから、照合のためのハードウェア量も、所要時間も大幅に減少し、全体としてはハードウェア量で約2倍（二重化で2倍、照合部僅少）、速度ではほぼ低下なし（一旦フリップフロップで受けて照合するから、次のクロスステップ内におさまればよい）である。

3.2.3 ハードウェアバグに起因するエラーの検知

殆どのハードウェア論理のバグは、システム検証時、および試使用運転時に摘出されるので、現実に稼働中に顕在化するハードウェアバグは、きわめて深いレベルに潜在していたものである。かかるバグは、或種の走行環境条件が重なり合った希少なタイミングで顕在化し、誤ったアドレスをセットしたり、データを誤操作したりする。そのタイミングは瞬時に消滅し、次に再現するのは１カ月後であったり１年後であったりする。従って、現象としては、これはハードウェア瞬時故障と同じであり、

ハードウェア故障で論じたことをほぼ同様に適用してよい。但し、本質的な相違が一点ある。それは、バグの場合は設計時点からそのCPUに潜在しており、同じ設計で作られた他のCPUにも同じく潜在しているということである。

同じCPUを二重化して同じ条件で運転を行ない、出力の照合チェックを行なっている場合は、バグが顕在化する条件は両方のCPUに同等に発生する。従って、両方のCPUで同時に同じエラー現象が発生し、出力の照合ではエラーを検知しえない恐れがある。とりわけ、両方のCPUをクロックレベルで完全に同期運転している場合には、この危険が高い。他方、CPU単位でなく、磁気ディスク等の外部メモリまで含めたコンピュータサブシステムとしての単位で二重系運転している場合は、両系の走行のタイミングは磁気ディスクの回転のずれ（同期回転していないものとして）のため、数ミリ秒のオーダでずれており、これに起因して、片系でバグを顕在化せしめる希少な条件が発生しても、他系では発生しない場合が殆どである。従って、この場合は同じバグが潜在しながらも、エラーが起こるのは片系だけであり、照合チェックが有効に働く。

3.2.4 ソフトウェアバグに起因するエラーの検知

ハードウェアバグの場合と同じく、殆どのソフトウェアバグはシステム検証時、および試使用運転時に摘出される。よって、稼働中に顕在化するバグは、極めて深いレベルに潜在していたものである。ここまでは、ハードウェアバグと同じである。ただし、ソフトウェアバグの方が一般に１０倍以上多い。

稼働中に、事前テストでは起こらなかった希少な環境条件が生起したとき、ソフトウェアバグが顕在化し、システム設計者或いはプログラマが意図した動作とは異なる動作を行なう。このとき、ハードウェアバグと異なる点は、ハードウェアバグが機械語命令とかキャッシュメモリ制御とかのレベルで異常動作、矛盾動作、不正動作を行なうのに対し、ソフトウェアバグの場合は、このレベルの異常動作ではなく、プログラムアルゴリズム或いはプログラムの文脈の流れの上で、設計者が予期していなかったフラグを立てたり、逆方向へブランチしたりする。

ハードウェア故障やハードウェアバグの場合は、CPUの基本動作のレベルで異常

を起こすが、ソフトウェアバグの場合は、個々の機械語命令は正常に動作しているので、相対的に、エラー現象が波及しCPU異常に至る速度が遅く、ばらつきが大きい。

ソフトウェアバグは、設計仕様を定義した時点やプログラムを作成した時点で潜入したものであり、同じソフトウェアが乗っている別のCPUでも同様に潜入し潜在している。従って、二重系による比較照合チェックを行なっていても、両方の系に同様に発生するので、検知しえない恐れが強い。ハードウェアバグの場合は、ハードウェア論理の本質から、CPU走行時の論理面、タイミング面での希少条件時に顕在化することが多く、両系同時走行していてもディスクのタイミングのずれがあることにより、片系でしか条件が生起しないことが殆どであるが、ソフトウェアバグの場合は、かかるタイミング条件よりもイベントの生起の数とか、順序とか、データ量の多寡とかの組合せが顕在化のトリガになることが多い。従って、ハードウェアバグよりも、両系揃って生起する危険が大きい。

3.2.5 リアルタイム制御システムにおけるエラー検知

前節までに、エラー検知を一般的に論じた。とりわけCPUのハードウェア故障に対しては、古くから変わらずCPU一括しての比較照合チェックが有効なるも、コスト負担が大きいこと、ハードウェアバグに対しては、照合単位をもっと大きくして、コンピュータサブシステムとして比較照合チェックが有効なるも、同じコスト負担が大なること、ソフトウェアバグに対しては、これら照合チェックも有効性が薄く、ここまでの範囲では専ら事前除去の努力しか対策がないことが要旨である。

さて、本研究で対象としているリアルタイム制御システムに焦点を絞ってエラー検知の問題を更にほり下げる。

リアルタイム制御システムでは、エラー検知したあと、次章にて論ずる冗長系への引継ぎのための時間を長くとることは許されない。この視点からすると、単なる二重照合チェックは、リアルタイムシステムには不適である。何故ならば、次の２つの動作のため、運転再開に時間を要するからである。

(1) 照合チェックによりエラーを検知したあと、いづれが正しい出力を出しているのか、改めて判定せねばならない。一般的な手法は、各個別のCPU（又はコンピュータ）において、診断プログラムを一通り走らせ正常終了した側を正常と認定する。

(2) 正常系を認定したあと、運転再開するに際し、単一系で再開したのでは照合相手がないからエラー検知ができない。従って、待機させてある第三のスペア系を立上げる必要がある。

以上の結果をまとめると、リアルタイム制御システムにとっては、一般には有効とされる二重照合チェック方式は引継ぎ時間を要する点で不適であり、又ソフトウェアバグへの無力の点でも不適である。これらの点を解決するため、筆者は次のエラー検知方式を考案し、実用した。

3.2.6 ソフトウェアを用いてのエラー検知

上記の欠点を回避する手段として、筆者は次の2つのアプローチをとった。

(1) エラー検知のために相手系を必要とせず、単独自己系の中で自己のエラー検知を行なう方式を探す。

これができれば、ホットスタンバイ二重系において、エラー検知後、直ちにホットスタンバイ系へ切替え、その系は単独系として運転を引継ぐ。

(2) ソフトウェアバグ検知を行なうには、或種の知識処理的な手法を探す。

リアルタイム制御システムでは、リアルタイム制御という時間制約があるが、１０ミリ秒のオーダの余裕は一般にあるので、かなりの処理判断を試みる時間はある。また、制御対象が物理的、化学的な実現象であり、運転パターンが或程度限られている点を知識として使える可能性がある。

以上の2つのアプローチ方針から導かれる方法は、単独系内でのソフトウェアを用いての自己エラー検知方式である。

ソフトウェアを用いてのエラー検知方式のやり方としては幾つかの方式があり、分類すると次の3つになる。

(1) パトロール方式

リアルタイム制御システムでは、必ず一定時間間隔で行なう計測や制御がある。これらの業務と同様に一定間隔のタイマ信号をトリガとして、比較的短時間に終了するCPU自己診断プログラムを走らせる。巡査が受け持ちエリアを一定時間毎に巡回パトロールするとの類推で、パトロール方式と呼ばれている。ハードウェア固定故障に有効である。診断プログラムはCPUの大部分をカバーすることができて、しかもコストアップがない。しかし、ハードウェア瞬時故障に対してはパトロール時には消滅しているため、検知することは期待できない。又、ハードウェアの深いバグ、ソフトウェアの深いバグについては、パトロールは単純な条件下でのテストであるためCPUハードウェアは正常に反応するので、やはり検知することは期待できない。

(2) ループチェック方式

リアルタイム制御システムでは、多数のプロセスIO出力信号がある。これらが現場機器に与えられ、制御対象に何らかの制御を行なう。他方、制御対象の状態をセンスするために多数のプロセスIO入力信号がある。これらは、一般に或レベルを越すとコンピュータの制御プログラムを起動し、出力信号が対応して出てくる。

これらの信号経路は、コンピュータと制御対象とを環流するループとなっている。これに着目し、プロセスIO出力信号をCPUが指示してのち、一定の時間を待って対応するプロセスIO入力信号をセンスし、CPUにとり入れたセンス信号が先に与えた出力信号に対し予期した反応を示しているか否かをチェックする。このチェックは、時系列的に複数の出力信号シーケンスと複数の入力信号シーケンスの対応をチェックすることも含む。或いは、或1個の出力信号を起点としてチェックするだけではなく、それよりもっと時間的に遡って別のリ

ンクを持つイベントを起点としてチェックしてもよい。これをループチェックと呼ぶ。

このチェックは、CPUのハードウェア固定故障のみならず、入出力チャネル、プロセスIO、ケーブル、現場機器を含む一連のループ上の全てのハードウェア固定故障をチェックできる。ハードウェア瞬時故障、ハードウェアバグ、ソフトウェアバグに対しては一般的な検知能力は殆どないが、そのバグに起因するエラー現象が該当ループに或閾値以上の影響を与えた時は検知することができる。リアルタイム制御システムにおいては、実用的にはこれで十分である。逆に言えば、或閾値以下のレベルでの細かなエラーを見逃しても、実用上支障にはならない。このチェックは、CPU全体がハングしたり、暴走したりしたときは検知信号自体も出せず沈黙してしまうが、かかる場合はウォッチドッグタイマに頼るのが通例である。なお、このループチェックは、元々存在している信号ラインを高度に使うことで実現するものであり、ハードウェアコストアップがない。

(3) パリディティチェック方式

リアルタイム制御システムのプロセス出力信号は、制御対象のプラントの状態をセンスして、それに何らかの対応をすべくCPUが出力指示するものである。極めて単純な対応を行なうものもあれば、CPU内部で高度な演算処理をした結果出力される信号もある。この場合の高度な演算とは、制御対象の振舞いのシミュレーション計算であることが多い。かかる場合に、CPU内に故障があったり、ハードウェアバグやソフトウェアバグがあって計算結果を狂わせる（値の面、時間の面）ことが起こりうる。かかる高度なシミュレーション計算の結果に対し、或種の信憑性、或いは妥当性のチェックを行なうことが考えられる。これをパリディティチェックと呼ぶ。

このチェックはシミュレーション計算とは全く別の簡単なアルゴリズム（例えば簡単な表をひく）を用い、プログラム上は別のルーチン、別のアドレス、別のタイミングでパリディティ上限値、下限値を生成し、シミュレーション値が

上限値、下限値の範囲内に入っているか否かをチェックする。このチェックは、CPUのハードウェア固定故障を検知しうる。又、ハードウェア瞬時故障、ハードウェアバグ、ソフトウェアバグに対し、一般的な検知能力は薄いが、それらに起因するエラー現象が該当出力値に或閾値以上の影響を与えたときは検知することができる。リアルタイム制御システムにおいては、実用的にはこれで十分である。システムにとってクリティカルな出力信号が全てかかる方式でチェックされておれば、システムは当面、おおよそ正常に運転することができる。このチェックにかからない細かなレベルのエラーは、見逃したところで直ちには実害にならない。それらのエラーは、いづれは累積して大きなレベルになったとき、或いはフォールトが持続するようになってパトロールにかかった時点でチェックされる。

このチェックは、CPUがハングしたり、暴走したりした場合には、検知信号自体を発信できずに沈黙してしまう。かかる場合は、上記ループチェックと同じく、ウォッチドッグタイマチェックに頼ることとなる。なお、このパリディティチェックは、ハードウェアのコストアップなしに実現できる。

筆者がリアルタイム制御システムにおいて、極めて有効と考えるソフトウェアを用いての単独系内でのエラー検知方式をここで3種類提示した。最初のパトロール方式は広く使われている方式であるが、残りのループチェック方式とパリディティチェック方式は、筆者が１９６５年に開発開始し、１９６８年１０月に実働開始した国鉄郡山操車場YACシステムで初めてかかるコンセプトを提案し、実設計し、実用した方式である。

次にYACシステムにおける実施例を示す。

3.3 実用事例 国鉄（現JR東日本）郡山YACシステム

過去に開発され、稼動実績をあげたリアルタイムコンピュータ制御システムの一例として、本節で郡山ＹＡＣシステムの概要について述べ、引き続き後節で、ＹＡＣで開発したエラー検知技術、エラー記録技術、待機系への切替技術等について論じ

る。

3.3.1 ＹＡＣシステムの概要

郡山ＹＡＣシステムは、国鉄東北線郡山駅に隣接するハンプヤード式の貨車操車場の自動化システムである。昭和３９年３月に国鉄本社内に「ヤード近代化研究委員会」が設置され、昭和４０年４月より三菱電機を主契約者とする開発チームが発足、昭和４２年１０月より手動運用開始、昭和４３年４月より全自動試運用開始、昭和４３年１０月より運用開始、その後昭和５７年９月中旬まで同一コンピュータによる約１４年間の運用を行った[22]。

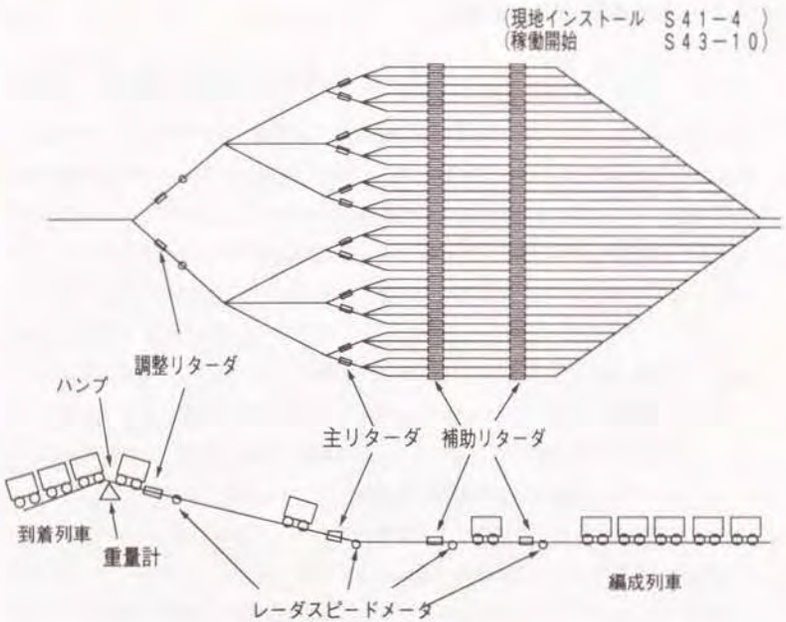
筆者は、昭和３９年１１月よりシステムエンジニアのチーフとして参画し、その後４０年４月から４３年１２月までハードウェア全体システムのリーダ兼マネジャーとして仕様打合せ、開発、製造、現地試験、そして保守の実務と管理を行った。

図3.2に郡山操車場の平面レイアウト図と垂直断面図とを簡略化して示す。

貨車操車場は、簡単に言うと、貨車の編成組立て変更を行うところである。そのためにソーターポケットとして、数百米に及ぶ長い待機線路が３６本並列に設置されている。操車場に到着した貨物列車は、ハンプと呼ぶ小高い丘の上へ押し上げられ、そこで（普通）一輌づつ切離される。切離された貨車は３０秒程度の間隔で次々に斜面を下り、惰力で転走する。ＹＡＣはレール各所に設置された車輌検知機（トレッドル）により貨車の通過を検知し、予め知らされている目標駅に対応して途中の分岐機（ポイント）を次々に左右に転換し、貨車を所定の線路へ導入する。

貨車は、待機線路に入ったあと、自走を続け、既に待機停止している貨車列に１～２ m/s の速度で追突し、自動連結する。１～２ m/s の速度で追突するようにＹＡＣは、斜面途中と待機線途中に設置されている減速装置（リターダ）に、車輌締めつけ／解放、の指示を与え、貨車の速度を減速する。リターダの前方にレーダスピードメータが設置してあり、ＹＡＣはこのメータにより、貨車の減速状況をリアルタイムに監視しつつ、解放タイミングを判断する。

図3.3は、郡山操車場を俯瞰した図である。図の中央にハンプがあり、その横にコントロールタワー兼コンピュータ屋舎がある。続く３つの図は、線路に沿って布設されている検知機器、制御機器である。



・貨物列車の編成組替えを行なう

・ハンプから一両づつ切離し、重力による散転

・行先別にポインとを切替え、所定の仕訳線へ

・編成列車後尾へ1．5m／s（±0．5）で追突連結

・追突速度の制御はリターダとレーダスピードメータ
（50msサンプリング）

・リターダ解放後の自走をシミュレーション
（重量、抵抗、他）

図 3.2: 郡山 YAC 概略図

図 3.3: 郡山操車場

図 3.4: 電磁トレッドル

図 3.5: レーダスピードメータ

図 3.6: リターダ

3.3.2 ＹＡＣにおけるエラー検知方式

ＹＡＣにおけるシステム運転仕様、システム信頼度目標、それらを実現するためのコンピュータシステムの構成等については、第４章で論ずる。本章では、エラー検知方式を中心に論じているので、それに関連する事柄だけに触れる。ＹＡＣのコンピュータシステムは図3.7に示すようにホットスタンバイ二重系になっている。そして、3.2節で論じた如くCPUに関しては照合によるエラー検知方式は採らず、片系づつ独立にエラー検知することを目指し、ソフトウェアによるチェックを中心とした。

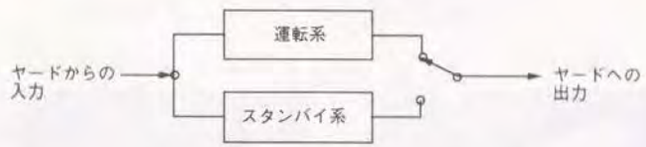


図 3.7: YAC コンピュータシステム基本構成

主メモリ、外部メモリ、バスに関しては当然パリティチェックを行なっているが、その他の電源チェック、ソフトウェア暴走／ハングのチェック（ウォッチドッグタイマ）により、ソフトウェアチェックでカバーしきれぬコンピュータ片系全体のフェイリャをカバーしている。表3.1にこれらを示す。

プロセスIOパリディティチェックとプロセスIOループチェックが、筆者が独立片系チェックのコンセプトに沿って考案したＹＡＣに特有のチェックであり、次節でその内容を説明する。他のチェック機能は、当時出現したIBM360に代表される当時の新型汎用コンピュータにも共通的に採用されている。

3.3.3 プロセスＩＯループチェック

リアルタイム制御システムにおいては、多数の現場（制御対象プラント）の機器とインタフェースを持ち、それらの制御を行う。インタフェース回路は、ゲート、レジスタ、ドライバ等の電子回路、コネクタ、リレー等の接点類、更にケーブル等から

表 3.1: YAC におけるエラー検知方式

	ハードウェアによる検知		ソフトウェアによる検知
ハードウェア フォールト	メモリ系	主メモリパリティチェック ドラムメモリ多重パリティチェック	
	バス系	バスパリティチェック プロトコル時間チェック	
	ＣＰＵ		パトロールチェック プロセスＩＯパリディティチェック
	全体包括	ＡＣ、ＤＣ電源レベルモニタ	プロセスＩＯループチェック
		ウォッチドッグタイマチェック	
ソフトウェア フォールト	メモリプロテクションチェック 不正命令チェック ウォッチドッグタイマチェック		プロセスＩＯパリディティチェック プロセスＩＯループチェック

成る。ここでは、これらをプロセスＩＯと称する。これらは、個々には比較的単純な構成ながら、多種類の素子が直列に連鎖状につながり、更にそれらが多数並列に使用され、総量としてはシステム構成の大半を占めることが多い。その上プロセスＩＯの障害は、現場機器の不具合動作に直結する。従って、フォールトトレラントシステムを構築する上では、これらのプロセスＩＯに障害が起った場合、いかにしてそれを検知するかが重大なテーマとなる。

図3.8にプロセスＩＯの出力信号の一連のつながりを簡単に示す。

図3.8に示すようにプロセスＩＯは単純な１ビット信号で出されることが多い。しかも、個々の信号は、独立で、タイミングもバラバラである。そこで、個々の１ビット信号に冗長コードを付けることが考えらるが、それは、経済的に非常に効率が悪い。又、どの範囲迄をチェックすべきかを考えると、コンピュータ単体としてはドライバ出力あたり迄をチェックすればよいかもしれないが、システム全体としては、現場器まで包括してチェックしたい。

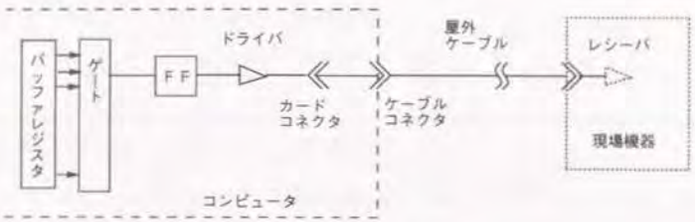


図 3.8: プロセス IO 出力信号の例

他方で、プロセスＩＯの信号のレートを時間軸から眺めると、コンピュータ CPU 信号よりも少なくとも３ケタ（1000 倍）は遅い。

プロセスＩＯに関するこのような特性からして、筆者は、上記の時間的余裕を活用し、ソフトウェアによる包括的チェック、すなわち、プロセスＩＯの出力信号に反応して現場の制御対象の変化状況を入力信号として読みとり、両者間の関係の正常性を調べることが最も経済的かつ洩れのないチェックになると考えた。リアルタイム制御システムでは元々、制御目的のために多数の入力信号を扱っているので、チェックのためだけに余計な入力信号を付加する必要はほとんどない。

郡山ＹＡＣで研究し実行した具体例を以下に２例示す。

まず、分岐機への出力に対するレスポンスをチェックする例を図 3.9に示す。

分岐機転換出力 → 分岐機作動 → 分岐機転換状態入力

図 3.9: 出力信号／入力信号のリンク

分岐機は、貨車を所定の行先に対応した所定の待機線路へ誘導するためのレール転轍機であり、コンピュータからの出力信号を受けて左右に転換する。コンピュータは各所に散在する車輌検知器（トレッドル）によって、貨車の現在位置を知り、前貨車の最終車軸が分岐ポイントを通過したあと、次貨車に備え、分岐機を新たな方向へ

と転換作動せしめる。分岐機は作動に 0.5 秒程度を要するため、コンピュータは分岐出力を出したあと、転換フィードバック信号が、許容時間内に戻ってくるか否かをチェックする。

上記レスポンスチェックは、ある出力信号と単純なペアを組む入力信号について説明したが、実際は更に広い範囲で適用可能である。

例えば、ハンプ頂上で切り離され、転走を開始した貨車が、第１のトレッドル（車軸通過検知器）に到達するまでに５秒程度かかるとすれば、その期待時刻に適当な前後巾を持たせて第１トレッドルからの信号入着チェックを行なう。

また、その貨車の前輪、後輪の車輌間隔データが判っているから、前輪のトレッドル信号の入着の後、例えば０．８秒程度あとに後輪のトレッドル信号が入着することを期待してチェックを行なう。

次に、図 3.10に、より高度なループチェックを示す。ＹＡＣは、貨車の最終追突速度を決められた範囲内にするため、貨車の転送の途中の速度を制御する。それにはリターダと称する線路に沿って設置された長いブレーキシューを作動させ、貨車の車輪を締め付けて速度を低下させる。貨車の速度を近くのレーダスピードメータで刻々と測定し、コンピュータに入力する。

すなわち、ある出力信号とある入力信号との間に１つのリンクが存在する。

リターダ出力 → 貨車の減速 → レーダスピードメータ信号入力

図 3.10: 出力信号／入力信号のリンク

コンピュータは広い操車場の中で、あちらこちらに自走している複数台の貨車（郡山ＹＡＣの場合、常時６台程度）を同時に監視し制御しているので、レーダスピードメータ入力についても常時１台を注視している訳ではなく、適当なサンプリング間隔で、全点をスキャニングしている。（郡山ＹＡＣでは５０ ms インタバル）。

ループチェックは、リターダ出力に対し、レーダスピードメータからの入力信号が、予測される範囲内の値に入っているか否かを、ソフトウェアによりチェックする

ものである。ＹＡＣではコンピュータは、貨車の最終追突速度をシミュレートしつつ、現在速度をどこまで低下せしめるかリアルタイムで予測制御するために、下記の情報を、あらかじめ、実験や、測定や、貨物列車通報によって知っている。

・該当貨車データ　（重量、進入速度、形状、行先、固有振動数、等）
・該当リターダ　（摩擦係数、ブレーキシュー長さ、等）
・誘導先線路データ　（滞留貨車までの距離、摩擦係数、等）
・現在の気象データ　（風力風向、雨量、気温、等）
・貨車運動方程式

コンピュータはこれらのデータや予備知識によって、現在出力しているリターダ圧力の結果、時間経過とともに貨車の速度がどのように変化するかをシミュレーションにより先回り予測することが可能である。従って、現実にレーダスピードメータによって測定した貨車の速度が、予測値に対し許容誤差内に入っているか否かの判断が可能となる。勿論、貨車速度入力値には貨車車体振動の効果を除去するための適切なフィルタリングが必要である。

この減速状況チェックは、単に出力信号に対する入力信号の反応をチェックするだけではなく、制御のアルゴリズムが期待通りの範囲に入っているか否かをチェックするもので、制御の信憑性をチェックしている。その意味で、このチェックは次節で論ずるバリディティチェックをも包含したものと言える。

3.3.4　プロセスＩＯバリディティチェック

前節に示したループチェックは、主としてプロセスIO出力指示を起点としてコンピュータの出力回路、現場出力機器、制御対象の反応、現場入力機器、コンピュータの入力回路をたどり、結果として入力信号が期待に沿っているか否かを調べる。本節に示すバリディティチェックは、このように現場までループに含めるチェックではなく、コンピュータの出力指示信号の値（或いは時間）をコンピュータ自らが或判断基準に照らし妥当な出力指示であるか否かを調べるものである。

YACの場合は、貨車の速度を制御するリターダの解放速度値とリターダ圧力出力値に対し、バリディティチェックをを行なっている。前節で示したように、コンピュータはリターダで貨車を減速するとき、貨車の運動方程式を基に先回りしてシ

ミュレーション計算を行ない、その結果リターダを解放し貨車を自由走行に戻す解放速度と、その速度まで減速させるに必要なリターダ圧力値を算出する。解放速度でリターダブレーキから放たれた貨車は、その殆ど水平に近い緩やかな勾配の線路を自由転走し、滞留貨車の最後尾に１～２ m/s の速度で追突連結する。解放速度のバリディティチェックは、別のシミュレーション計算を行なうのではなく、貨車重量と連結地点までの距離とをパラメータとする解放速度の簡単な表を作り、その表の値のある巾の中に、シミュレーション計算値が入っているか否かをチェックする。又、リターダ圧力出力値に対しては、貨車重量と貨車入来時の初速度とをパラメータとするリターダ圧力値の簡単な表を作り、その表の値と計算結果値とのチェックを同様に行なう。

3.4　プロセスIOのソフトウェアによるエラー検知の特長

以上に郡山ＹＡＣにおける具体例と共に２種類のプロセスＩＯのエラー検知方式を示した。その特徴は次のようになる。

- 現場機器まで含め、制御対象全体のチェックが可能。
- コンピュータシステム側では、CPU 演算回路、入出力チャネルからプロセスIOに至る全体のチェックが可能。
- ハードウェア機器は制御のために元来設置されているものを使用し、エラーチェックはソフトウェアにより実施するため、経済的に実現可能。
- リアルタイム制御システムでは、後述の二重系がよく使用されるが、ここで示したエラーチェック方式は、二重系による比較照合とは異なり、全く片方の単一系内で閉じてチェックが実施できる。

他方、この方式には、下記の弱点が認められる。

（1）ソフトウェア全体がハングしたり暴走したりすると、このチェックにかからない。（ウォッチドッグタイマに頼る）

（2） 障害部位に対する位置指摘の分解能が低い。

（3） 障害の時間分解能がやや粗い。従って、コンピュータからの誤出力が一瞬たりとも許されない過敏な（時定数が極めて小さい）対象には、適用困難である。

現実には、これらの弱点は、(2) に関しては、プロセスＩＯの信号ルートがシンプルであること、(3) に関しては、時定数の小さな制御対象の場合、コンピュータ側もそれに対応して計測や制御出力のレートを短く設計することからして、実用上解決可能である。

3.5 エラーの記録

3.5.1 エラー記録の必要性

コンピュータが何らかのエラーを検知したとき、まず最初に行うべきことは、

- どんなエラーが起ったのか。　（認識）
- それに対しどういう処置をとるか。（判断）

であり、これらの認識と判断を行うために、コンピュータの動作としてまず実施すべきことは、エラーの記録である。これは即時に行う方が、より正確なエラー時点での記録がとれる。そこで、ＣＰＵは、エラー検知信号を受けとると、

1．エラー検知時点でのＣＰＵの動作の凍結。

2．凍結されたＣＰＵ内部のステータス情報の記録。

を行う。

この目的は上述のようにまずはＣＰＵの次の動作の判断材料にするためであるが、将来の修理のための情報を保守員に与えることも含まれる。

一般的にＣＰＵはメモリやディスクに比し、多岐にわたる内部ステータスがあり、これが命令実行と共に次々と移行する。ＩＯ命令のように多クロックにわたる長

い時間がかかる命令の場合は、１ケの命令の実行の中でサブステータスがどんどん移行する。従って、エラーを検知した時点が、多クロック命令の実行途中であったとしても、その命令の終了を待たず、即刻にステータスを記録すべきである。この目的のためには、ＣＰＵのハードウェアの中核の制御ロジック部が、ソフトウェアの助けを何ら借りずに、ハードウェア自身で、実行凍結とステータス記録とを実行せねばならない。何故なら、ソフトウェアへ制御を渡し、ソフトウェアで記録をとって貰おうとすると、現在命令を一旦終了まで実行し、次の命令をエラー記録ルーチンの先頭からチェックすることとなるが、かかる動作を行う間にエラー時点のステータスがどんどん変わってゆくからである。

これを防ぐための別の手段としては、エラーが検知された瞬間に記録したいステータスを写しとるための特別のレジスタ群をあらかじめ用意し、ＣＰＵの要所要所に常時待機させておくことが考えられる。こうしておけば、エラー検知の瞬間にステータスを写しとり、その後制御をソフトウェアに渡し、写しとったデータをソフトウェアが改めてメモリへ格納すればよい。ただし、この方法をとるには、ＣＰＵのハードウェアが2倍近くに増大する。

筆者は、主任として開発した制御用計算機 MELCOM350/30 のＣＰＵにおいてエラーの瞬時記録の必要を感じ、前者のやり方、すなわち、ＣＰＵを瞬時に凍結する方法を開発した。[23] 写像用のレジスタを用意する方式については、昭和４２年当時のハードウェアとしては経済的損失が極めて大きかったため、採用しなかった。

3.5.2 即時記録の必要性

筆者は、前掲の郡山ＹＡＣプロジェクトが現地郡山操車場内にコンピュータハードウェアの据付けと試験を完了し、引き続いて、ソフトウェアの総合動作テストを実施していた昭和４２年に郡山ＹＡＣプロジェクトに並行して、MELCOM350/30 のＣＰＵ開発の主任技師をつとめた。そのとき、郡山ＹＡＣの現地デバッグの経験から次の事実が判明した。

すなわち、

ハードウェア故障の殆ど大半（８０％以上）は、瞬時故障である。

ということである。これは、当時 IBM でも発表された [20]。郡山ＹＡＣのコンピュータハードウェアは、洋服ダンス大の架が５０架からなる巨大なハードウェア量があり、普通の１０システム程度の量となっており、製作期日や使用材料もバラツイているので、この経験は一般的なものであると考えた。

瞬時故障は、システムを保守、修理する立場からは、扱いにくい故障である。故障が持続しておれば、当該ユニットはエラーを持続し、正常運転はできないものの、故障部位を特定することは各種の診断プログラムや計測器を用いて比較的容易に確認できる。しかし、瞬時故障の場合は、一瞬のエラー発生ののち、正常動作に戻ってしまい、故障部位は現象としては姿を隠してしまう。従って、かかる潜在障害を修理するためには、エラーの記録を頼りに、エラー部位を推定することとなる。このとき、エラー記録がエラー発生時点の状況を正確に、できるだけ多く、記録しておくことが、推定を助ける。

なお、この時期、IBM360 がエラー状態を自動記録することが判っていたものの詳細メカニズムは不明であった [7]。瞬時故障の原因推定の必要性を認識し、そのため即時記録の必要性を認識し、その方式を次項のごとく開発し発表したのは MELCOM350/30 が最初であった。

3.5.3 制御用計算機 MELCOM350/30 のエラー記録方式

3.5.1 に述べたごとく、MELCOM350/30 CPU において、ハードウェアコストに負担をかけず、エラー状態を即刻記録するため、図 3.11、図 3.12に示すエラー記録機構を開発した。記録する先は予め決めてある主メモリの特定番地である。

図 3.11はＣＰＵの状態遷移図である。通常の命令走行状態ではＣＰＵはＩＮＳモードになっている。操作パネルからの指示により、クロックステップとかインストラクションステップのような特殊な間欠的走行を行う場合はＭＡＮモードとなる。その他のモードではＣＰＵは命令実行せず、ＣＰＵ内外で発生する所定の信号により起動され、状態遷移すると、あらかじめ決められている固有動作を行う。TRP は各

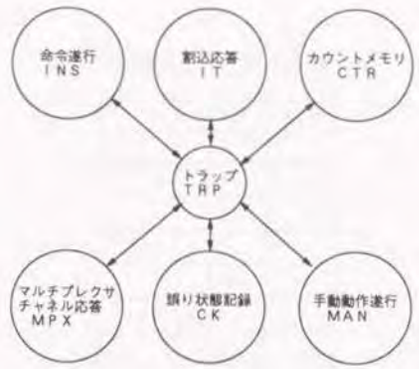


図 3.11: M350/30 CPU のモード転移図

モードの転移制御を簡易化するために設けた中継ステップであり、必ず１クロックで次のモードとなる。

表 3.2は、各モードが同時発生した場合の優先度を示す。それらの中でＣＫモードが最高位の優先度をもつ。ＣＫモードはエラー検知信号により起動される。CK モードのみは、他の全てのモードの途中であっても、瞬時にＴＲＰを経由してＣＫモードに切り替わる。IT、CTR、MPX モードは現行モードの動作終了、INS モードにおける１命令実行終了、もしくは MAN モードにおける１命令／１ステップ終了のあと転移を起こす。ＣＫモードでは前モードで処理途中であったすべての動作を凍結し、ＣＰＵ内の揮発性の全情報をメモリの所定番地へ記録する。これは完全にあらかじめ決められた動作であり、ハードワイヤドロジックで実行制御される。

図 3.12は、ＩＮＳモードの途中でエラーを検知し、ＣＫモードに移ったのち、所定の記録を済ませてエラー処理ルーチン（ＩＮＳモード）へ移行する時間軸上の状況を示す。

図 3.12のＣＫモードの先頭近辺で記録すべきステータス情報は、命令語が次へ進めば失われてしまう質の揮発性の情報である。それらは、よく知られている命令語、命令語番地、ＣＰＵモード、加算器周辺フラグ (CR,OVF,ZERO) 等のソフトウェア

表 3.2: 各モードの優先度

優先度	モード	註
1	CK	
2	IT−1	SVC
3	CTR	
4	IT−2	WDT
5	MPX	
6	IT−3	外部IT
7	INS	
8	MAN	

から見えるプログラムステータスがあるが、その他に、これらの陰でソフトウェアから見えずにテンポラリに情報を受渡したり、ストアしたりしているバッファレジスタやテンポラリフラグがある。これらはハードウェアの設計の都合で設置されているものであるが、先に述べたハードウェアの瞬時故障の原因追及の助けのために記録しておく方がよい。これらテンポラリな情報は、主メモリへ記録する途中で記録動作自身のため、他のテンポラリ情報を破壊してしまうものもあるので、記録の順番と、データ経路とを選んで記録せねばならない。

なお、CKモード中で再度エラーを検知すると、CKモードの実行を停止し、即ハードストップする。（これはクロック停止するため、図3.11には示されていない。）かかる場合は、中枢部重障害と判断し、記録をとることも停止する。なぜなら、記録内容には所詮信憑性がなく、動作を強行すると逆にエラー範囲を拡大してしまう危険が大きいからである。

3.5.4 瞬時故障の修理

前述の如く、ハードウェア故障の８０％以上は、瞬時故障である。瞬時故障は、次節で述べるリトライにより、システム的には救済される場合が多い。しかし、これに安住して放置しておくと、繰り返し同じ瞬時故障を発生し、ついには固定故障に至

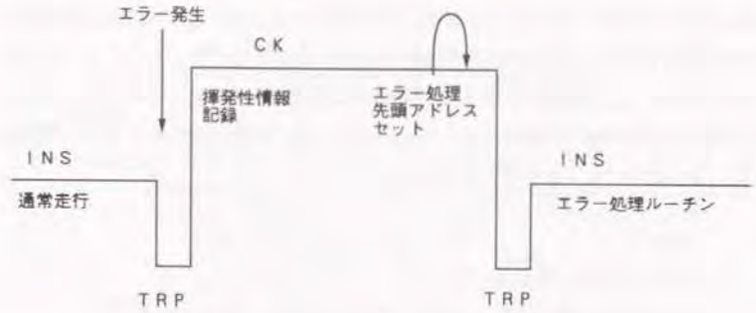


図 3.12: CPU 動作モード転移

る。

瞬時故障は、回路素子に対応して色々なものがあるが、一般的に言うと、素子や配線にもともと製造時点から、ある種の半欠陥が潜在しており、出荷検査時やシステム稼動の初期にはそれが正常動作するため欠陥として顕在化せず、時間の経過すなわち温度サイクルや通電時間の経過と共にエレクトロマイグレーションの進行や、酸化の進行、塵埃の侵入等、により、極めて微細なレベルで欠陥が成長し、そのうちに瞬間的に顕在化するフェーズに至るものである。従って、瞬時故障は、固定故障が発生する前兆的警報である、とも考えられる。

固定故障が発生する以前に、瞬時故障のときのエラーの記録をもとに、故障潜在部位を推定し、その部分を交換してしまえば、システム故障率は数分の一に低減しうる。とりわけ２４時間稼働を続けることが多いリアルタイム制御システムの場合は、システムを停止しての抜本的診断が許されないため、なおのこと、エラーのオンライン記録のみに基づいて障害潜在部位を推定し交換する方法がとられている。

図3.13に某電力会社の４ケ所の発電プラントに納入されたMELCOM350/30F（350/30の後継機）（各シングル系４台）の昭和５０年から昭和６０年までの１１年間の稼動実績と、同期間のボード交換数の実績を示す[24]。これらのボードが全て推定交換されたのか否かは実は記録に記載されてないが、ボード交換数の多寡と稼働実

績とは明かに比例していない。これはシングル系であるため、これらが全て固定障害に伴う交換であったならば、1件の障害につき必ずシステムダウンを起こす。その後のトラブルシュート時間、交換時間、再投入確認時間等々のため、しかも電力プラントの場合は全て慎重に実施されることもあって、平均4時間程度はかかることからすると、数倍悪いダウン率を示すはずである。

（計算例）

昭和57年ボード交換 11 件

全件固定障害と仮定すると全ダウンタイム 4 Hr × 11 = 44Hr

44Hr/8700Hr = 0.5 %　（ダウン率）

100 − 0.5 %= 99.5 %　（稼働率）

実際の稼働率 → 99.96 %

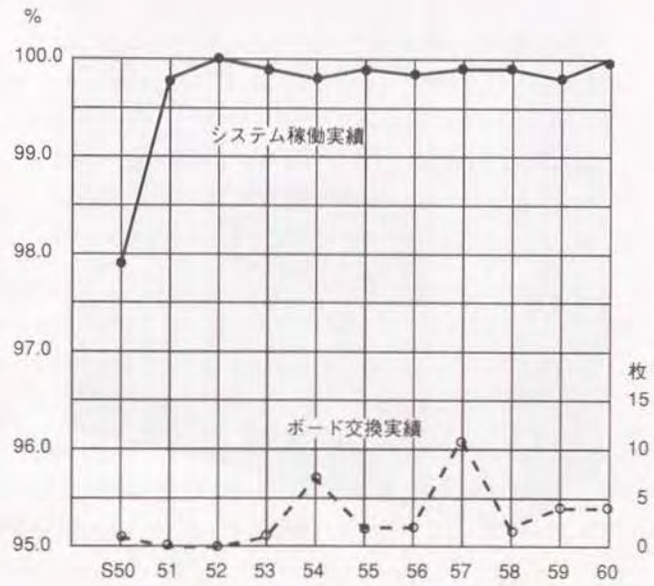
すなわち、実際の稼働率は、固定障害によるボード交換を仮定したケースよりも明らかに良好な値を示している。換言すれば、ボード交換は固定障害が発生した後、システムダウンを各々4時間起こして実施されたとすると計算が合わない。11件のうち1件は真にダウンを引き起こしたであろうが、他の10件は実は固定障害になっていない段階で計画的に短い予防保守時間帯の中で推定交換したものと考えると、下記の如く説明がつく。

全ダウンタイム 4 Hr × 1件 = 4Hr

4Hr/8700Hr = 0.05 %

稼働率　　= 99.95 %

	S50	51	52	53	54	55	56	57	58	59	60
システム稼働率（4台平均）	97.93	99.84	100.0	99.92	99.87	99.96	99.90	99.96	99.93	99.87	100.0
ボード交換枚数	1	0	0	1	6	2	2	11	2	4	4



・データロギングシステム
・各システムは単一ＣＰＵ
・瞬時故障の効率的除去により単一ＣＰＵでも高稼働率達成

図 3.13: システム稼働実績とボード交換実績
（某電力会社納入 M350/30F(30後継機) 4システム平均）
（株）メルコムサービス保守記録データ

図 3.14に MELCOM350/30F の本体部分を示す。

図 3.14: MELCOM350/30F 本体部

3.6 ソフトウェアバグの早期収束策

第 2 章で述べた如く、リアルタイム制御システムにおいては、システムの任務と運転形態が固定され、特定の運転員が操作を行うため、ソフトウェアバグが長期間発生し続けることは一般にはない。

しかし、稼働初期の期間といえどもソフトウェアバグがあっては困るので、稼働に入る前の試験期間や試運転期間に極力ソフトウェアバグを摘出してしまう必要がある。かかるソフトウェアバグの追跡容易性はリアルタイム制御システムおよびリアルタイムコンピュータの一つの要求仕様である。

3.5 節で述べているエラーの記録は、ハードウェア障害、とりわけ瞬時障害の原因推定に役立つものであるが、同時にソフトウェアバグの原因追及にも大いに役に立つものである。何故なら、エラーの記録は、エラーを起こした原因がハードウェア障害であろうと、ソフトウエアバグであろうと、全く同様にステータス記録を行うからである。換言すれば、記録を行っている段階では、エラー原因のことは不問である。

MELCOM350-30F では、ソフトウェアバグの追及容易性を更に高めるため、エラーステータス記録の他に次の機能を持たせることとした。すなわち、

ブランチアドレスの記録

である。これはこの時期筆者等がはじめて考案し具体化した機能である。これは、如何なるプログラムが走行している間も常に、ブランチタイプの命令が実行されると必ず、そのブランチタイプ命令の所在アドレスを特定のレジスタへ記録する機能である [25]。

従って、特定レジスタには最後のブランチ命令のアドレスが記録されている。ソフトウェアバグの場合、ソフトウェアの走行シーケンスが乱れるケースが多く、エラー検知時期で走っているプログラム以前のブランチのアドレスを知ることは大いに参考になる。

この機能はエラー記録とは異なる機能であるが、リアルタイムコンピュータに要求されるソフトウェアバグの早期収束策に有効な機能であるためここで述べた。

なお、同様の機能は、最近のコンピュータには、ほぼ常備されるようになってきている。

3.7 リトライによるエラーのマスク

エラー訂正符号は、ハードウェアに冗長ビットを付加することによりエラーの検知と訂正を行なう。これに対し、リトライはエラーを検知したあと、同じ動作をもう一度（又は複数回）繰り返すことによりエラーが何らかの理由で消滅し正常動作が行なわれることを期待する。時間的な冗長コストを支払うとの見方もできる。リトライを行なう前に、エラーの原因であるフォールトに対して何ら積極的な対応策を行なうことはせず、単に同じ動作を繰り返すだけのものであるから、結果の保証はない。しかし、3.5.2にも示した如く、エラー現象としては瞬時的なものが多いとの実績があり、現実には有効な手法である。

3.7.1 CPUエラー時のリトライ

リトライは、エラーを検知したあと、ハードウェアによる状態記録、ソフトウェアによる状態記録に続いて行なわれる。リトライを行なうには、エラーを検知したために中断した作業をエラー検知前の適当な区切りに遡ってスタート状態に設定せねばならない。それと同時に、エラーの波及範囲が限定されていること、すなわち中断した作業以外までエラーが波及していないことも条件となる。これらの条件は、CPUのエラー検知が時間分解能の面で精度よく行なわれなければならぬことを意味する。従って、クロックレベルでのCPUの二重照合又は三重多数決チェックが前提条件となる。

リアルタイム制御システムにおいては、2.2.5に示した如く、引継ぎ時間の制約から二重照合チェックは不適切であり、リトライを実施する条件が満たされない。三重多数決チェックの場合は、引継ぎ時間零であるから、リアルタイム向きである。よって、この場合はリトライ条件は満足される。しかし、実際にリトライを試みる間は短時間ながらシステム出力は中断するので、クリティカルなリアルタイム系ではそれが許されるかどうか問題である。三重系ならば、一つの系のCPUがフェイリャになっても、残2系でそのまま業務続行が可能であるから、フェイリャとなったCPUのリトライは別のタイミングで実施するのが妥当である[26][27]。

3.7.2 プロセスIOのリトライ

リアルタイム制御システムでは、3.3.2および3.3.3に述べたように、ソフトウェアによるプロセスＩＯのハードウェア動作のチェックが有効である。それと同時に、エラー検知したあと、ソフトウェアによるプロセスＩＯ動作のリトライが経験的に有効である。理由としては、リレー接点の接触不良とか、現場機器のメカニカルな異物混入不良のような場合に対し、リトライ動作が何らかの回復効果をもつものと考えられる。

以下に郡山ＹＡＣにおける分岐機レスポンスチェックのあとのリトライの有効性を具体的に説明する。

- 分岐機転換出力に対し、規定時間内に転換応答入力無し
- 分岐機レスポンスチェックにより、エラー発生検知
- エラーステータス記録のあと、エラー処理ルーチンへ切替え
- エラー処理ルーチンで分岐機への再出力を試行
- 実際の障害モードとしては分岐機レール間に小石がはさまる例が多い。
- よって、エラー処理ルーチンは再出力の前に分岐機を逆方向へ一旦引き戻す。
- その直後に転換命令を出す。
- 必要あればn回反復。
- この結果小石がはじき飛ばされレスポンスが正常に戻ってくる。
- レスポンスが正常になれば元のプログラムへリターンする。

リアルタイム制御システムに一般的に使用されるリレー接点出力のレスポンスチェックエラーの場合にも、この種のリトライが有効である。

3.8 まとめ

エラーの検知は、重要使命を担うコンピュータにおいては必須な技術であり、とりわけフォールトトレラントコンピュータにおいて最も基本となる技術であることを示した。

特に、本論文で主題としているリアルタイムコンピュータシステムにおいて、引継ぎ時間の制約から、二重照合チェックは有効ではなく、チェックは各単独系内で独立に実施される必要があることを示した。これが実現できれば、二重系で片系がフェイリャしたとき、残片系のみで運転続行しうるので、第3のスペアを立上げる必要はない。この独立チェックを実現するには、リアルタイム制御システムの特質を生かしたソフトウェアによる自己チェックが有効であることを示した。

また、エラーステータスの記録は、ハードウェアには瞬時故障が多いという現実の体験から、それを克服する手段として有効なこと、また実際にある納入先における１１年間の稼働記録とボード交換記録とを比較参照してみるとき、瞬時故障に対応してのボードの推定交換が有効に働いていることを示した。

更に、リトライは、瞬時故障に対してのみならず、リアルタイム制御系の現場においてもよく用いられるリレーとか転換器とかの不具合に対しても、塵埃や異物をはねとばす効果があり、一般的に有効な手段であることを示した。

第４章

フォールトユニット切離しと冗長ユニットによる引継ぎ

第 4 章

フォールトユニット切離しと冗長ユニットによる引継ぎ

4.1 はじめに

本章においては、システムの一部にフォールトが発生した際に、システム全体としては正常運転を継続するための中核技術である、フォールトユニット切離しと冗長ユニットによる引継ぎについて述べる。

以下、4．2節においては、フォールト部の切離しを確実に行なうための条件、切離す部分の単位、冗長ユニットの事前準備について述べる。4．3節においては、フォールトトレラント リアルタイム制御システムにおける現実解として実施例の多い独立型ホットスタンバイシステムを構築するに必要な要件を明らかにする。4．4節においては、YAC システムにおける独立型ホットスタンバイ系の実例を示し、そこで研究し実使用した構築技術を述べる。4．5節においては、YAC システムの稼働実績を分析する。4．6節および4．7節においては、市販マイクロプロセッサを使った三重多数決方式に基づく新方式「予防引継ぎ方式」を提案し、その信頼性の評価を行なう。

4.2 フォールトユニット切離しと冗長ユニットによる引継ぎ

前章において、エラー検知、記録、リトライについて述べたが、リトライ不成功のあとは、システムはここで述べるフォールトユニット切離しと冗長ユニット立上げを行ないシステム全体としての正常運転の続行をはかる。フォールトユニットとしては、運転の中心となっている CPU 又はコンピュータサブシステムである場合が問題

となるので、以下ではそれらの場合について論ずる。

4.2.1 確実な切離し

フォールトCPU切離しについての技術的ポイントは、当該フォールトＣＰＵを確実に切離す、ということである。それには、次の３点が重要な点である。

- フォールトＣＰＵ自身に切離し動作の最終責任を持たせぬこと。

  これは、切離しの指示、実行がフォールトを抱えるＣＰＵ自身には確実に行われる保証がないためである。エラーを検知したあと、エラー記録とかリトライを行なうからには、自分自身の切離しも又、実行できるケースは多い。しかしそうでない重症の障害もありうる。

  よって、システム全体を監視している専用装置か、もしくは正常運転している冗長ＣＰＵによって切離し動作を行うべきである。

- 切離すということは電気的に切離すこと（ハイインピーダンス化）である。

  論理ゲートによる論理的な切離しでは不十分である。これは、フォールトＣＰＵの論理動作自体が正しく実行されぬケースがあり得るためで、かかる場合電気的接続が残存していると、フォールトＣＰＵから意図せぬ信号が出て、他のユニットに擾乱を与える恐れがある。

- 電気的な切離し（および投入）回路は、それ自身がフォールトトレラントになること。

  切離し（および投入）回路の典型的な例はリレーである。このリレーに単一フォールト（例えば接点接触不良、或は接点癒着）が発生すると、冗長CPUを用意していてもそれを生かすことができない。従って、かかるリレーはシステム全体にとって一重素子となっている。こういう場合は、このリレー自身をフォールトトレラントにすべきである。

  すなわち、図4.1に示す構成が良く知られている一例である。

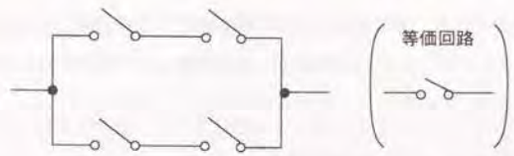


図 4.1: 単一故障にトレラントなリレー構成

この場合、単一のリレーの接触不良率を λ_1 とすると、図4.1の場合の接触不良は、$4\lambda_1^2$ となる。又、単一のリレーの癒着不良率を λ_2 とすると、図4.1の場合の癒着不良率は、$2\lambda_2^2$ となる。 $\lambda_1 \ll 1$、 $\lambda_2 \ll 1$ であるから、いづれの不良に対しても図4.1は２乗のオーダで故障率が減少する。

4.2.2 切離しユニットの単位

上述の如く、フォールトＣＰＵを切離すとき、エラーを報告した当該ＣＰＵだけを切離せばよいかどうか、予めシステム構築の際に検討しておく必要がある。

ＳＳＩ、ＭＳＩ時代までのコンピュータでは、性能を得るためにコンピュータ全体の物理的な大きさを抑える必要があり、主メモリ、ＣＰＵ、ＩＯチャネル等の実装構造がお互いに密着しており、その中からＣＰＵだけを切離せるような構造をしていなかった。従って切離し単位としては、少なくとも主メモリもＩＯチャネルも一体となったコンピュータ本体部一式が単位であった。

ＬＳＩ時代のコンピュータでは、実装構造としては、バス構造となりバスの上に、主メモリやＣＰＵやＩＯチャネルがボード単位で実装されている。よって、或るＣＰＵだけを切離すことは可能である。この場合は切離し単位としては、物理的、構造的制約ではなくて、エラー検知の位置分解能の点で決まる。すなわち、ＣＰＵだけを切離すケース（メモリは切離さない）では、エラーがＣＰＵの内部に留まっており、メモリ内容には波及していないことが必要であり、それが可能となるような細かく且つ、即時のエラー検知能力が必要である。

最近のコンピュータはＬＳＩの高集積化が更に一段と進み、メモリもＣＰＵもＩ

Oチャネルもその他も、ワンボード上に集約されている例が増大している（特にワークステーション、パソコン）。このような高集積度のハードウェアになると再び切離し単位はＭＳＩ時代と同じく、コンピュータまるごと一式に戻る。

4.2.3 冗長ユニットによる引継ぎ

冗長ユニットの投入に際し、システムとしては、外部の制御対象に対して切れ目なく正常な運転を継続する必要がある。そのためにはフォールトＣＰＵ／又はコンピュータの切離しに引続き、直ちに冗長ＣＰＵ／又はコンピュータの投入が行なわれ、且つ、冗長ユニットは、直ちに正常運転をエラー検知直前の状態に連続して開始せねばならない。このとき、冗長ユニットが正常運転開始までに必要とする時間は、リアルタイム制御対象の時定数によって巾がある。特に、ミリ秒オーダを求められるクリティカルなリアルタイム制御システムにおいては注意を要する。

冗長ユニットとして投入されるのがコンピューター式の場合、投入後に運転立上げをしていては間に合わない。かかる場合には、冗長コンピュータも投入される以前から、運転中のコンピュータと同時並列に同じ入力信号を受けとり、同じ制御プログラムが走行しつつ、投入に備えておかなければ、ミリ秒の単位で運転を引継ぐことはできない。いわゆるホットスタンバイ（熱予備待機）が必要となる。

冗長ユニットとして投入されるのが、ＣＰＵ単位の場合は、時間がかかる大容量外部メモリの立上げが不要（既に運転中のものをそのまま使う）なので、極めて短時間に立上る。よって、ホットスタンバイでなくてもよい。つまり投入される前は、制御に関係ない別の仕事をやっていても構わない。ただし、その仕事は、いつ中断されても構わないものとする必要がある。

4.3 独立型ホットスタンバイシステム

4.3.1 切替単位

リアルタイム制御システムの中枢となるコンピュータをフォールトトレラント化するための構築技術として最も現実的な解は、コンピュータまるごと一式を運転系、他の一式をホットスタンバイ待機系とするやり方である。この方法は、ＬＳＩが発達

してボードＣＰＵが実現している現在では、古いやり方と考えられぬこともないが、それでもなお、下記の理由により現在でも一般性がある。

- まるごと一式を運転系・他系を待機系、と区分することは、区分単位として、標準製品のハードウエアをほぼそのまま使える。
- 逆に言うと、バス方式コンピュータにおいて、ボードＣＰＵ単位での切替えを行うならば、ＣＰＵは当然２式用意するとして、バス、電源、クロック、等についても、各々二重構成となるように用意せねばならず、これは、実装上から一般標準製品ではなく、専用特殊製品となる。

  また、バス上のフォールトボードを確実に切離すためには、単純にバスインタフェース論理ゲートを閉じるだけでなく、ハイインピーダンス化する必要があり、そのための手当てがいる。
- ＬＳＩ化が更に発達すると、システムオンボードとなるので、その場合は、ＣＰＵ単位の切替えは不可能で、システム単位の切替えとなる。

4.3.2 独立型ホットスタンバイシステムの信頼度の数学的モデル

リアルタイム制御システムが安定した稼働期に入ったとき、その故障率はよく知られているように、図4.2に示すバスタブカーブの底の偶発故障期の状態になる。

このとき、時間の関数である故障率 $\lambda(t)$ は大体一定の値をとる。

$$\lambda(t) = \lambda \cdots \text{一定値} \tag{4.1}$$

時間の関数である信頼度関数を $R(t)$、不信頼度関数を $F(t) = 1 - R(t)$ および故障密度関数を $f(t) = dF(t)/dt$（単位時間あたりの故障数）とすると、よく知られているように指数分布

$$R(t) = e^{-\lambda t} \tag{4.2}$$

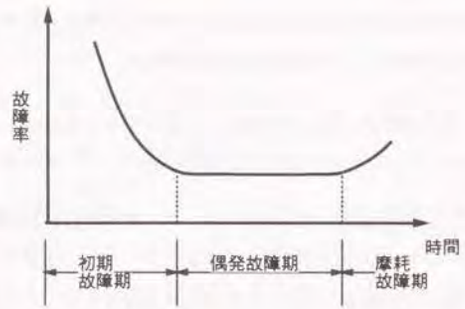


図 4.2: 故障率曲線

$$F(t) = 1 - e^{-\lambda t} \tag{4.3}$$

$$f(t) = \lambda e^{-\lambda t} \tag{4.4}$$

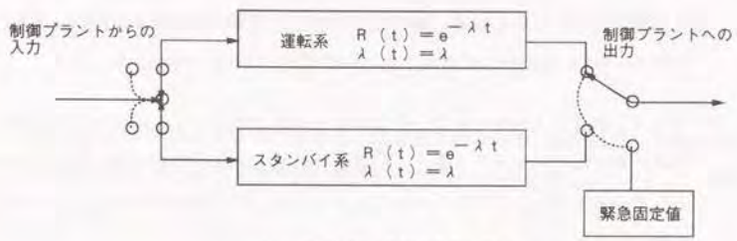
が導かれ、又同時に

$$MTBF = \int_0^\infty t f(t) dt = \frac{1}{\lambda} \tag{4.5}$$

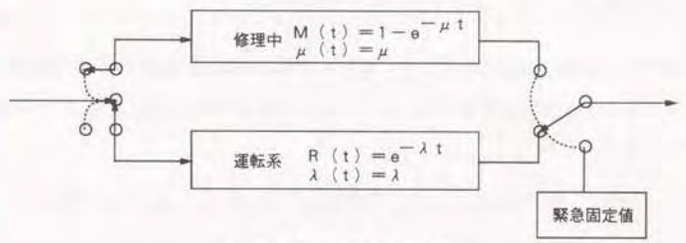
が導かれる。

これらは、偶発故障期に入ったリアルタイムコンピュータシステムの１つの系の信頼度モデルである。これらを、本研究で筆者が示す独立型ホットスタンバイ二重系として構築した場合の全体系の信頼度の諸指標は以下の如くになる。一般には、並列二重系の信頼度として既に知られているが [28]、ここで改めて独立型ホットスタンバイシステムの定義を明確にし、その定義に沿って信頼度指標を算出する。

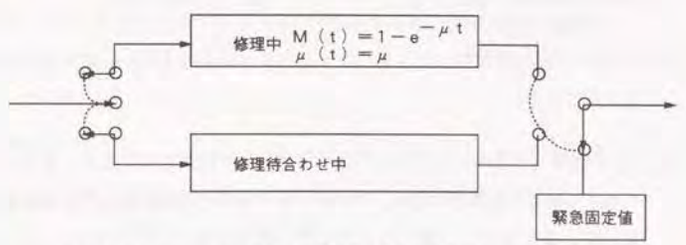
独立型ホットスタンバイシステムの３つの状態を図 4.3 (a), (b), (c) に示した。各々の運転条件を以下のように定義する。ただし、図 4.3(b)(c) において、修理が完了する確率関数を M(t) とし、修理が続行する（故障継続）確率関数を 1-M(t)、修理率を μ (t) とし、一定技能の保守員を仮定し、$\mu(t) = \mu$ ･･･一定、すなわち $M(t) = 1 - e^{-\mu t}$ という分布関数を仮定する。



（a）正常運転（並列二重系）状態



（b）片系運転、片系修理中状態



（c）システムフェイリャ状態、両系故障、うち片系修理中、片系修理待合わせ中

図 4.3: 独立型ホットスタンバイシステムの運転モデル図

(1) 全体系は、同等でかつ独立に運転される２つのコンピュータサブシステムからなる。独立という意味は、正常運転時には相互間でのデータの授受なく運転されるということである。又、各々の系の中で、CPU、主メモリ、外部メモリ、プロセスIO一式を持つということでもある。

(2) プラントからの入力データは、２つの系に同時に与えられる。プラントへの出力は片系からのみ与えられる。この系を運転系と呼ぶ。他方の系も同じ出力を出しているものの、プラントへは与えられない。この系をホットスタンバイ系と呼ぶ。

(3) 各系とも、エラー検知を自己系内で実施する。

(4) 図4.3には、図示されていない簡単な全体系監視切替装置があり、運転系からのエラー検知報告を受けて、入力、出力の経路を図4.3(a)から(b)へと切替える。

同様に、片系運転時にエラー報告を受けると(b)から(c)へと切替える。

又、同様に、両系故障時に片系から修理完了報告を受けると(c)から(b)、片系運転時に修理完了報告を受けると(b)から(a)へと切替える。

(5) (a)から(b)へ切替わったとき、ホットスタンバイ系は直ちに正常運転を行なうことができる。

(6) (b)の状態で片系のハードウエア面の修理完了が確認されたとき、直ちに(b)から(a)へ切替わるのではなく、ソフトウェア面での準備を行なう必要がある。すなわち、修理完了したばかりの系のファイルを立上げる必要がある。このとき、両系の間にデータの授受がある。すなわち、運転系から修理完了系へ最新のファイルデータが送られる。しかるのち、(b)から(a)へ切替わる。

(7) 厳密には、全体監視切替装置にも或る故障率がある。この装置の故障は、後で別途評価することとし、ここでは 故障率＝零 を仮定する。

(8) 運転中の故障の修理に対応できる保守員は１人（又は１チーム）居るものと仮定する。両系が共に故障したときは、保守員は先に故障した系の修理を続行し、あとで故障した系の修理はその間待ち状態となる。先の系の修理が完了したら、直ちに待たせていた系の修理にとりかかる。

(9) 厳密には、片系運転中の片系修理と両系故障時の片系（先故障系）修理との修理率は異なると見るべきであるが、ここでは同一かつ一定 $\mu(t) = \mu$ ･･･一定値とする。

筆者のいう独立型ホットスタンバイ系を図4.3(a)(b)(c)および上記(1)～(9)の如く定義する。そうすると、これらの定義からして、この全体系の状態を図4.4のようなマルコフ過程遷移図に表すことができる。

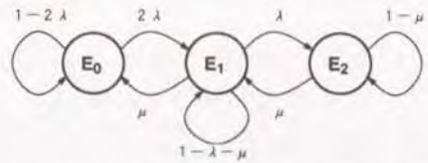


図 4.4: 独立型ホットスタンバイシステムのマルコフ過程遷移図

上図において、

E_0　　：両系稼働状態
E_1　　：片系稼働、片系修理状態
E_2　　：両系故障、うち片系修理、片系修理待ち状態
$E_0 \rightarrow E_1$　：故障率λの２つの系のいづれか片方の故障により移る。
$E_1 \rightarrow E_2$　：故障率λの１つの系故障により移る。
$E_2 \rightarrow E_1$　：修理率μの１つの系の修理完により移る。
$E_1 \rightarrow E_0$　：同上。

上図より、遷移確率行列Pは次のように表される。

$$P = \begin{pmatrix} E_{00} & E_{01} & E_{02} \\ E_{10} & E_{11} & E_{12} \\ E_{20} & E_{21} & E_{22} \end{pmatrix} = \begin{pmatrix} 1-2\lambda & 2\lambda & 0 \\ \mu & 1-\lambda-\mu & \lambda \\ 0 & \mu & 1-\mu \end{pmatrix} \tag{4.6}$$

ここで、十分長い時間経過したあと、システム全体は E_0、E_1、E_2 間を一定の確率で往復する平衡状態に至る。その平衡状態時において、E_0、E_1、E_2 の極限確率を P_0、P_1、P_2 とすると、

$$(\ P_0\ \ P_1\ \ P_2\ )P\ =\ (\ P_0\ \ P_1\ \ P_2\ ) \tag{4.7}$$

$$P_0 + P_1 + P_2\ =\ 1 \tag{4.8}$$

が成立する。

(4.6) (4.7) を展開すると、次のようになる。

$$P_0\ =\ (1-2\lambda)P_0\ +\ \mu P_1 \tag{4.9}$$

$$P_1\ =\ 2\lambda P_0 + (1-\lambda-\mu)P_1 + \mu P_2 \tag{4.10}$$

$$P_2\ =\ \lambda P_1 + (1-\mu)P_2 \tag{4.11}$$

(4.8) および (4.9)(4.10)(4.11) を解くと、P_0、P_1、P_2 は次のように表される。

$$P_0\ =\ \frac{\mu^2}{\lambda^2 + (\lambda+\mu)^2} \tag{4.12}$$

$$P_1\ =\ \frac{2\lambda\mu}{\lambda^2 + (\lambda+\mu)^2} \tag{4.13}$$

$$P_2\ =\ \frac{2\lambda^2}{\lambda^2 + (\lambda+\mu)^2} \tag{4.14}$$

(4.12)(4.13)(4.14) を利用して、全体系の MTBF を次のように求める。

$$\text{全体系稼働確率（平衡時）}\ =\ A\ =\ P_0 + P_1 \tag{4.15}$$

$$\text{全体系フェイリャ確率（平衡時）}\ =\ F\ =\ P_2 \tag{4.16}$$

$$\text{全体系}MTBF : \text{全体系}MTTR\ =\ A\ :\ F \tag{4.17}$$

ここで全体系 MTTR は、$E_2 \rightarrow E_1$ への修理員 1 名による修理完了確率を μ とし、分布を指数分布 $M(t) = 1 - e^{-\mu t}$ としているから、修理が続行（故障が続行）する確率 $D(t)$ は、

$$D(t)\ =\ 1 - M(t)\ =\ e^{-\mu t} \tag{4.18}$$

となり、その平均持続時間が全体系 MTTR となる。よって、次式となる。

$$\text{全体系}MTTR\ =\ \int_0^\infty tD(t)dt\ =\ \frac{1}{\mu} \tag{4.19}$$

(4.12)(4.13)(4.14)(4.19) を (4.17) に代入すると、次式を得る。

$$\text{全体系}MTBF\ :\ \frac{1}{\mu}\ =\ \frac{\mu^2 + 2\lambda\mu}{\lambda^2 + (\lambda+\mu)^2}\ :\ \frac{2\lambda^2}{\lambda^2 + (\lambda+\mu)^2} \tag{4.20}$$

(4.20) から、全体系 MTBF は次の如く求まる。

$$\text{全体系}MTBF\ =\ \frac{\mu + 2\lambda}{2\lambda^2} \tag{4.21}$$

一般に、$\mu \gg \lambda$ であるから、(4.21) は近似的には次式となる。

$$\text{全体系}MTBF\ \cong\ \frac{\mu}{2\lambda^2}\ =\ \frac{1}{2} \cdot \frac{(\text{片系}MTBF)^2}{(\text{片系}MTTR)} \tag{4.22}$$

(4.21) によって得た結果は、既に先人により二重系における MTBF として示されている値に等しい [28]。本節では、独立型ホットスタンバイシステムの運転形態を明確に定義した上で、数式的に同じ結果を得た。逆に言うと、独立型ホットスタンバイシステムの定義からはずれた別形態の二重系（或いは不明確な二重系）では、別の数式結果となる可能性がある。更に換言するならば、(4.21) 或いは (4.22) を期待して二

重系を構築する場合は、運転条件をここに示した独立型ホットスタンバイシステムを満足するように設計すればよい。さもないときは、改めて数学モデルを作って検証せねばならない。

4.3.3 独立型ホットスタンバイシステム構築上の注意点

独立型ホットスタンバイシステムの定義を前節に示したが、これを実際に構築する場合のシステム技術上の注意点は次の2点である。

(1) エラー検知の独立性

各系は、単独系としてエラーを自己検知する必要がある。相手系との照合チェックに依存する場合は、片系がフェイリャし、残片系で運転することができない。すなわち、図4.4マルコフ遷移図における状態 E_1 が存在し得ない。

(2) 修理の独立性

保守員が故障した系の修理にあたるときは、故障部位の特定、修理実施、正常動作可能となったことの確認の各作業を通して、運転中の片系の運転を妨げることなく、作業が実行されねばならない。さもないと、4.4における状態 E_1 から E_0 への復帰が図のルートでは果たされない。

リアルタイム制御システムの場合は、量的に巨大なプロセスIO部分の最終正常動作確認を具体的にどう実施するかが問題である。プラントからの多数の入力信号の受入れ回路を如何にして確認するか、プラントへ与える出力信号、とりわけ出力の微妙なタイミングが問題になるもののタイミングをどう確認するかが問題である。なお、ここでいう確認とは、実運転投入の前の最終総合確認である。

これらの確認を行いたいとき、実際のプラントは、他系が接続して制御を続行しているから、診断や確認のため横からその場へ割り込むことはできない。従って、オフライン状態にある単独系のプロセスＩＯの修理と確認の何らかの実現手段をあらかじめ持っている必要がある。さもなくば、ＭＴＴＲが非常に長びいてしまい、本来単独系として持っているＭＴＴＲ値を大幅に上廻ってしまう。

すなわち、単独系であったならばシステムダウン後の回復確認試験のとき、プラントと所定の信号往復を確認することがおおむね可能であり、効率的に確認試験しうる。これにほぼ等価な確認試験をオフラインで実施できるような用意が必要となる。

4.4 具体的実施例　ＹＡＣシステムにおける独立型ホットスタンバイ方式

4.4.1 ＹＡＣコンピュータの信頼性ニーズ

郡山ＹＡＣのシステムの概要は3.3.1にて述べた。ここで、ＹＡＣにおけるコンピュータへの処理能力仕様、信頼性に対する要求仕様の概要を述べると、下記のようになる。なお、当然ながら国鉄の業務は1日24時間、1年365日の運転である。とりわけ貨物列車は旅客列車の密度が薄い夜間がピークとなる。

(1) 取扱貨車車輌数　　２４００輌／日

(2) コンピュータ稼働率　　９９．９％

(1)については、1日24時間あたり2時間程度の操作員の交替休憩時間と、ハンプ押上機関車の入れ換え時間を考慮すると、おおよそ３０秒毎に1車輌の散転走行のペースとなる。1車輌が平均して3分程度かかって追突連結するので、ＹＡＣコンピュータはあちこちに散転走行中の貨車を同時に6台程度、監視制御する必要がある。　走行制御するのと全く独立に、常時、隣接の何ヶ所かの操車場から到着予定貨物列車の編成内容の情報（貨報）が入信し、他方で郡山操車場で編成を終了し出発する貨物列車の貨報を他操車場へ次々と発信する必要がある。当然ながら到着予定ファイル、到着済みファイル、編成組替計画ファイル、散転ファイル、散転終了ファイル、編成完了ファイル、出発予定ファイル、出発済みファイル等を業務進行のステップバイステップで貨車の所在状況を正確にファイルしてゆくこととなる。

前者の貨車散転速度のリアルタイム制御に関係する業務を一括して「速度制御」、後者の貨報の入発信と操車場内での所在フォロウに関係する業務を「情報処理」と呼ぶ。

(2) の稼働率については、月に1回、30分程度のシステムダウンなら許容しうるという漠然とした要求と、この他に最大限度復旧時間1時間という厳しい定量的要求があった。

後者については、郡山操車場で1時間以上の操業停止が発生すると、操車場へ近づきつつある貨物列車が、郡山操車場の収容能力パンクのため途中の駅で時間待ちすることとなり、これがひいては、東北本線の旅客列車ダイヤを遅らせてしまうこととなることから発生した要求であった。

4.4.2 郡山ＹＡＣのコンピュータ構成

コンピュータ構成については、まず、処理能力の面から検討した。業務としては大きくて分けて、リアルタイムの「速度制御」と時間制約のゆるやかな「情報処理」があり、この両者を同時に1台のコンピュータで処理することとは、当時の最新鋭コンピュータをもってしても無理であった。

そこで、速度制御を担当するコンピュータＦＣＣ (Freight Car Computer) と情報処理を担当するコンピュータＤＰＣ (Data Processing Computer) と2台のコンピュータの分担により業務をこなすこととし、両コンピュータの間のデータ授受を迅速ならしめるため、チャネル／チャネル直結装置で接続した。

次に信頼性の面からの要求を満たすためには、当時の汎用コンピュータの平均的な稼働率99％をもってしては、単独系での達成は不可能であり、これを二重系構成とし、かつ系の切替時においてもリアルタイム制御を連続的に継続させるため、ホットスタンバイ方式をとった。又、三重系にすること、或いはスペア系を1式持つことは、コンピュータ1式のコストが極めて高額であった当時では、システム全体コストから許容しえず、二重系が限界であった。そこで、前節に論じた独立型ホットスタンバイシステムを採用することとした。図4.5に郡山ＹＡＣのコンピュータシステムの構成図を示す [8]。

図4.5に示す速度制御コンピュータＦＣＣＩとＦＣＣⅡとは、独立型ホットスタンバイ系を形成している。情報処理コンピュータＤＰＣＩとＤＰＣⅡについては、図の上ではホットスタンバイの如き接続状態となってはいるが、ＤＰＣの場合はオンライ

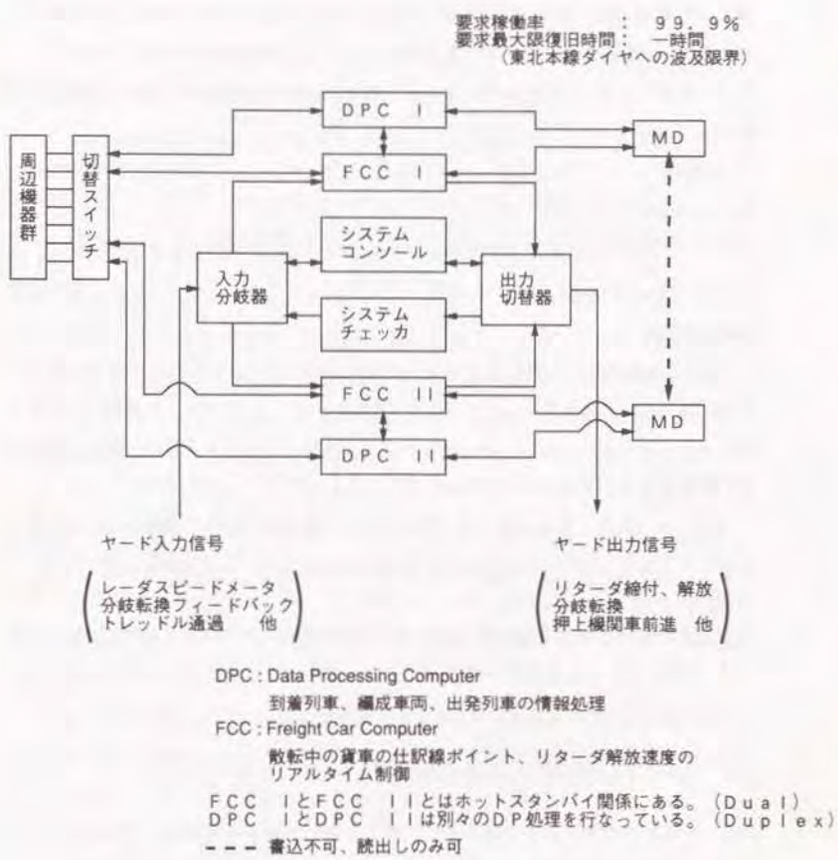


図 4.5: YAC コンピュータシステム構成図

ン処理はあるものの、きびしいリアルタイム処理はなく時間的な余裕があるため、同じ作業はやっておらず、従ってホットスタンバイではない。換言すると、ＦＣＣＩとＦＣＣⅡはＤｕａｌ系、、ＤＰＣＩとＤＰＣⅡはＤｕｐｌｅｘ系を構成している。この結果、コンピュータにフォールトが発生し、系の切替えを行うにもＦＣＣフォールトの場合とＤＰＣフォールトの場合とで異なる切替方式となり、その状態遷移図を図4.6に示す。

具体的に、フォールトユニットの切離しと冗長ユニットの引継ぎについて図4.5と図4.6を用いて説明する。

ＤＰＣ、ＦＣＣいずれのコンピュータもエラーを自己検知すると、システムコンソール（図4.5中央部に図示）に通報し、システムコンソールがシステム全体の状態遷移を制御する。

なお、当然ながら、運転員がマニュアルオーバーライドすることも可能となっている。又、ＤＰＣの自己チェックは表3.1に示すプロセスIOチェックの他はＦＣＣと同じチェックをやっており、その他にファイル相互の一貫性チェック、駅名、貨車型名、貨車重量データ等のバリディティチェックを行なっている。

例えば、図4.6の正常運転状態 ① において、運転系のＦＣＣがエラーを自己検知すると、システムコンソールは全体系を ② の状態へ移す。具体的な動作としては、

（1） 図4.5における出力切替器をそれまでの運転系サイドからスタンバイサイドへ切り替える。すなわちその後のヤード出力信号はそれまでのスタンバイ系ＦＣＣから与えられることとなる。

（2） それまでの運転系ＦＣＣおよびＤＰＣに対し停止を指示する。

（3） それまでのスタンバイ系ＤＰＣに対し、運転準備を指示する。そのスタンバイ系ＤＰＣは図4.5の点線で示すリードオンリのバスを経由して、運転系だった磁気ドラムから自己の磁気ドラムへＤＰＣ運転に必要なファイルをとりこむ。

（4） ＤＰＣ準備（ファイル準備）が完了するとシステムコンソールへ通知を出し、システムコンソールは全体系を ② から ③ へ移す。

（5） 保守員はエラー検知し停止しているＦＣＣの故障原因の究明をオフラインで行なう。原因が判明し、修理処理を完了すると、保守員はシステムコンソールを操作し、現在ダウンしている系のＦＣＣ入力、出力にシステムチェッカ（図4.5中央に図示）を接続し、プロセスＩＯ診断プログラムを走行せしめつつ、ＦＣＣ全体動作の最終確認を行う。

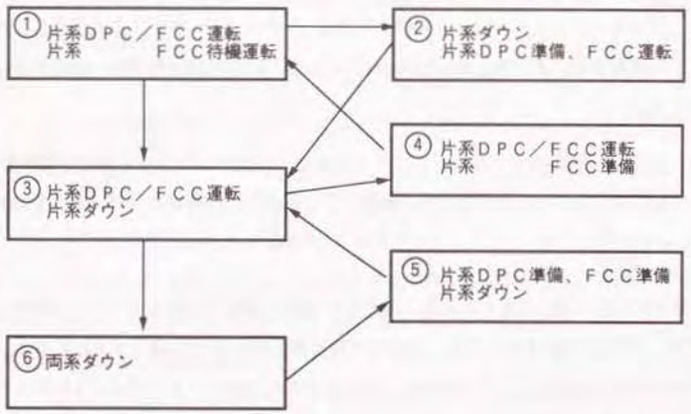
その結果ＯＫであれば、システムチェッカを切離す。

（6） システムコンソールは、その後全体系を ③ から ④ へ移す。ここでＦＣＣは系に投入され、再び図4.5の点線のバスにより、転送準備中の貨車の情報を受けとる。

（7） ＦＣＣの貨車情報受取り完了したところで、システムコンソールは全体系を ④ から ① へ戻す。このときから準備完了したＦＣＣはホットスタンバイを開始する。

図4.6の② ④ ⑤ に示す「準備」ステータスは、運転のためのファイルの準備期間であり、FCC準備 は約１０秒、DPC準備は約１分の時間で完了する。しかし、この短時間の間に運転系がダウンすることも理論的にはありうるので、そういうケースが起きると両系ダウンとなる。従って、厳密にいうと、② ④ ⑤ から ⑥ への矢印もあり、同様に ④ から ③ への矢印もある。図4.6は、図を簡略化するため、確率的に少ない矢印を省略している。

なお、コンピュータシステムの設置状況を図4.7、システムコンソールを図4.8に示す。システムは全部で５０架からなる巨大なもので、この図の写真はその一部を写している。



(註) ② ④ ⑤ は、たかだか１分程度の短時間で次の状態へ移る。
この短時間の間に別のダウンが発生することも理論上ありうるが、
その場合の移行の矢印は簡略化して省いてある。

図 4.6: YAC 二重系（Dual Duplex）のシステム運転状況

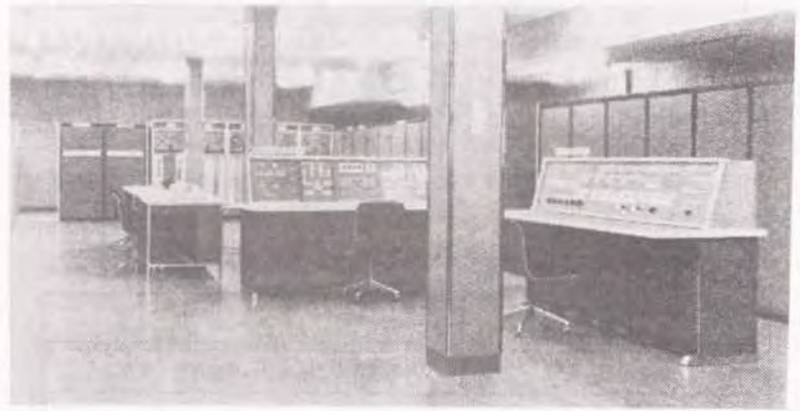


図 4.7: 郡山 YAC コンピュータシステム（一部）

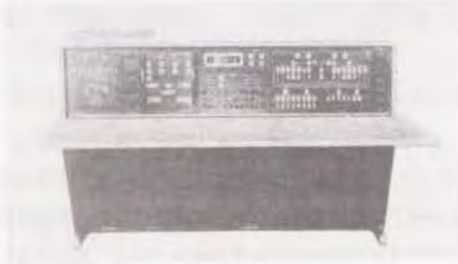


図 4.8: 郡山 YAC システムコンソール

4.4.3 郡山ＹＡＣ実現の上でのシステム構築技術

4.3.3 で述べたように、待機切替二重系が有効に働くためには、片系だけで独立した、エラー検知能力を備える必要がある。そのため郡山ＹＡＣではCPUおよび大きな構成要素たるプロセスＩＯに対して、第3章表3.1に示すごとく、ＳＷによるエラー検知を行ない、片系内に閉じてエラー検知を達成した。この状況を表4.1に要約して示す。

表 4.1: CPUおよびプロセスIOエラー検知方式比較

方　式	特　徴
両系出力照合	片系運転のときエラー検知能力を失う。
ハードウェアによる自己検知	片系内で検知できるが、検知回路のコストが大きい。ＣＰＵでは性能低下も大きい。
ソフトウェアによる自己検知	片系内で検知でき、かつコスト負担が小さい。時間分解能がハードウェア検知より粗い。

4.3.3 で指摘した他の点、すなわち、片系内で閉じて修理とその確認を行うことについては、郡山ＹＡＣでは以下に示すシステムチェッカというプラントシミュレータを開発し、これを系内に常設することにより解決した。図4.5の中央部にそれが示されている。

一般にＣＰＵとかメモリでエラーを検知した場合、その修理も確認も、片系内で独立に実施できるので、問題にはならない。問題はプラントの現場機器と膨大なインタフェースを持つプロセスＩＯ部分の修理と確認である。郡山ＹＡＣの場合、上述の片系内で閉じて、プロセスＩＯのエラー検知を行うため、現場からの反応信号をソフトウェアでチェックしているがため、プロセスＩＯの修理をし、確認をするときも実は同じように、現場との信号のやりとりでチェックしたい。ところが、そのときは、

もう片方の系が実運転を行っており、システムとしては稼働の最中である。それでは、システムが休憩時間に入るまで、何時間か待ち、プラントが休止開放されるのを待って修理すみのプロセスＩＯ動作の確認をやることも考えられる。しかし、それをやっていたのではＭＴＴＲが非常に長びいたのと同じことであり、4.3.2 式 (4.23) から判るごとく、システムＭＴＢＦが反比例して短縮する。

郡山ＹＡＣでは、修理すみのプロセスＩＯをもつ片系ＦＣＣをシステムチェッカに接続し、システムチェッカを擬似プラントと考えて信号の入出力を行ない、その反応をチェックし、動作確認する。システムチェッカは、小さなプラントシミュレータであり、現場の機器がＹＡＣからの指示信号を受けて反応するのと類似の反応を行なう。ただし、リターダに入った貨車の運動方程式を正確にシミュレートするのではなく、ＣＲ回路で単純な速度低下信号を作り出しているだけである。

図4.9にシステムチェッカの外観を示す。

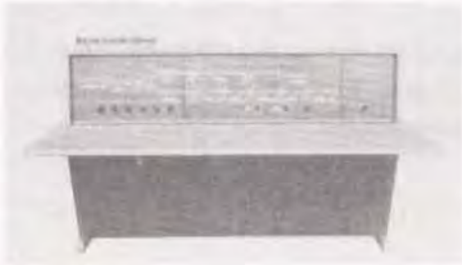


図 4.9: YAC システムチェッカ

4.5 郡山ＹＡＣ稼働実績

郡山ＹＡＣはクリティカルリアルタイム制御システムとして、我国で初めての本格的な独立型ホットスタンバイシステムによるコンピュータコントロールシステムで

あった。片系システムの大きさが当時の平均的コンピュータシステムの約５倍、両系合わせると約１０倍の巨大なシステムであった。真に二重系としての効果が出るのかどうか、注目の的であった。もし、真に二重系としての効果が出なければ、片系システムの巨大さが逆にハンディキャップとなり信頼性面の目標を達成しない恐れがあった。図4.10に４３年１０月に稼働開始してから８ケ月間の稼働実績および同期間のコンピュータのフォールトの状況を示す[29]。

稼働実績グラフから容易に判るように、３月、４月に稼働率が低下している。これは送電線の事故によりこの間に全停電が４回発生したことに起因している。。ＹＡＣでは、送電線事故もあらかじめ想定し、異なる２変電所２系統から受電していたが、それにも拘らず、両系ともに停電となってしまった。この教訓から、電源については、このあと第３のバックアップとしてエンジン発電機を備えた。稼働実績グラフの点線はこの電源系統の停電を除外して仮りに算出した稼働率である。

図4.10の下半分に示す障害件数は両系全合計である。これらの件数に対応するＭＴＴＲが記録されていないが、郡山ＹＡＣの場合、最長ダウンタイム１Hrという厳しい要求に対応し、テストプログラムの充実と保守要員教育を徹底したため平均0.5Ｈｒ程度は達成していた。従って、10月の障害44件に対応して22Ｈｒのダウンタイムがあったことになる。仮にこれを片系２２件で11Ｈｒづつあったとすると、片系の１０月１箇月間のMTBFは

$$片系MTBF = \frac{22Hr \times 31日}{22件} = 31Hr \tag{4.23}$$

となる。

これに対し、システム全体の実績稼働率は99.92％を示しており、この値は４３年１０月の稼働が

全体系システム稼働時間	約 ６８１．５ Hr
全体系システムフェィリャ時間 約	０．５ Hr （１回）
合計	６８２．０ Hr （２２Hr ×３１日）

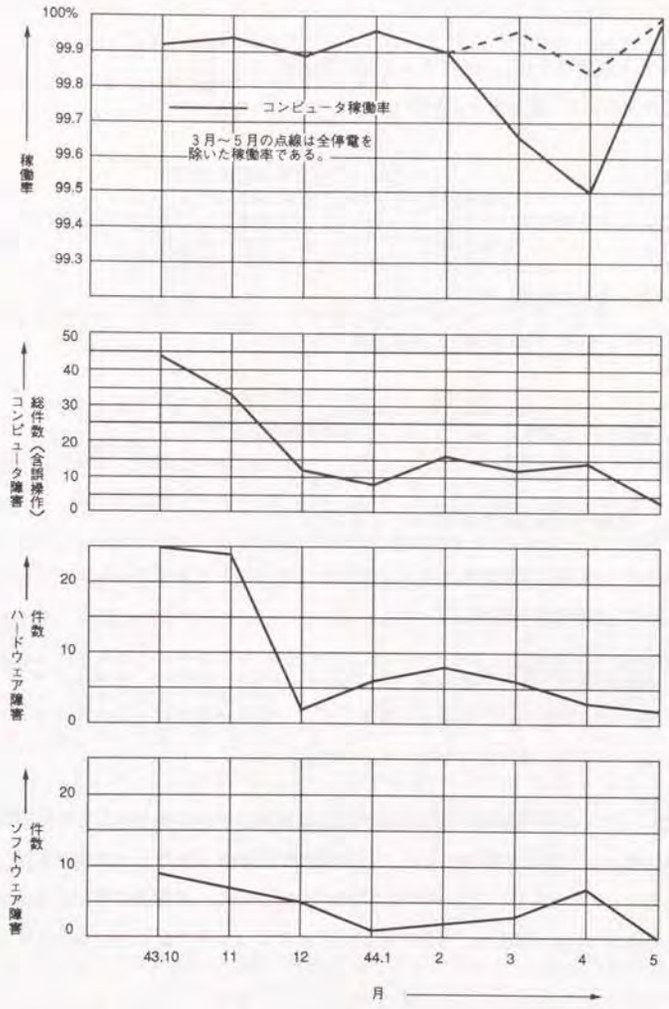


図 4.10: 郡山 YAC の稼働率と障害件数

であったと逆算できる。(MTTR ≅ 0.5Hr を仮定)

これからして、実績データとして、

$$\text{全体系MTBF} \cong 680Hr \tag{4.24}$$

$$\text{全体系MTTR} \cong 0.5Hr\text{（仮定）} \tag{4.25}$$

であったと考えられる。

これで、4.3.2 の式 (4.22) と対比させてみると、

$$\text{理論上の全体系MTBF} = \frac{1}{2}(\text{単系MTBF})^2 \cdot \left(\frac{1}{\text{単系MTTR}}\right) \cong 960Hr. \tag{4.26}$$

$$\text{実際の全体系MTBF} \cong 680Hr.$$

となり、実績値が理論値よりは少ないものの、約７０％の値となり、おおよそ期待通りの二重系効果を発揮している。

> [註] 片系ＭＴＢＦが３１ Hr である点は、当時のコンピュータとしてやや短いが、郡山 YAC の場合、片系だけで一般汎用機の約５倍の大きさがあったので、ほぼ妥当な値と考えられる。

ところが、同様の試算を障害件数が大巾に減少している４４年１月度に着目して実施してみると異なる様相となる。１月度の障害は図 4.10から、全体で８件、うちハードウェア障害６件、ソフトウェア障害（バグ）１件、誤操作１件である。１０月度と同様 MTTR = 0.5Hr、８件の障害から４件づつ片系で起こったものと仮定すると、

$$\text{片系MTBF} = \frac{22\text{Hr} \times 31\text{日}}{4\text{件}} \cong 170\text{Hr} \tag{4.27}$$

となる。

これに対し、全体系の実績稼働率は、９９．９６％を示している。これを MTTR = 0.5Hr として MTBF を逆算すると、

$$\text{全体系稼働時間} \quad 約681.7Hr \tag{4.28}$$

$$\text{全体系システムフェイリャ時間} \quad 約0.3Hr \quad (\text{ダウン回数換算0.6回}) \tag{4.29}$$

$$\text{全体系換算MTBF} \quad 約1130Hr \quad (682/0.6) \tag{4.30}$$

を得る。このとき、4.3.2 の式 (4.22) による理論上の MTBF を計算すると、

$$\text{理論上のMTBF} = \frac{1}{2} \cdot \frac{(\text{単系}MTBF)^2}{\text{単系}MTTR} = \frac{1}{2} \cdot \frac{(170)^2}{0.5} = 28900Hr \tag{4.31}$$

を得る。この場合、理論上の MTBF が極めて長いのに比し、実績 MTBF がそれ程伸びておらず、理論値の約１／２５にしか達していない。この理由は障害の内訳を見れば比較的容易に想像がつく。すなわち、誤操作１件が含まれていることが、理論と実績の乖離の原因と考えられる。なぜなら、誤操作とはシステムコンソールにおける誤操作であり、一般に直接システムフェイリャを起こしてしまうからである。かかるケースが起こりうることは第２章 表 2.1で示したが、システム構築の上の方式技術の問題としては取り上げてこなかった。しかし、少なくともこれがシステム全体へ与える影響の度合についてはここで評価を行なう。

ホットスタンバイシステムで正常運転（両系正常）しているときに、フォールトが起こると一般には片系運転へ遷移する。しかし、現実には或種のフォールトが起こると、片系運転にならず一挙に両系に影響を及ぼし、システム全体のフェイリャとなる場合がある。この現象を２通りに分類することができる。

(1) 片系で発生するフォールトは、普通のフォールトであり、本来なら他の片系運転へ切替わる。しかしながら、切替回路に何らかのフォールトが潜在しており、切替えが不成功となる。切替回路に潜在していたフォールトは、片系でフォールトが起こるまでは出現する機会がなかったから潜在していた。

(2) システムコンソールの誤操作に代表される全体システム中の一重系部分でフォールトが発生すると、システムに一挙にシステムフェイリャになる。

（1）の現象をモデル化するため、カバレッジの概念が提唱されている [21]。その考え方を図4.11に示す。

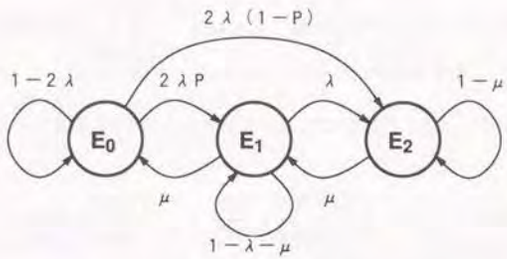


図 4.11: カバレッジ概念を入れた遷移図

図4.11に示す確率Pがうまく切替えが成功する確率で、これがカバレッジである。上記（1）の現象はこのモデルでうまく表現できるが、（2）の現象は異なるモデル化が必要である。何故なら、（1）では一次的に生起するフォールトはあくまで従来と同じで故障率λであるが、そのとき切替回路又はその等価部分に切替失敗を招くフォールトが潜在しているかいないかを問題にしている。他方（2）では、切替回路の潜在フォールトとは全く無関係に、別の種類のフォールト、すなわち両系フェイリャを一挙に引き起こしてしまうフォールトが発生する。システムコンソール誤操作がその代表例である。この場合、発生率自体がλとは全く別である。更に、この種のフェイリャの場合は、原因が単純な場合が殆んどであり、修理率もμとは全く別である。（修理が早い）そこで、かかる現象をモデル化すると図4.12となる。

図4.12において、E_0、E_1、E_2およびその間の遷移は図4.4に同じである。E_3を新しく追加した。

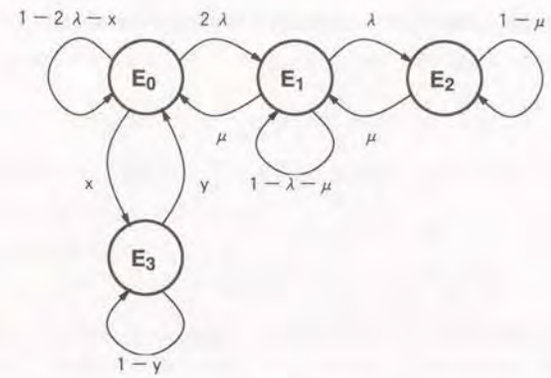


図 4.12: 誤操作又はそれと等価なフォールトを入れた場合の
ホットスタンバイシステムのマルコフ過程遷移図

E_3　　：両系共通部フォールト（主として誤操作）システムフェイリャ
$E_0 \rightarrow E_3$　：共通部のフォールト発生率xにより一挙にE_3へ移行。
$E_3 \rightarrow E_0$　：共通部のフォールト修理率yにより一挙にE_0へ移行。

図4.12より、遷移確率行列Pは、次の如くなる。

$$P = \begin{pmatrix} E_{00} & E_{01} & E_{02} & E_{03} \\ E_{10} & E_{11} & E_{12} & E_{13} \\ E_{20} & E_{21} & E_{22} & E_{23} \\ E_{30} & E_{31} & E_{32} & E_{33} \end{pmatrix} = \begin{pmatrix} 1-2\lambda-x & 2\lambda & 0 & x \\ \mu & 1-\lambda-\mu & \lambda & 0 \\ 0 & \mu & 1-\mu & 0 \\ y & 0 & 0 & 1-y \end{pmatrix} \tag{4.32}$$

平衡状態となったときのE_0、E_1、E_2、E_3の極限確率をP_0、P_1、P_2、P_3とすると、

$$(P_0 \ P_1 \ P_2 \ P_3)P = (P_0 \ P_1 \ P_2 \ P_3) \tag{4.33}$$

$$P_0 + P_1 + P_2 + P_3 = 1 \tag{4.34}$$

が成立する。前回と同様 (4.32)(4.33) (4.34) を解くと以下の結果を得る。

$$P_0 = \frac{1}{K}\cdot\mu^2 y \tag{4.35}$$

$$P_1 = \frac{1}{K}\cdot 2\lambda\mu y \tag{4.36}$$

$$P_2 = \frac{1}{K}\cdot 2\lambda^2 y \tag{4.37}$$

$$P_3 = \frac{1}{K}\cdot\mu^2 x \tag{4.38}$$

$$\text{ここに、}K = \mu^2 y + 2\lambda\mu y + 2\lambda^2 y + \mu^2 x \tag{4.39}$$

システム全体の MTBF、MTTR の比は、平衡確率の比に等しいから次の式が成り立つ。

$$\begin{aligned} MTBF : MTTR &= (P_0+P_1) : (P_2+P_3) \\ &= (\mu^2 y + 2\lambda\mu y) : (2\lambda^2 y + \mu^2 x) \end{aligned} \tag{4.40}$$

このシステム全体の MTTR は、P_2 の $MTTR=\frac{1}{\mu}$ と P_3 の $MTTR=\frac{1}{y}$ との加重平均となるから、

$$MTTR = \frac{P_2}{P_2+P_3}\cdot\frac{1}{\mu}+\frac{P_3}{P_2+P_3}\cdot\frac{1}{y} = \frac{2\lambda^2 y^2+\mu^3 x}{(2\lambda^2 y+\mu^2 x)\mu y} \tag{4.41}$$

よって

$$MTBF = \frac{(P_0+P_1)}{(P_2+P_3)}\cdot MTTR = \frac{(\mu+2\lambda)(2\lambda^2 y^2+\mu^3 x)}{(2\lambda^2 y+\mu^2 x)^2} \tag{4.42}$$

を得る。式 (4.42) において、$x=0$, $y=1$ とすると、図 4.4と同じ場合となり、式 (4.42) も $\frac{(\mu+2\lambda)2\lambda^2}{4\lambda^4} = \frac{\mu+2\lambda}{2\lambda^2}$ となり、式 (4.21) に帰着する。

図 4.12、或は式 (4.42) がどの程度郡山ＹＡＣの４９年１月度の実績を説明しうるのかを示す。

ここで、

$$\lambda = 6\cdot 10^{-3} \quad \text{片系}MTBF = \frac{1}{\lambda} \cong 170\text{Hr} \tag{4.43}$$

$$\mu = 2 \quad \text{片系}MTTR = \frac{1}{\mu} \cong 0.5\text{Hr} \tag{4.44}$$

とする。これらの値は１月度の実績に近い値である。

ここで、誤操作によるシステムフェイリヤ E_3 を考慮しない場合の MTBF は、

$$\frac{\mu+2\lambda}{2\lambda^2} \cong \frac{2}{2\cdot 36}\cdot 10^6 \cong 28000Hr \tag{4.45}$$

となるが、誤操作率 x、誤操作フェイリヤからの修理率 y の値を仮定すると、これがどう変化するのかを式 (4.42) をもとに示す。

(4.42) に (4.43)(4.44) を代入すると

$$\frac{(\mu+2\lambda)(2\lambda^2 y^2+\mu^3 x)}{(2\lambda^2 y+\mu^2 x)^2} = \frac{x+9\cdot 10^{-6}\cdot y^2}{(x+18\cdot 10^{-6}\cdot y)^2} \tag{4.46}$$

となる。ここで、誤操作によるシステムフェイリヤからの修理率 y は、一般のフォールトからの修理率より５倍良好（修理時間が１／５）と仮定すると

$$y = 10 \tag{4.47}$$

となり、これにより式 (4.46) は

$$\frac{x+9\cdot 10^{-6}\cdot y^2}{(x+18\cdot 10^{-6}\cdot y)^2} = \frac{x+9\cdot 10^{-4}}{(x+18\cdot 10^{-5})^2} \tag{4.48}$$

となる。式 (4.48) において、x の値を λ に比較して 1/20、1/10、1/6、1/3 とした場合の値は次の如くなる。

$$x = \frac{1}{20}\lambda = 3\cdot 10^{-4} \quad \text{式}(4.48)\cong 5200Hr \tag{4.49}$$

$$x = \frac{1}{10}\lambda = 6\cdot 10^{-4} \quad \text{式}(4.48)\cong 2500Hr \tag{4.50}$$

$$x = \frac{1}{6}\lambda = 1\cdot 10^{-3} \quad \text{式}(4.48)\cong 1370Hr \tag{4.51}$$

$$x = \frac{1}{3}\lambda = 2\cdot 10^{-3} \quad \text{式}(4.48)\cong 610Hr \tag{4.52}$$

x=0 とした場合は、式 (4.45) となる。(4.45) (4.49) (4.50) (4.51) (4.52) を片対数グラフにプロットすると図 4.13となる。

大雑把に言うと、誤操作型フォールトが片系独立フォールトの 1/10 程度あるときは、システム全体の MTBF は誤操作型フォールトが零のときの 1/10 程度に落ちる（28000 → 2500）。

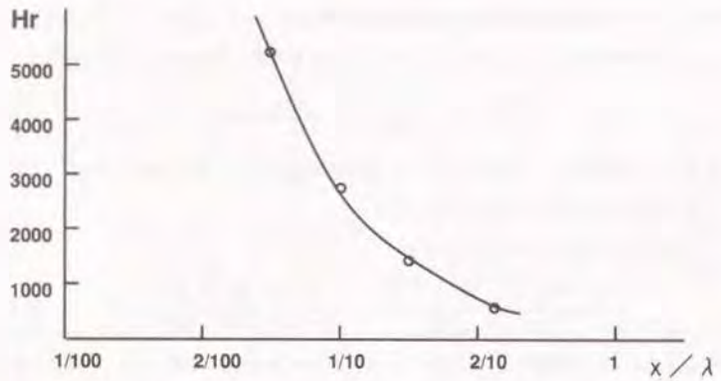


図 4.13: システムコンソール誤操作（又は等価フォールト）による
ホットスタンバイ系の MTBF への影響

郡山ＹＡＣにおける４４年１月度のフォールトは８件中１件が誤操作であったから、一般のフォールト７件に対し、誤操作１件であり、フォールトの発生率は 1/7 と見ることができる。この場合は上例における x/ λ =1/6 の場合に近く、x/ λ =1/6 の場合の MTBF モデル計算例、1370Hr に対し、実績 MTBF（換算値）1130Hr であり、良好な一致を示す。

従って、実際のホットスタンバイシステムを構築する場合の MTBF 値の評価を行なうとき、両系が真に独立ということはあり得ず、若干の誤操作率によるシステムフェイリャを覚悟すべきであり、その場合は式

$$MTBF = \frac{\mu + 2\lambda}{2\lambda^2} \tag{4.21}$$

よりも

$$MTBF = \frac{(\mu + 2\lambda)(2\lambda^2 y^2)}{(2\lambda^2 y + \mu^2 x)^2} \tag{4.42}$$

を用いるべきである。

4.6 3重多数決システム

第２章 表2.4に示したように、多数決方式は一つの系が故障しても残っている正常系が２つ以上ある。従って、故障した系の特定が直ちに可能であり、その系を切離すだけで全体系は正常運転を続行できる。CPU をフォールトトレラント構成にするには、理想的な機能を持つ。しかしながら、最低限３台を要するためにハードウェアのコスト上の負担が大きく、商用機としてはなかなか実現しなかった。しかし、近年、マイクロプロセッサの発展により、コスト上の負担が少なくなり、1.2.2 に示した商用機事例が出はじめた。マイクロプロセッサの発展が、多数決システムの実用化に密接に関連していると言える。本節では、３重多数決システムについて論ずるとともに、システム構築技術としては、マイクロプロセッサを使う場合に焦点をあてる。

4.6.1 3重多数決による運転の基本方式

３重化したユニットを並列に同期運転し、その各々の出力を常時観測し、不一致の場合は２対１の多数決により２の方を正として選ぶ。この原理は、1.2.1 に述べたように、古くから提唱されており、次の図4.14に示す。

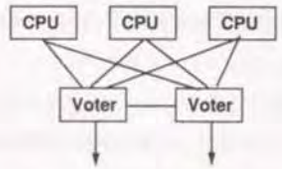


図 4.14: ３重多数決方式

３重化する範囲は、現状では図に示すようにマイクロプロセッサをベースとするCPU とするのが妥当と考える。図の CPU 内にキャッシュメモリが有るケースとないケースがありうる。コンピュータサブシステムを３重化することは、まだコスト上の負担が大きいこと、および保守交換単位が大き過ぎて回復処理が複雑化することから有利ではない。

CPU 以外の残りの構成ユニット（バス、電源、メモリ、入出力チャネル等）については、２重系を構成する。多数決を行なう Voter もこれに従って、２重化すると共に相手 Voter と照合する機能をも内蔵させる。

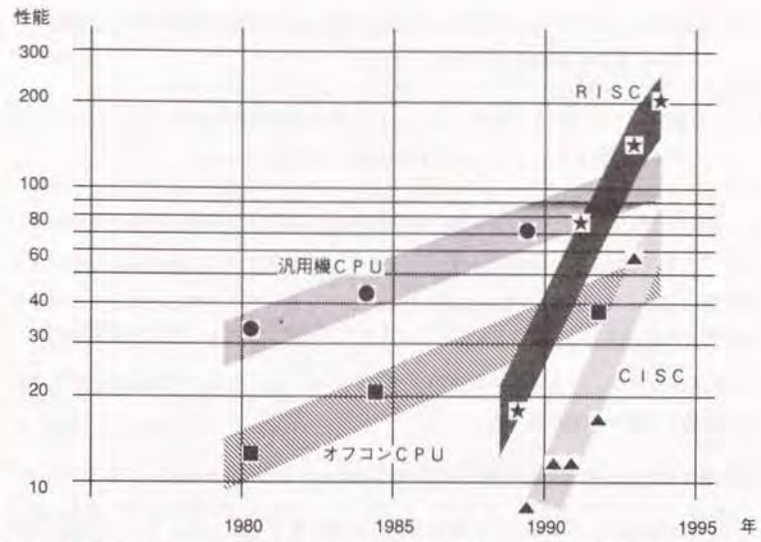
３台の CPU のうち、１台にフォールトが発生し、その結果その CPU から他の２台と異なる信号が出たとき、Voter はこれを検知し、その CPU を切離す。全体系は残り２台で正常運転を続行するが、保守員は１台の CPU が切離された時点で報告を受け、修理交換作業を開始する。交換単位が CPU １台であれば、切離されている CPU を交換し、並列同期をとりなおして３台でスタートする。このとき、運転を継続していた２台の CPU は、作業の切れ目でキャッシュメモリのフラッシュを含むイニシャライズを行なうこととなる。

4.6.2 マイクロプロセッサの発達とフォールトトレラントシステムへの利用

近年、とりわけ１９９０年代に入って RISC マイクロプロセッサの発達が顕著である。CPU 演算性能の面では、汎用大型機の CPU の性能を凌駕するものも出現してきた。この状況を図 4.15に示す。

マイクロプロセッサの発達は、直接的には LSI 技術の発達に依るが、他に見落とせない要因がある。それは、或１つのマイクロプロセッサは必ず２社以上が同一アーキテクチャのものを製造販売しており、かつ、マイクロプロセッサチップ単体の形で大量に市販され、同一アーキテクチャのもの同志は勿論のこと、異なるアーキテクチャのものの間でも激しい市場競争が展開されているということである。その意味では、本論文ではこれらを市販マイクロプロセッサと呼ぶ。これに対し、特別な仕様のもとに特別の用途に限って開発、製造されるマイクロプロセッサもあるが、量の面で市販マイクロプロセッサとは比較にならず、従って価格の面でも比較にならない。又、性能の面でも激しい市場競争の下にある市販マイクロプロセッサが常に業界トップ性能を争っている。従って、フォールトトレラントシステム構築の上で対象としてとりあげてゆくのは市販マイクロプロセッサである。

コストと性能の面で圧倒的に有利な市販マイクロプロセッサも、フォールトトレ



註１．汎用機、オフコンは、上記の他にＩＯ性能、ＲＡＳ性能、ＳＷ蓄積等の別次元の指標で優れており、トータル機能として、この図のみで総合判断をすることはできない。
(RAS: Reliability, Availability, Serviceability)

図 4.15: 各種 CPU の性能発展

ラントコンピュータの観点からすると、１点だけ問題になる点がある。それは、チップ内のエラー検知能力が低い、ということである。市販マイクロプロセッサは元来パソコン、ワークステーション用のＣＰＵとして開発され、販売されている。したがって、これらの用途には、チップ一石、それ自身の信頼性で十二分に実用品質を達成している。それゆえに、市販マイクロプロセッサチップには、それ以上の余計な内部エラー検知回路は入っていない。

しかし、フォールトトレラントコンピュータは、その任務上、パソコン、ワークステーションよりも２桁程度は高い信頼性を要求されるので、ワンチップといえども、その内部エラーを検知できる方が望ましい。

この解決策としては、２通りのやり方がある。

（1） 市販マイクロプロセッサのかわりに、内部にエラー検知機能を備えた特殊マイクロプロセッサを開発する [32] ～ [34]。

（2） 市販マイクロプロセッサをパラレルに2箇以上同時同一動作させ、常時その出力信号を比較することにより、エラー検知する [35] ～ [38]。

前者の方法は、マイクロプロセッサの性能を 1/2 程度に低下させる恐れがあり、さらに大量市販されているが故の安価さをも失う。数量が少ない場合は、開発コストが単価に占める率が非常に大きくなるので、コストとしては一挙に１０００倍以上はね上がる恐れが大きい。

後者の方法は、コスト的に数倍の負担となるが、元々のチップの安価さからすれば十分実用可能レベルに収まる。

4.6.3 オープン指向フォールトトレラントコンピュータ

ここ数年来、コンピュータの構成方式に極めて大きな変化が起こっている。それはオープン性である。

コンピュータの構成をオープンにする、ということは、パソコン、ワークステーションの世界から始まった。すなわち、

- 市販流通応用ソフトウェアパッケージ
- 市場標準のミドルウェア（データベース、ネットワーク等）
- 市場標準のオペレーティングシステム
- 市場標準のマイクロプロセッサ

という構成をとる。各レイヤーの間のインタフェースが市場標準となっているため、それに合致するものであれば、色々なベンダーのものをセレクトし、組合わせて構成できる。

リアルタイムコンピュータシステムの世界においても、かかるオープン化のニーズが急速に増大してきている。その理由は、

- 安くて良い市販品を組合わせて使いたい。
- オープンな事務処理システム、ＣＡＤシステム等と密に連携してデータの交換をしたい。
- オープン性ある市販ワークステーション上で、リアルタイムシステムの応用ソフトの開発を行いたい。その結果が、ほぼそのままリアルタイムコンピュータ上で走ってほしい。
- 稼働状況の表示とか統計処理とかには、市販ソフトウェアをそのまま利用したい。

ということにある。

この結果として、今後のリアルコンピュータのプラットフォームに要求される仕様は次のようになってきている。

- オペレーティングシステムの核（資源管理、ジョブ管理）については、リアルタイムオペレーティングシステム本来の高速レスポンス性、レスポンスタイム有限性、をもつこと。
- しかしオペレーティングシステム外殻のＡＰＩについては、リアルタイムＵＮＩＸインタフェースを持つこと。（POSIX １００３．４Ａ）
- このインタフェース規格の将来のバージョンアップに迅速に追従すること。
- ハードウェアとしては、市販、市場標準のマイクロプロセッサをベースとし、高信頼性、フォールトトレラント性は、その上に付加する形で実現すること。

前節で述べた市販マイクロプロセッサを使用し、かつ ここでで述べたオープン指向なフォールトトレラントコンピュータが今後の一般的方向である。とすると、フォールトトレラントとするために必要な技術アイテムと、オープンであるために必要な技術アイテムを、マイクロプロセッサと標準オペレーティングシステムとで矛盾

や排反なく実現できるかどうかが課題である。

4.6.4 予防引継ぎ方式

筆者は、4.6.2に論じた如く、市販マイクロプロセッサによりフォールトトレラントコンピュータを実現し、かつ4.6.3で論じたように標準オペレーティングシステムをほとんどそのまま使用してオープン性を持たせ、市販のミドルウェアやアプリケーションソフトを組みこめる形にすることがこれからのフォールトトレラントコンピュータにとって極めて重要であると考えた。

ここで筆者等は、冗長系による引継ぎに関して全く新しいコンセプトすなわち予防引継ぎという考え方を考案し、これが筆者の上記目的に極めて良好に適合することを机上検証した。なお現在日本および米国に特許出願中である。以下に予防引継ぎのコンセプトとその評価検討結果を述べる。

(1) システム構成

図4.16にシステム構成を示す。この図は3箇のＢＰＵ（basic processing unit）A、B、Cを含んでいる。A、B、Cは各々内部に3個のプロセッサPEを持つ。A、B、Cそれぞれがリプレーサブルユニット（ボード）である[39][40]。システムの負荷に応じて3箇をもっと増やすことも可能であり、その制約はオペレーティングシステムが扱えるＢＰＵ箇数で抑えられるし、場合によってはメモリバスの物理的な長さや電源容量で抑えられる。

(2) システムの動作

動作の要点を以下に示す。

- 各ＢＰＵは各々非同期で独立に動作する。
- オペレーティングシステムは説明上、全体に一式共通とする。すなわち、タイトカップル、シンメトリックマルチプロセッサを用いるものとする。

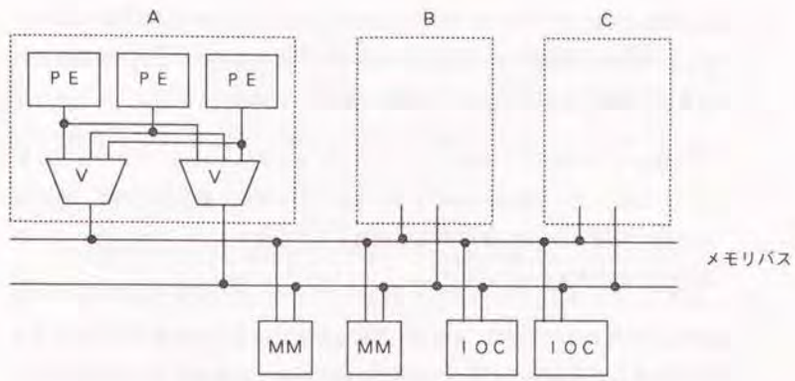


図 4.16: 予防引継ぎマルチプロセッサ
（タイトリカップル・シンメトリック・マルチプロセッサ）

（本件は、予防切替実現のための必須要件ではなく、説明上の設定である。）

- 各ＢＰＵは各々内部に3ヶのＰＥ（processing element）を持つ。
- 3ヶのＰＥはクロックレベルで同期運転、同一動作を行なう。
- 3ヶのＰＥは各々キャッシュメモリを内蔵している。
- 3ヶのＰＥのメモリバスへの出力信号はクロックレベルで2 out of 3多数決比較を行なう。（V：Voter）
- 3ヶのＰＥのうち、どれか1ヶのＰＥにフォールトが発生すると、そのＰＥの出力信号に異常が現れた時点で2 out of 3回路により検知される。この場合、多数決により、出力一致している2箇のＰＥを正常と見なし、異なる1箇のＰＥを異常と見なす。ＢＰＵからメモリバスへ出す信号は正常と見なす2箇のＰＥのものを採用し、異なる1箇のＰＥは遮断する。

- 結果的に１箇のＰＥにフォールトが発生したＢＰＵは表面的には異常を起こさず、正常運転を続行する。しかし内部的には３箇のＰＥのうち、２箇は健全なるも、１箇は脱落し、全体的には健康状態が低下したとも言える。

- この時点で、そのＢＰＵはオペレーティングシステムに対し、アラーム信号を出す。このアラーム信号の意味は「私はまだ正常運転を続行しうるが、内部的に半健全状態（あとでこれを準正常状態として定義する。）となったので、予防的に引継ぎ準備をはじめてほしい。」ということである。

- オペレーティングシステムはアラーム信号を受取ると、該当ＢＰＵの現行プロセスはそのまま継続させるが、次の新しいプロセスを該当ＢＰＵには割り当てないようにする。又、サーバプロセスのような常駐プロセスは、元来他のBPUにも乗っているので、該当BPUの常駐プロセスは現在やっているサービスの切れ目で終了させる。

  例外として、該当BPU以外に正常運転できるBPUが存在しない場合は、やむを得ずＯＳは該当BPUに新しいプロセスを与える。

- 結果として、該当ＢＰＵは現行プロセスを終了したあとアイドル状態となる。そこでこのＢＰＵはキャッシュメモリを主メモリへライトバックし、第二の信号をオペレーティングシステムへ送る。

  第二の信号の意味は「私はいつでも交換可能な状態だ。」ということである。

- オペレーティングシステムを経由して、交換要求を受けた操作員は、該当ＢＰＵを良品に交換する。このとき他のＢＰＵは運転を続行している。新しく投入されたＢＰＵはセルフテストのあと、オペレーティングシステムに対しタスク受入ＯＫの信号を発し、システムは完全に元の正常状態へ復帰する。

- 図4.16中、主メモリ（ＭＭ）はＥＣＣ冗長コードチェック、バスはパリティチェックおよびタイミングチェックされており、ＩＯＣチップについては図示はしていないがA、B、Cの中のＰＥと同じ2 out of 3チェックしている。

以上に動作の要点を示したが、特長となるポイントは、BPUの３個のPEのうち、１個が故障し、２個はまだ健全であり、正常動作を行なえるにも拘らず、早くもその時点で引継ぎ要求のためのアラームを出すという点である。そして、該当BPUが正常動作しつつ、少しづつ（新しい業務を）引継ぎを行なうのが特長である。

4.6.5 準正常状態

予防引継ぎの動作を前節に説明したが、その最大の特長は、オペレーティングシステムは時間余裕をもって、ゆっくりプロセス引継ぎできるということである。このことはアラーム信号を発生するタイミングに関して、或るＢＰＵがダウンしてから発生するのではなく、まだ十分動作可能な段階で予防的に発生するということで実現している。別の面からいえば、従来のシステムはその構成ユニットの正常性については、正常状態か、故障状態かの２つのステータスしか考えておらず、正常状態から故障状態へ移った時にアラームが出ていた。予防引継ぎシステムにおいては、ユニットの状態は

- 正常状態
- 準正常状態
- 故障状態

の３つのステータスを考えていることとなる。アラームは準正常状態に移ったときに発生する。ユニットは短期間の準正常状態ののち予防的に交換されるので普通は故障状態には落ちない。この考え方を図4.17、図4.18に示す。

準正常状態とは、筆者がここで命名し、提唱する新しい概念である。新しい概念、準正常状態は次のように定義する。

あるユニットが準正常状態にある、ということは次の状態を指す。

（1） あるユニットが部分的欠陥を内包しており、

（2） そのユニットは部分的欠陥が内在していることを認識しており、

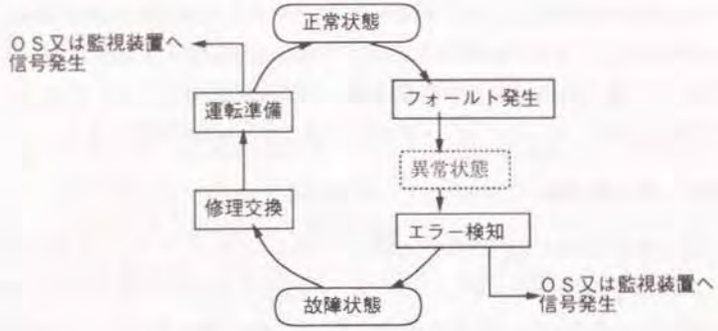


図 4.17: 従来のユニットの状態移行概念図

（3） そのユニットは適当な手段により部分的欠陥による影響を外部に出さないようにしており、

（4） そのユニットは「自分が現在部分欠陥内在状態にある。」という状態情報を外部へ送出している。

準正常状態にあるユニットの外部に対する挙動は次のようになる。

（1） 該ユニットは、次の（2）に述べる例外を除いて、正常状態にあるユニットと全く同じ動作／反応、を外部に対して行なう。

（2） 例外として、該ユニットは外部に向かって、「自分は現在、準正常状態にある。」という状態情報を出し、また外部から状態について問合わせを受けた場合、「現在準正常状態にある。」との応答を返す。

準正常状態にあるユニットが状態変化を起こすと次にように他の状態へ遷移する。

（1） 該ユニットに内在する部分欠陥が何らかの方法により除去され（修理、あるいは自然回復）、ユニットが自己診断の結果、欠陥の消滅を確認すると正常状態

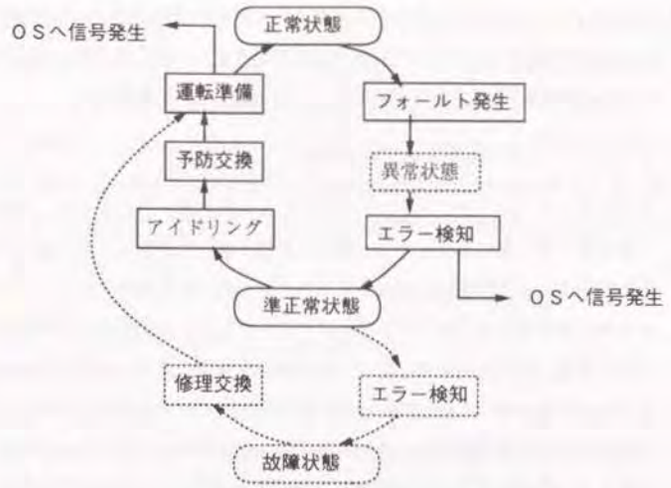


図 4.18: 準正常状態への状態移行概念図

へ遷移する。

（2） ユニットに部分欠陥が内在したまま更に欠陥が拡大し、ユニット全体として欠陥による影響を外部へ出ないように抑制し得なくなると、故障状態へ遷移する。

筆者が提案する予防引継ぎ方式は、1つのユニットについて見ると図 4.18の正常状態と準正常状態との間を往復するものであり、準正常状態を長時間放置しなければ確率的に故障状態へは落ちない

4.6.6 予防引継ぎマルチプロセッサの特徴

予防引継ぎマルチプロセッサの動作を 4.6.4 で示したが、この動作によりフォールトトレラントコンピュータとしては、オペレーティングシステムの負担が軽微、最大速度による走行、円滑なリカバリ、経済性といった利点があるが、一方、一時的なシ

ステム能力低下といった欠点も存在する。以下、それらの項目について市販マイクロプロセッサを多重使用し、ＯＳへの負担が軽微なペア＆スペア方式を用いているストラタス社 XA-2000[32] と対比しつつ論ずる。(XA-2000 は図 4.21参照)

（1） オペレーティングシステムの負担が軽微

ＢＰＵからオペレーティングシステムに伝えられるアラーム信号は、該当ＢＰＵが故障状態に落ちたときでなく準正常状態へ移ったときである。従って該当ＢＰＵのプロセスが突然不測のタイミングで中断されたわけでなく、プロセスは正常に遂行される。従ってオペレーティングシステムは突然のプロセス中断という事態に迫られたのではなく、単に該当ＢＰＵに新しいプロセスを供給することを中断すればよい。これは単に該当ＢＰＵのプロセスキューテーブルから新しいプロセスを抹消し、それを他のＢＰＵのプロセスキューテーブルへ移せばよい。これはプロセス実行途中の切替えではなく、まだ実行されていない待ち状態のプロセスの割り当て変更であり極めて単純な作業である。

更に、実行途中のプロセス切替えが発生しないため、オペレーティングシステムは予めそれに備えてリスタート用チェックポイントリザーブを常時行う負担から解放される。

従って、市販又は既存標準商用機のＯＳを使うことが可能となる。

この状況を表 4.2に要約する。

表 4.2: オペレーティングシステムの負担の比較

	故障包含ＢＰＵ	冗長ＢＰＵ
従来型引継ぎ	プロセスを実行途中で中断する。	途中まで実行し中断したプロセスを引継ぐ。
予防引継ぎ	実行中のプロセスはそのまま完了する。	待ち状態のプロセスを引き取る。

（2） 最大速度による走行

予防引継ぎマルチプロセッサ方式では、各ＢＰＵは相互に独立非同期の走行を行う。すなわち相互にクロック同期をとる必要はない。クロック同期しているのは各ＢＰＵの内部である。ストラタスＸＡ－２０００のように、クロック同期運転を行う場合は、メモリバス上に共通クロックを流す必要がある。かかるバス上にクロックを走らせると、バスの物理的長さのためクロック速度が若干低下する。

（3） 円滑なリカバリー

準正常状態になったＢＰＵがアイドルとなり、キャッシュをライトバックしたあと完全正常ＢＰＵと交換される。このとき他のＢＰＵは運転を続行している最中であるからボードの交換はそれら他のＢＰＵの運転に何ら支障を与えずに実施できるよう予め突入電流防止等の手当をしておく必要がある。このことは活線挿抜ということで一般化している技術である [41] ～ [43]。

さて、かかるボード交換により新しいＢＰＵがシステムに投入されたとき、ストラタスＸＡ—２０００の如く同期運転を行う場合は対となる運転中のＢＰＵがそのキャッシュを一旦フラッシュ（クリヤ）して新規投入ＢＰＵの初期状態にあるキャッシュに歩調を合わせてやる必要が生ずる。すなわち新規ＢＰＵに歩調を合わせる為、運転中のＢＰＵが一旦ストップしキャッシュをクリヤし新規にスタートし直す。

従って、運転続行ＢＰＵは、その後若干の時間キャッシュミスヒットのための速度低下を起こす。予防引継ぎマルチプロセッサの場合は、各ＢＰＵは全く独立に動いているからかかる同期のためのロスは何らなく、他のＢＰＵへの何らの影響なく新ＢＰＵはスタートする。

（4） 経済性

ストラタスＸＡ—２０００の場合、９９％以上を占める正常運転している時間帯では合計４台のＰＥが同じ動作を行っており、外部から見ると１台のアウト

プットしか得られない。これはフォールトトレラントを達成するためのコストである。

他方、予防引継ぎマルチプロセッサの場合、正常運転している間は合計3台のＰＥが同じ動作を行っており、1台分の動きしかしていない。これも又フォールトトレラントを達成するためのコストであるがストラタスの4台よりは1台少なくてすむ。

（5） バグに依るエラーには無力

ハードウェアバグが顕在化して何らかのエラーを起こしたとき、クロック同期運転している3台のＰＥに同じエラーを起こすので、多数決検知にはかからない。又ソフトウェアバグが顕在化したときは、そもそもＣＰＵ機械語命令レベルでは何もエラーを発生しないから、当然多数決検知にもかからない。かかる事情はストラタス ペア＆スペアにも全く同様にあてはまる。バグに対しては、3.2.6 に論じたように、別途ソフトウェアを用いて知識処理的なチェックを行なうべきである。

（6） 一時的なシステム能力低下

ストラタスＸＡ―２０００の場合、1台のＢＰＵでフォールトが起こってもスペアＢＰＵが動いている限り、システム能力は何ら低下しない。他方、予防切替マルチプロセッサの場合、例えば図4.16でＢＰＵ Ａでフォールトが発生すると、オペレーティングシステムは新しいプロセスをＢＰＵ ＢとＣに再配分するからシステム全体能力はこの場合２／３に低下する。これは、Ａが良品に交換されれば終了するが、人間による保守作業を伴うため５分～１０分程度、２／３の低下状態が持続することも考えられる。

（7） ＢＰＵ故障の危険

或るＢＰＵが準正常状態となってアラームを発生したとき、何らかの都合で操作員に伝達されず長期間そのまま放置されると、該当ＢＰＵにおいて、残存している２箇の正常ＰＥのうち更にもう１箇がフォールトを発生する危険が理論

的にはある。ただし、実際には放置されている間は、該当ＢＰＵはアイドル状態になっているため、その BPU が故障したところでシステムフェイリャになることはない。システム全体の負荷オーバーが続くという結果になる。

同様なシチュエーション（放置）では、ストラタスの場合の方が危険が高い。すなわちペアとして残って運転している正常ＢＰＵ側の２箇のＰＥのうち１箇が故障するとシステムフェイリャとなる。次節でこの定量的な評価を行なう。

4.7 予防引継ぎシステムの信頼性の評価

前節までに提示した予防引継ぎシステムの信頼性を定量的に評価し、類似度（ハードウェアによるエラー検知、ＯＳ負担の少ない引継ぎ）があるストラタス社 XA-2000 のそれと比較を行なう。

予防引継ぎシステムは、フレキシブルなシステム構成をとりうるが、定量比較するため、その構成を図 4.19のように最小構成ＢＰＵ２台（ＰＥ合計６台）に固定して考える。

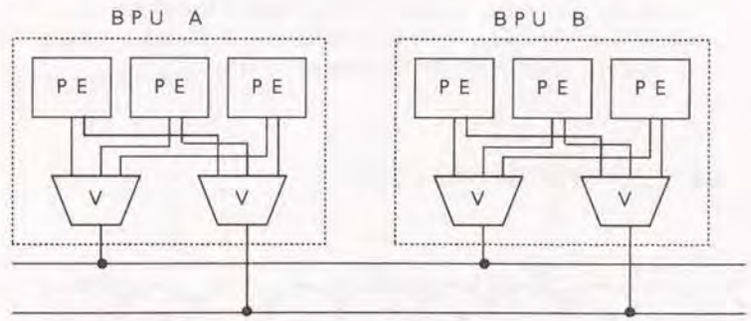


図 4.19: 予防引継ぎシステム（最小構成）

このときシステムがとりうる状態として、４種類を考える。それを表4.3に示す。

表 4.3: 予防引継ぎシステム状態表

状態名	ＢＰＵ状態	負荷状態（又は作業状態）	ＰＥ状態
E_0	正常　正常	リアルタイム負荷　バックグラウンド負荷	6台全部正常
E_1	正常　(準正常の後)修理中	リアルタイム負荷　修理作業	6台のＰＥの中で1台がエラー
E_2	準正常　(準正常の後)修理中	リアルタイム負荷　修理作業	片ＢＰＵ修理中に他ＢＰＵの3台のＰＥの1台がエラー
E_3	(準正常の後)故障修理待ち　(準正常の後)修理中	修理待ち　修理作業	片ＢＰＵ修理中に準正常運転していたＢＰＵで2台目ＰＥがエラー

仮定：

- 保守員は1名とする。
- $E_2 \rightarrow E_3$ となったとき、先に修理中の BPU を急遽準正常状態を修理しきらぬうちにラインへ戻し、片系準正常とし運転することはやらぬ。
- 修理中に、同じ BPU 内の別のＰＥが故障することはない。

マルコフ過程遷移図は図 4.20のようになる。

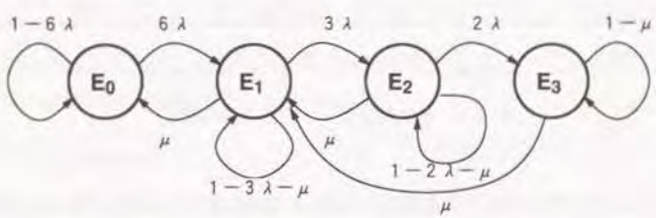


図 4.20: 予防引継ぎシステム（最小構成）の状態遷移図

$E_0 \rightarrow E_1$ ：6個のＰＥのいづれか故障により移行する。

E_1 ：片系正常、片系正常で両系運転可なるも予防引継ぎのコンセプトから準正常BPUは短時間に負荷をおとし、修理作業に入る。

$E_1 \rightarrow E_2$ ：正常BPUの3個のＰＥのいづれか故障により移行する。

E_2 ：先に準正常経由修理に入ったBPUは、そのまま修理継続。後で準正常となったBPUは、他に負荷を移すべき冗長BPUが存在しないため、準正常のまま運転続行。

$E_2 \rightarrow E_3$ ：準正常で運転していたBPUの2個のＰＥのうちいづれか故障により移行。このとき、先に準正常経由修理に入っているBPUを急遽そのままライン復帰させることはしないこととする。

E_3 ：片BPUは準正常経由修理中、他BPUはPE2台故障で修理待ち。

$E_3 \rightarrow E_1$ ：片BPUが修理完。ライン復帰。
他BPUはPE2台故障に対し修理中。このとき、$E_0 \rightarrow E_1$の移行時のE_1とは修理対象がＰＥ1個かＰＥ2個かによって異なるが、現実にはボード交換と診断プログラム確認作業であり修理作業、修理率とも、故障ＰＥが1個であろうと2個であろうと変らない。よって、状態として区別せず両方ともE_1として扱う。

$E_2 \rightarrow E_1$ ：修理中の1個のＰＥの修理完で、片系完全正常へ復帰。

$E_1 \rightarrow E_0$ ：修理中の1個のＰＥの修理完で両BPUとも正常へ復帰。

遷移確率行列Ｐは次のようになる。但し、λはＰＥ1個の故障率、μは BPU ボード1枚の修理率である。

$$P = \begin{pmatrix} 1-6\lambda & 6\lambda & 0 & 0 \\ \mu & 1-3\lambda-\mu & 3\lambda & 0 \\ 0 & \mu & 1-2\lambda-\mu & 2\lambda \\ 0 & \mu & 0 & 1-\mu \end{pmatrix} \tag{4.53}$$

極限確率 P_0、P_1、P_2、P_3 について、次の連立式が成立する。

$$(\ P_0\ \ P_1\ \ P_2\ \ P_3\ )P = (\ P_0\ \ P_1\ \ P_2\ \ P_3\ ) \tag{4.54}$$

$$P_0 + P_1 + P_2 + P_3 = 1 \tag{4.55}$$

式 (4.53)(4.54)(4.55) より、先に式 (4.42) を導いたのと同じ手順をとって次の結果を得る。

$$\text{予防引継ぎシステム(最小構成)}MTBF = \frac{1}{36\lambda^3}(30\lambda^2+8\lambda\mu+\mu^2) \quad (4.56)$$

$$(\text{一般に}\mu \gg \lambda\text{であるから}) \cong \frac{\mu^2}{36\lambda^3} \quad (4.57)$$

続いて、ストラタス社 XA-2000 の場合の定量評価を行なう。この場合のシステム構成は、次の如くなる。

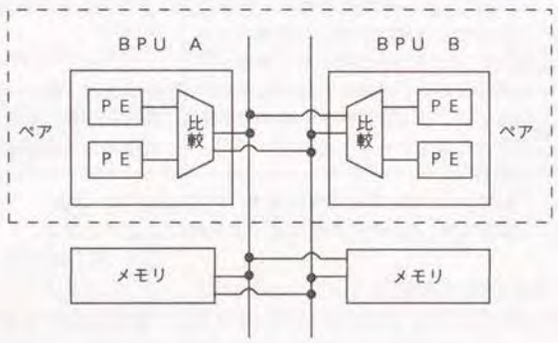


図 4.21: ストラタス社 XA-2000 の例（最小構成）

このとき、システムがとりうる状態は、表 4.4の３種類となる。

表 4.4: ストラタス社 XA-2000 システム状態表

状態名	ＢＰＵ状態	負荷状態（又は作業状態）	ＰＥ状態
E_0	正常　正常	リアルタイム負荷　同一リアルタイム負荷	４台全部正常
E_1	正常　修理中	リアルタイム負荷　修理作業	４台中どれか１台でエラー
E_2	修理待ち　修理中	修理待ち　修理作業	片ＢＰＵ修理中に他ＢＰＵの２台ＰＥ中１台がエラー

マルコフ過程遷移図は図 4.22のようになる。ただし、λ はＰＥ１個の故障率、μ はBPU ボード１枚の修理率である。

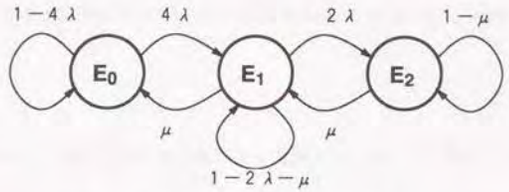


図 4.22: ストラタス XA-2000 状態遷移図

遷移確率行列 P は次のようになる。

$$P = \begin{pmatrix} 1-4\lambda & 4\lambda & 0 \\ \mu & 1-2\lambda-\mu & 2\lambda \\ 0 & \mu & 1-\mu \end{pmatrix} \quad (4.58)$$

極限確率 P_0、P_1、P_2 について、次の連立式が成立する。

$$(P_0 \quad P_1 \quad P_2)P \quad = \quad (P_0 \quad P_1 \quad P_2) \tag{4.59}$$

$$P_0 + P_1 + P_2 \quad = \quad 1 \tag{4.60}$$

式 (4.57)(4.58)(4.59) を解くと次の結果を得る。

$$\text{ストラタス}XA-2000\ (\text{ペア\&スペア})\ \text{最小構成}MTBF \quad = \quad \frac{4\lambda+\mu}{8\lambda^2} \tag{4.61}$$

$$(\text{一般に}\mu \gg \lambda\ \text{であるから}) \quad \cong \quad \frac{\mu}{8\lambda^2} \tag{4.62}$$

ここで、予防引継ぎシステム（最小構成）の MTBF とストラタス XA-2000（最小構成）の MTBF との比を式 (4.57) と式 (4.58) から求める。

$$\frac{\text{予防引継ぎシステム（最小構成）}\ MTBF}{\text{ストラタス}XA-2000\ (\text{最小構成})\ MTBF} \cong \frac{\mu^2}{36\lambda^3} \cdot \frac{8\lambda^2}{\mu} = \frac{2}{9} \cdot \frac{\mu}{\lambda} \tag{4.63}$$

を得る。一般に、$\mu \gg \lambda$ であるから式 (4.63) は大きな値となる。具体数値として、

$$\lambda \ = \ 1 \cdot 10^3 \quad \mu \ = \ 1 \text{ にて試算すると式(4.63)は} \frac{2}{9} \cdot 10^3 \ = \ 222 \tag{4.64}$$

となり、この場合は予防引継ぎシステムの MTBF の方が 200 倍程度長いことが判る。

4.8 まとめ

本章においては、フォールトトレラントシステムを構築するとき、その中核技術となる冗長ユニットによる引継ぎ方式について、そのうち 2 つの方式を主体に論じた。

その第一は、リアルタイム制御システムにおけるフォールトトレラントコンピュータとしては、前章で論じたソフトウェアによるエラー検知方式と併せて、独立型ホットスタンバイ二重系が適合していることを論じ、そのための設計要件を示した。その具体的実施例として国鉄郡山操車場ＹＡＣシステムの構築方式と実績を示し、定量的にも予期した成果を上げ得たことを示した。なお、かかる二重系における信頼性には、システムコンソール誤操作の如きシステム全体共通部にかかわるフォールトが、僅かな発生確率であっても全体の信頼性に大きく影響することをＹＡＣ実績とモデル計算とで示した。

第二は、現在および今後の冗長構成技術として、発達著しい市販マイクロプロセッサを使って、三重多数決方式による冗長構成をとり、予防引継ぎ方式と名付けたホットスタンバイでもコールドスタンバイでもない、新しい引継ぎ方式を提唱し、この方式がＯＳとして既存標準のものが使えるためオープン指向であること、又同様な方向を目指すストラタス社ペア＆スペア方式に比べて経済的であり、更に信頼度の面でも優れていることを定量的に示した。

第5章

高信頼度分散システムへの
アプローチ

第 5 章

高信頼度分散システムへのアプローチ

5.1 はじめに

ＶＬＳＩ技術の発展による計算機のダウンサイジング、そしてネットワークの高信頼化、高速化によって分散処理が計算機システムの主要な形態になってきた。分散処理システムは、計算機の特性に応じた役割の分担や負荷の適正配分によるスループットの向上を可能としている。

更に、現在では分散処理システムを制御系やトランザクション処理など、従来集中型のシステムが主に用いられてきた分野に適用する試みが行われており、そのための技術開発が行われている。分散処理システムをそれらの分野に適用するには、分散環境をいかに高信頼化するかが重要な技術となる。

分散処理システムは、処理要素の物理的な分散によってフォールトの影響が局所化され、同一要素の配置によって冗長性が生じるなど、本質的に高信頼システムに不可欠な性質を持っている。一方、そのような分散した構成要素を一貫性を持つシステムとして動作させるには、適正なシステム管理を行なう必要があり、またそのようなシステムを開発することは容易ではない。

本章では、上記の背景から分散処理システムにおける高信頼化技術の検討を行った結果について論じる。また、本章では、リアルタイムシステムに焦点をあてるよりは、広く分散処理システムを対象としている。リアルタイムシステムもまた分散処理システムの方向へ動いているからである。将来の姿としては、業務の相違によるシステム形態の相違がだんだん無くなる方向へ動いている。本章は、時間軸では数年後に

実用化されると考えられる技術を研究対象としているので、単に分散処理システムという広い対象をとりあげていく。

5.2 分散システム上のフォールトトレラント技術

フォールトトレラントシステムの技術要素は、第2章で論じたように、（1）エラーの検知、（2）フォールトユニットの切り離し、再構成、（3）代替冗長ユニットによる引継ぎ、そして再び冗長系への復旧といえる。また、（4）これらの技術要素を分散環境でいかに実現し管理するかが分散環境におけるフォールトトレラントの実現技術である。

（1）の分散環境におけるエラー検知の問題では、エラーを検知する方法、エラーを検知するまでの時間が重要である。エラーを検知する方法としては、分散環境におけるウォッチドッグタイマによるプロセスの生存の確認、複数プロセスの処理結果を照合することによる多数決論理の採用、テストプログラムによる定期的なプロセスの動作チェックなどが有効である。

（2）のフォールトユニットの切り離し、再構成では、どのような手順で再構成するか、どの冗長系をどう組み合わせるかによって、サービスの再開までの時間、再開されたサービスの縮退の度合などが変わってくる。許容されるサービス停止時間や低下するサービス品質に応じて、柔軟に対応できる必要がある。

（3）の冗長機による引継ぎでは、他の系の要素に大きな影響を与えることなくスムーズに引継ぎ作業を開始できる必要がある。他のサービスを停止して、新しい正常機を投入するのでは、せっかくサービスを停止させずに動作させている意味がなくなる。また、正常機が正しく動作するためには、一貫性がとれたプロセスの状態を他のプロセス、あるいはデータ管理機構から受け継ぐと同時に、自分が正常に動作を開始したことを管理機構に登録することが必要である。このタイミングは、一貫性を保証するために、アトミックに行なわれる必要がある。

（4）の冗長系の構築、管理においては、ユーザ、あるいはプログラマがいかに分散型のシステム上に冗長系を構築し、管理するかという技術である。分散環境における冗長系は、プロセスとデータが一貫性を持った状態で分散されており、常に状態

管理メッセージを交換することで状態の保持に注意する必要がある。この冗長系の構築をユーザが自分で行なうのは多くの労力を必要とするし、専門的な知識も必要となる。従って、冗長系の構築や管理は、多くの場合ユーザには直接見えないように構築することが有効である。

これらの技術を実現するにあたっては、分散環境における冗長性と分散性にもかかわらず、全体として論理的一貫性を持つものでなければならない。これは、集中システムでは容易に管理できることであるが、分散システムでは一貫性を損なう要素が幾つも存在する。

（A）非同期性

複数のプロセスが共有する資源へほぼ同時にアクセスしようとする場合には、相互干渉によって、本来ならあり得ない不合理な状態が出現する可能性がある。このため、ハードウェアレベルからプロセスレベルまでの各レベルで「相互排除」を実現する問題が生じる。また、共有データへのアクセス動作に対して相互排除が実現されたとしても、トランザクションレベルでの相互干渉によって一貫性が失われることもあるため、「並行性制御」が必要になる。

（B）多重化

異るノードに分散された多重化データを更新する場合に、全てのコピーに対して同じ条件でアクセスできるとは限らない。もし、すべてのコピーに対して全く同じ更新処理が行なわれないと、それらの多重化コピーの間のデータとしての論理的な一貫性が失われる。このため、常に論理的一貫性を保つことができる「多重化データ更新」が必要である。

（C）故障

故障あるいは処理の取消が発生した場合、誤り状態から回復するために関連する各プロセスをそれぞれ故障発生以前のある時点へ戻そうとすると、それまでにすでに実行されてしまったプロセス間通信は取り消すことができないので、通信に直接的あるいは間接的に関与したプロセス間で互いに矛盾するシステム

状態が起こり得る。このため、常に論理的一貫性を保つことができる「誤り回復」のメカニズムが必要である。

5.3 従来の取り組み

上記の技術要素に対して、従来幾つかの研究が行なわれてきた。

5.3.1 分散環境におけるエラー検知

エラーが発生した段階で、正しくエラーユニットを検知することがフォールトトレラント構築の第1歩である。

分散環境においては、遠隔地からエラーを検知することも必要であり、そのような場合にはローカルなマシンの中でハードウェア的にエラーを検知するのに比べて通信の遅延が掛かる。従って、ローカルなマシン内でのエラー検知とリカバリ、及び遠隔からのエラー検知と大規模なリカバリとの組み合わせにより、最適な役割の分担をする必要がある。ローカルなマシンのエラーの検知、及びリカバリ技術は本論文では第4章までに論じた。ここでは、分散環境におけるエラー検知の方式とシステム管理の側面について述べる。

分散環境におけるエラー検知の代表的な方法としては、

（1）自己診断プログラムの起動

自己診断プログラムを起動して、エラーを検出した場合は分散環境（あるいは環境の管理系）に対してエラー発生を通知する。環境は、以後そのモジュール（あるいはノード）に対するメッセージの抑制／経路の再設定、システム再構成をする。そして、リカバリルーチンの起動を行なう。

（2）ウォッチドッグタイマの使用

タイマは、周期的に正常に動作するモジュールによってリセットされるものとする。もし、許容時間内にリセットされない場合は、タイムアウトが発生し、モジュールに何らかのフォールトが発生したと判定される。ただし、ウォッチドッグタイマ自体は、モジュールが正常に結果を出したことを保証するものではない。

（3）外部テストプログラム

外部環境から、定期的にテストメッセージを分散環境内の各モジュールに対して送信する。正常な応答が正常な時間内に戻らない場合、そのモジュールに何らかのフォールトがあると判断する。

（4）多数決論理による結果の診断

複数のモジュールに対して、同じ計算を実行させ、その応答メッセージを比較することにより、多数決によるエラー検知を行なう。

エラー検知の方式は、エラーの発生の頻度と、検知に掛かるコスト、検知能力が異り、システムの性質に応じて選択、組み合わせが必要である。

5.3.2 フォールトユニットの切り離しと再構成

エラーが検知された後は、どのようにフォールトユニットを分散システムから切り離し、再構成するかが重要になる。これに対して、文献[44]ではアベイラビリティ管理サービス(AMS)という機構を提案している。AMSは、システムのフォールトやメンテナンス、拡張などによって、ノードの削減や再起動が任意の数で並行に起こる場合でも、ユーザに対しては継続的な可用性を提供する機構である。

この機構は、同期的なアトミックブロードキャストをベースに構築されている。AMSは、各ノードに動作しており、初期状態（アクティブなノード、サービスが稼働可能なノード、どのノードにどのサービスがプライマリ／バックアップとして動いているか）を持っている。そして、AMSは人間や環境、時間などからのトリガーイベントによって反応し、状態遷移を行なっている。この状態変化は、全てのノードで一貫した状態を保つ必要があるため、上記のアトミックブロードキャストが必要となる。

例えば、最も代表的なアベイラビリティ管理のポリシでは、サービスに対して少

なくとも1つのプライマリと一つのバックアップが動作することになる。もし、プライマリが障害を起こした場合、バックアップをプライマリにすると共に、サービスを起動できる他のノードからバックアップノードを選挙して、バックアップを起動する。ここで、注意しなければいけないのは、プライマリが複数ある状態を作ってはいけないことである。

これらの状態遷移とノードの選挙やサービスの起動は、非集中的に行なわれており、どのノードにフォールトが発生した場合で、どのノードが追加された場合でも、整合がとれるように管理される。そのためには、上記のような各ノードにおける状態変数の一貫性のとれた更新メカニズムが重要である。

5.3.3 正常機による引継ぎと同期の問題

フォールトのリカバリにおける最終段階として、正常に動作するモジュール（マシン）の環境への投入とエラーで中断した業務の引継ぎ、およびエラーが発生する以前の状態への復旧がある。この段階で注意する点は、正常に動作するモジュール（マシン）が環境に投入されたという状態変化が、関係するモジュールすべてに対して同時に伝搬される必要があるということである。これは、一部のモジュールが、その状態変化の伝搬に不整合を生じていると、処理要求伝達の不整合やデータ管理の不整合を生じることになる。これは、メンバシップ情報の一貫性の保持という問題である。

一方、正常なモジュールが投入され、他のモジュールと同期して起動するためには、起動したことを他のモジュールに知らせると同時に、他のモジュールと同等なジョブを行なえるようにするために、モジュールの状態をあるチェックポイントの状態に設定することが必要である。あるチェックポイントではなく、他のモジュールの状態をすべてコピーする方式が用いられることもある。それらの状態転送と、モジュールの投入によるメンバシップ情報の変更は、アトミックに行なわれることが必要である。

5.3.4 分散環境における冗長化

分散環境における冗長化系の実現においては、分散環境における幾つかのプロセスをグループ化して、それを対象とする通信機構を実現する方法が検討されている[45][46]。[45]では、クライアントとサーバ間の通信インタフェースのグループ化を目的とした分散環境のモデルを提案している。[46]では、プロセスグループという論理的な通信対象をベースにしたリモートプロシジャコールシステムを構築している。

5.4 分散オブジェクト管理によるフォールトトレランス

5.4.1 分散処理と資源の管理

分散処理の目的の一つは、分散環境に含まれている資源の共有である。資源の共有は、高速プロセッサや特殊グラフィック表示装置、高速プリンタや専用ハードウェアなどへのアクセスを可能とし、更にデータベースなどの情報の共有も可能とする。

分散処理のもう一つの目的は、処理を複数のプロセッサで分担することにより、性能や信頼性を向上させることにある。大きなアプリケーションを、サーバ機能とクライアント機能の二つに分割し、それぞれ別の計算機で処理を行なうことにより、より高速な応答速度を得ることを目的としたクライアント・サーバ型アーキテクチャがその代表的なものである。

また、ネットワーク管理やファイル管理、システム管理、入出力管理などの資源管理アプリケーションは、その情報を集中的に管理し、最適な資源の管理を行なう必要がある。そのため、多くの資源管理アプリケーションは、一つの集中データを持ち、一つのノード上から多くの他のノードを監視する形態で処理を行なうようになる。この場合、その管理アプリケーションに集中することになる負荷は、分散の度合が大きければ大きいほど高くなり、処理効率は逆に悪くなる。

これは、即ち処理の効率を向上させるためには、分散の度合が大きい必要があり、管理の効率を向上させるためには、分散の度合が不利となる。従来、管理アプリケーションの設計では、その処理方式の効率化について考慮されることは少なかった。また、処理の分散において、その資源の管理の側面を考慮している例も少ない。

即ち、管理アプリケーションは、分散されて高速に処理されるべき通常のアプリケーションの一種でもあり、更にそのシステム資源を管理するという側面を持つ。また、通常アプリケーションは、処理が分散されることによって高速に処理を進めていくことが可能だが、その処理の中には、システム資源の状況を把握することによって高速化を行なうなど、管理の機能が含まれてくる。

これは、広くアプリケーション自体がただの計算や入出力処理から資源管理を含んだものへと変化していくことを意味しており、この新しいアプリケーションに対応するためには、従来の処理の分散と管理を分離した分散環境では困難である。

5.4.2 分散オブジェクト管理技術に基づく高信頼化

分散オブジェクト管理に基づいて、システムの高信頼化をすれば、ハードウェアやオペレーティングシステムなどに変更を加えることなく、標準的なシステムを用いて高信頼システムを構成できる。また、オブジェクト指向設計によるシステム開発が行なわれるため、開発の生産性の向上や、ソフトウェアの品質向上も期待できる。

しかし、分散環境におけるフォールトトレラントなシステムを実際に構築するためには、以下のような問題点が存在する。

(1) 対象システムにおける適用性

対象システムでは、どのような多重化、冗長化が必要であり、どのような管理方式が適しているか。どのようなプログラミングインタフェースを提供するのか。どのようなアーキテクチャを構築するのか。例えば、高信頼のネットワークと高信頼のファイルサーバとによって、分散環境の情報がチェックポイントごとに格納されるようなシステムでは、計算ノードの障害はそれほど大きなダメージをシステムに与えることはなく、プロセスの冗長化などは必要なくなる。逆に、二次記憶などの低速な記憶装置による情報の集中管理がなされないような、リアルタイム制御システムにおいては、ノードの障害はノード上で管理されているデータの消失という大きなダメージを伴うため、プロセスの冗長化などが必要となる。

(2) 分散システムの構築のしやすさ

分散環境では、分散型のソフトウェアを開発するのにも大変な労力を必要とする上に、プロセスの冗長化、プロセス間の一貫性のとれた状態変化、冗長化されたプロセスへの同報通信など、多くのプログラミング技法が要求される。そのようなシステム開発をいかに容易に行なうかが問題である。従来、分散プログラミングでは、RPC（リモートプロシジャコール）が使われており、有効性が示されている。しかし、オブジェクト指向システム開発を有効に分散環境に活かすために、オブジェクト間のメッセージパッシングを自然に記述できる開発環境が必要である。

このような、オブジェクト管理機構を提供することにより、分散環境のプログラミングがオブジェクト指向という枠組に基づいてシステマティックに行なえるが、それと高信頼化のための機構（プロセスの冗長化、冗長化されたプロセス間でのメッセージ通信、冗長化されたプロセスの状態管理）をユーザにどのように提示するかが問題である。

ここでは、このような問題を解決するための環境として、リソース指向分散環境ＲＯＤＳ（Resource-Oriented Distributed System）を提案する。ＲＯＤＳの目的は、処理の分散と資源の管理を含む新しいアプリケーションを効率良く開発し動作させることである。

5.4.3 リソース指向分散環境

RODS の基本動作モデル

ＲＯＤＳは、位置透過なオブジェクト管理機構、最適なオブジェクト選定機能、オブジェクト間でのメッセージ配送機構を含むインフラストラクチャと、インフラストラクチャ上で動作するオブジェクト自身の管理および動作機構から構成されている[47]。また、ＲＯＤＳ上で動作するオブジェクトには、他のオブジェクトからの要求に応じてサービスを提供するためのサービスインタフェースと、インフラストラクチャからの要求に応じて管理情報を提供したり、インフラストラクチャへ自律的に管理メッセージを送ったりするための管理インタフェースを持っている（図5.1）。こ

の管理インタフェースは、文献[45]で述べられているような、グループインタフェースのメンバを管理するためだけに用いられているものではない。

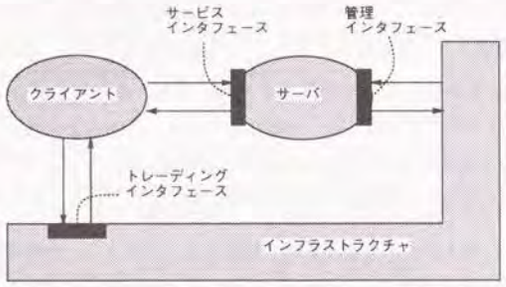


図 5.1: RODS の基本動作モデル

ＲＯＤＳでは、オブジェクトへの要求は通常の計算に関する要求の他に、オブジェクトの管理要求が含まれる。例えば、オブジェクトの生成、オブジェクトの削除、オブジェクトの属性変更などが要求として送られる。これに対して、ＲＯＤＳの環境を利用することにより、オブジェクトの位置を意識することなく、これらの管理業務を可能とすることができる。

ＲＯＤＳが提供する管理機能としては、(1) オブジェクトの生成、(2) オブジェクトの削除、(3) オブジェクトのタイプ設定、(4) オブジェクトのタイプ変更、(5) オブジェクトの複製の数の設定、(6) オブジェクトの生成可能ノードの管理、(7) オブジェクトの動作状況の管理、(8) オブジェクトが動作するノードの負荷状況の管理、があるが、これらを組合わせることでオブジェクトの管理を行ない、管理機能が含まれるアプリケーションの構築を容易にすることができる。

インフラストラクチャの構造

上記の機能を実現するために、インフラストラクチャでは以下のコンポーネントによる機能の切りわけを行なう。

(1) 基本通信機能

(2) トレーディング機能

(3) ネームサービス

(4) ドメイン管理

基本通信機能は、オブジェクトが位置透過にメッセージを送受信し、オブジェクトの生成／消去を行なうための、メッセージ通信インタフェース、およびオブジェクトの物理的作成機能を司る。

トレーディング機能は、オブジェクトがメッセージを送信する際に、複数の送信先候補からある選択基準に基づいて、最適なオブジェクトのアドレスを選択する機能である。例えば、ＲＯＤＳにおける複製を管理する機能は、トレーディング機能を使って実現される。ＲＯＤＳでは、複製が存在することはユーザには隠蔽される (複製透過)。幾つかの複製が存在する場合、そのオブジェクトへのメッセージはトレーディング機能によって最適なオブジェクトが選択されて送信されることになる。また、オブジェクトの存在状況は、ドメイン管理によって監視されており、複製に異常があれば、クライアントからのメッセージは異常なオブジェクトを避けて正常なオブジェクトに送信されることになる。

ネームサービスは、オブジェクトの名前とアドレスの対応を管理する。クライアントオブジェクトは、このネームサービスによって、実際のサーバオブジェクトのアドレスと、クラス名（タイプ名）との対応付けを行なってもらうことが可能であり、具体的なオブジェクトＩＤを指定しなくて良くなる。また、クラスの階層管理や、グループ管理により、あるサービスを提供するオブジェクト群に対応する名前を使った抽象的なサービス管理が可能となる。

ドメイン管理は、オブジェクトが実際にどのノードで動作しており、そのオブジェクトがどのような状態にあるか、そのノードがどのような状況にあるかを管理している。従って、負荷情報やオブジェクトの活性化状態はどのマネージャによって管理する。

RODSにおけるインフラストラクチャの構造を図5.2に示す。

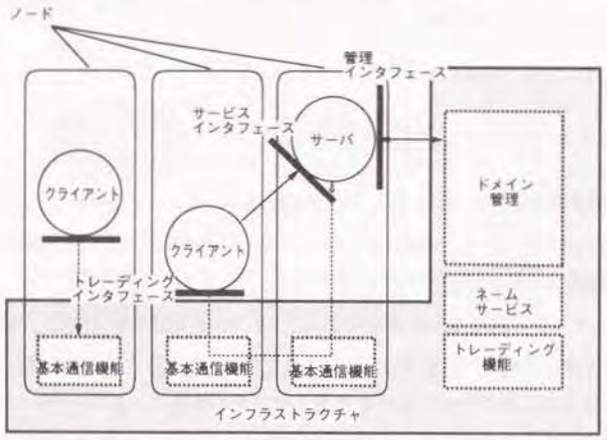


図 5.2: RODSにおけるインフラストラクチャの構造

5.5 RODS の実現方式

RODSの実現は、UNIX上にC言語を使って行なわれている。各マネージャ間は、共通のメッセージフォーマットを使って、メッセージ通信を行なっている。マネージャの複製機能により、フォールトトレラント性について考慮している。

RODSの実現アーキテクチャは、インフラストラクチャの構造から導出された機能要素を分散環境に展開したものとなっている。その実現アーキテクチャの機能要素は以下のようになる。

(A) ニュークリアス

(B) ノードマネージャ

(C) トレーディングファンクション

(D) タイプマネージャ

(E) ドメインマネージャ

インフラストラクチャにおける（1）基本通信機能は、クライアントからのサービス要求に対応する部分と、オブジェクトを物理的に生成／消去する部分に分割し、それぞれ必要な部分が必要なノードに存在できるようにした。前者を「ニュークリアス」、後者を「ノードマネージャ」として実現する。（2）のトレーディング機能は、ユーザの要求に付随したものとなっており、現在は複雑なアルゴリズムは採用していないこともあるため、ニュークリアスの内部に「トレーディングファンクション」として実現した。（3）のネームサービスは、クラスとオブジェクトの名前の管理、クラスのプログラムコードの存在するノードの管理を含めて、「タイプマネージャ」によって実現する。（4）のドメイン管理は、「ドメインマネージャ」によって対応することにした。

RODSのプログラマインタフェースは、オブジェクト指向のインタフェースとなっている。オブジェクトの生成、削除、メッセージ送信などの基本プリミティブがライブラリとして提供されているほか、オブジェクト指向C言語 [48] のインタフェースとしても提供されている。

また、RODSは、オブジェクトの管理に重点を置いており、単純な構造を持つことによって負荷の軽いシステムを目指している。また、クライアントのみのシステムや、サーバのみのシステムなど、自由な構成をとることができる。

RODS の特徴として、冗長性を持たせたオブジェクト間の通信をサポートし、その冗長化されたオブジェクト間の状態の一貫性を保証するメカニズムを内蔵させていることが挙げられる。ユーザは、冗長性を持つオブジェクトの間の複雑な管理や、オブジェクト間の一貫性管理に関する処理を意識する必要がない。ここでのオブジェクト間通信機構は、冗長化されたオブジェクトによるグループ通信のレイヤと、データの一貫性管理のレイヤの2つを持つことになる。

5.5.1 オブジェクトの構造

ＲＯＤＳでのオブジェクトは、利用者（クライアント）には、データ、手続き、手続きを処理する実行者が一つにカプセル化した論理的に一つの存在として見える。しかし、実際のオブジェクトの内部はメソッド部 (Method) とデータ部 (Data) に分離され、物理的に分散配置されている。そして、それぞれが複製を持つことも可能である [49]。このようなオブジェクトとそれを管理する環境を図 5.3に図示する。

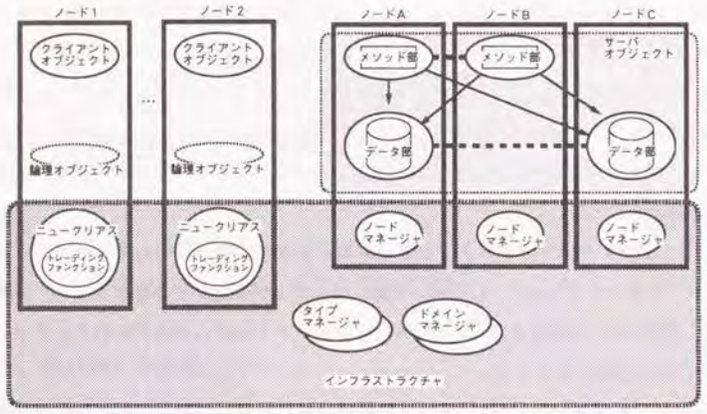


図 5.3: ＲＯＤＳの実現アーキテクチャ（手続きとデータが分散したオブジェクト）

（1）論理オブジェクト

論理オブジェクトは、オブジェクトの実体ではなく、クライアントに対して、物理的な位置や内部が分散されていることを隠し、論理的に１つのオブジェクトとして見せるものである。各論理オブジェクトはオブジェクト ID に対応している。

（2）データ部

オブジェクトのデータ部分が蓄積されている。複製を持つことにより、効率および信頼性を向上させることができる。また、データ全体が膨大な量で一つのノードでは保持しきれない場合には、複数のデータ部に分割することができる。

（3）メソッド部

クライアントがデータを利用する場合は、必ずメソッドを介してアクセスしなければならない。メソッド部もデータ部と同様複数のノードに分散している。このため、ノードの負荷やデータ部との関係などに応じてメソッドを動的に選択することにより、クライアントに最適なサービスを提供することができる。

各オブジェクトのメソッド部およびデータ部は、複製も含め、互いの情報を保持しており、それを管理する機能を持っている。

このようにメソッド部とデータ部に分離することにより、例えば、大容量の記憶領域を持つノードにデータ部を高速なＣＰＵを持つノードにメソッド部を配置したり、頻繁にアクセスを行なう利用者の近くにメソッドとデータの一部を配置するなど、利用形態に応じて最適なオブジェクト配置が可能となる。

5.5.2 オブジェクトの管理

ＲＯＤＳにおけるオブジェクトは、インフラストラクチャにより管理されている。インフラストラクチャは、図5.3に示したように、以下の構成要素からなる。

(1)ニュークリアス

クライアントを動作させる各物理ノードに常駐し、クライアントにオブジェクトの分散透過性を提供する。クライアントからの要求に基づきトレーディングファンクションを用いて最適なサーバオブジェクトを選択し、そのサーバに対してメッセージを転送する。

(2)トレーダ（トレーディングファンクション）

クライアントからの要求とタイプマネージャ、ドメインマネージャから得られるオブジェクトやノードに関する情報をもとに最適なサーバを決定する。負荷分散アルゴリズムの実現も、この構成要素で実現されている [50]。ただし、本実装では、トレーダは独立したプロセスとしてではなく、ニュークリアス内の一つの機能（トレーディングファンクション）として実現されている。

(3) タイプマネージャ

ドメイン内のオブジェクトやクラスに関する名前とオブジェクトとの間の関係、および、クラス名とそのクラス情報を持つノードの関係を管理している。

タイプマネージャが管理している名前には、クラスを一意に識別するクラス名の他に、クラスの性質やサービスによってグループ化したクラス群を表すクラスタイプ名があり、より抽象度を上げたオブジェクトの管理が可能となっている。

図5.4に、ＲＯＤＳにおけるクラスとオブジェクトの管理コンセプトを示す。ＲＯＤＳでは、オブジェクトとクラスを２つの階層で管理する。一つは、クラスのグループ化の概念であり、「クラス→クラスタイプ」、「オブジェクト→オブジェクトグループ」の対応付けがなされている。もう一つは、「オブジェクト→複製されたオブジェクト群」の対応付けであり、これによって複製がユーザに隠蔽された形で管理されている。

タイプマネージャは、概念的には、ドメイン内に一つの存在であるが、物理的には分散していてもよい。

(4) ドメインマネージャ

ドメイン内の各オブジェクトに関する管理情報を保持している。各ノードマネージャからの情報に基づいて、オブジェクトが動作しているノード、各ノードの負荷情報などを一元管理している。

ただし、タイプマネージャと同様に、物理的には分散していてもよい。

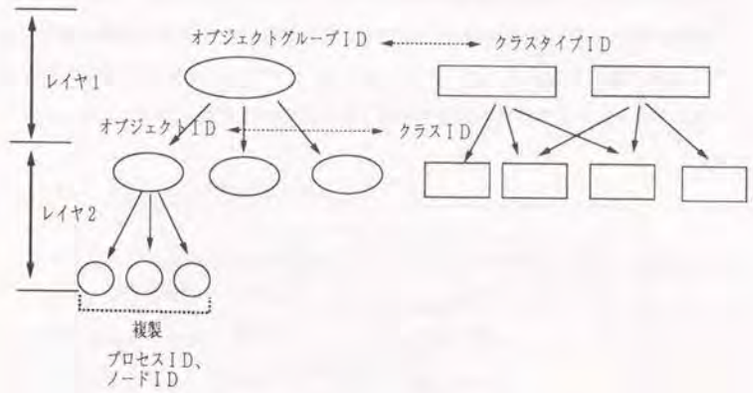


図 5.4: RODS のクラス、オブジェクト管理

(5) ノードマネージャ

サーバが動作する各物理ノードに常駐し、オブジェクトの動作状況や負荷情報をドメインマネージャに報告する。また、ノード毎のクラス情報も管理しており、オブジェクトの生成削除も行なう。

以上の構成要素が協調動作することにより、複数ノードに分散しているオブジェクトを利用者には、一つの論理的なオブジェクトとして透過的に見せている。

5.5.3 オブジェクトの状態管理

複製されたオブジェクト間の一貫性を管理するために、RODS では楽観的手法に基づく一貫性制御を行なっている。RODS は生産管理制御システムを主対象としている。その場合、オブジェクトが持つデータの更新はそれほど多くなくて、生産予定の個数や工作機械にダウンロードするためのプログラムなど、情報の読み込みが主な処理になると考えられる。そこで、高度なロック管理による並行制御を行なっても、それに掛かるコストが高すぎることが予想される。従って、楽観的手法による並行制御

が最もパフォーマンスが高いと考えられる。

RODS が用いている並行制御の方式では、データオブジェクトに管理されているオブジェクトのインスタンスデータと、メソッドオブジェクトが持つインスタンスデータのキャッシュとの一貫性制御が必要となる。その方式は、以下のような手順となる。

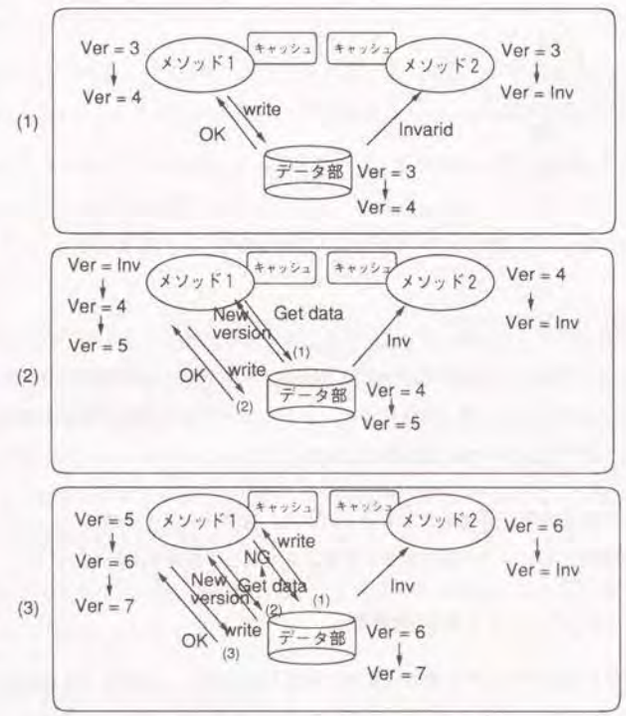


図 5.5: RODS のデータの一貫性管理

（1）キャッシュが無効化されていない

この場合、読み込みではそのままキャッシュ情報を読み込んで処理する。書き込みでは、データオブジェクトのバージョンと自分のバージョンを比較し、同じならデータオブジェクトとキャッシュのバージョンを上げて書き込む。そして、他のメソッドオブジェクトには、キャッシュの無効化を通知する。

（2）キャッシュが無効化されている

キャッシュが無効化されている場合、メソッドオブジェクトは処理を開始する前に、データオブジェクトから最新のデータを読み込んでくる。メソッドの処理が読み込み処理であれば、それを行なって終了する。もし、書き込み処理であれば、読み込んだ最新のデータに対して計算を行なって、データオブジェクトに書き込みを行なう。そのとき、データオブジェクトのバージョンと自分のバージョンが一致していることを再度確認する。そして、その時点で更にバージョンが異っていれば、再度最新のデータを読み込み、処理を繰り返す。

（3）キャッシュが無効化されていないが、書き込み時点で無効であることが判明

この場合、無効であることが判明した時点で、再度データオブジェクトから最新のバージョンをメソッドに読み込んでくる。そして、計算を読み込んだ最新のデータに対して再度行なって、改めてデータオブジェクトに書き込みを行なう。そのとき、データオブジェクトのバージョンと自分のバージョンが一致していることを再度確認する。そして、その時点で更にバージョンが異っていれば、再度最新のデータを読み込み、処理を繰り返す。

これは、データオブジェクトの更新とキャッシュのバージョンアップの関係で、メソッドオブジェクトが無限にキャッシュのバージョンアップを行なう可能性が残るが、はじめの条件で述べたように、対象となる制御システムのデータ更新の頻度が非常に低いことから、今回の設計では性能を重視するためこの方式をとる。

5.5.4 動作例

ここでは、今まで述べてきたインフラストラクチャ内の各マネージャ群とオブジェクトが実際にどのように協調動作しているかを、以下、オブジェクトの生成、削除、メッソッド呼び出しを例として示す。

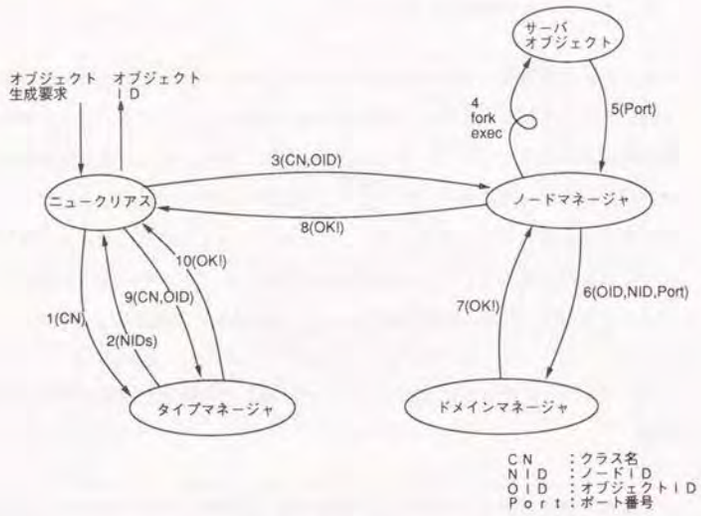


図 5.6: オブジェクトの生成

（1）オブジェクトの生成（図 5.6）

ニュークリアスは、タイプマネージャにクラスが存在するノードを問い合わせる。これに対してタイプマネージャは、クラスが存在する全てのノードとそのクラスに設定されている冗長度（複製の数）を返す。

ニュークリアスは、生成するノードのノードマネージャに生成要求を送る。

複製を生成する場合は、生成するすべてのノードに対して生成要求を送る。

ノードマネージャは、指定されたクラスのオブジェクトを起動し、プロセス（又はタスク）として生成する。

このときオブジェクトのメソッドおよびデータは、それぞれプロセスとして起動される。

メソッドは、Unix におけるトランスポート端点であるポートを確保し、このポート番号をＮＭに伝える。

ノードマネージャは、ポート番号を登録し、ドメインマネージャに生成を報告する。

ニュークリアスは、タイプマネージャにオブジェクトの生成を報告する。

（2）オブジェクトの削除

ニュークリアスは、ドメインマネージャに削除したいオブジェクトの動作ノードを問い合わせる。

ニュークリアスは、ノードマネージャに対して、オブジェクト (メソッド) の削除要求を送る。

ニュークリアスは、ドメインマネージャにメソッドを終了させたことを報告する。

ドメインマネージャは、オブジェクトが動作していた全てのノードからメソッドの終了報告が来たら (そのオブジェクトが動作しているノードがなくなったら)、タイプマネージャにオブジェクトの削除を要求する。

（3）メソッド呼出し

ニュークリアスは、クラス名、クラスタイプ名、オブジェクトタイプ名などを指定して、タイプマネージャに対してオブジェクトＩＤを問い合わせる。

ニュークリアスは、ドメインマネージャにオブジェクトの負荷情報を問い合わせる。

ニュークリアスは、最適なサーバオブジェクトを選択し、そこに要求を送信する。

メソッドは、要求を受けとり処理を開始するときに、自分がアクティブになったことをノードマネージャに伝える。

メソッドは、処理を終了してアイドルになったときに、自分がアイドルになったことをノードマネージャに伝える。

ノードマネージャは、ノード内でのアクティブなメソッド数をインクリメントし、その値が閾値を越えた場合は、負荷が「High」になったことをドメインマネージャに報告する。その値が閾値を下回った場合は、負荷が「Low」になったことをドメインマネージャに報告する。ここで、閾値は各ノードの管理者がそのノードの性能などを考慮して決定した値で、初期設定ファイルに指定されている。この他に、クラスの登録、サーバオブジェクトの登録、グループ化、名前付け、オブジェクトの検索などの手続きが提供される。

5.6 評価

ＲＯＤＳの機能面での評価と、性能面での評価を行なうために、簡単な分散アプリケーションの記述を行なった。この分散アプリケーションは、データベースという共有資源を階層的に管理するアプリケーションであり、信頼性と速度が要求されている。

これに対して、ＲＯＤＳではオブジェクトの階層的な管理機構を用いて、このシステムを自然に設計して実現できる。また、複製を用いることにより、このシステムの信頼性と性能の向上を期待することができる。

5.6.1 分散アプリケーションの概要

RODS が対象としている、生産システムの位置付けを図 5.7に示す。図 5.7は、製造業においてファクトリーオートメーション（FA）化を行なったときの、全体システム構成を表している。製造管理においては、その業務は大きく設計部門、生産管理部

門、生産システム部門に分けられる。生産システム部門では、各種のコントローラがネットワークで接続され、加工、組み立てなどの生産ラインを構成している。生産システムでは、生産管理部門からの生産計画に基づいて、設計部門からの仕様、あるいは加工プログラムによって生産を行なう。そして、定期的に生産状況のモニタリング、装置の動作状況などの通知によって、適切な生産管理を行なう。

コントローラは、制御対象によりオペレーティングシステムを持たないものなど様々であるが、 RODS の適用対象としては比較的 CPU の性能が高い、オペレーティングシステムを搭載したコントローラを考えている。

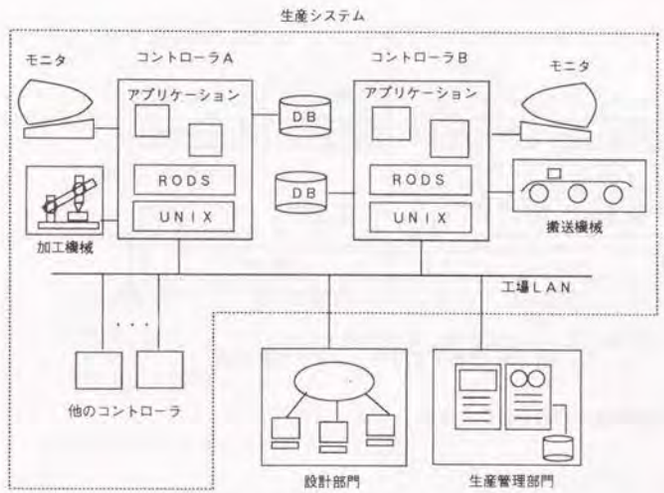


図 5.7: 生産システムの位置付け

なお、本項では工場での生産システムを分散制御するアプリケーションについて述べる。本アプリケーションは、ＲＯＤＳが持つ、1) 拡張性、2) 位置透過性、3) 負荷分散、4) フォールトトレラントといった特徴を検証するものである。以下に、その概要を述べる。

本アプリケーションは、データ管理部、機械管理部、ユーザ管理部から構成され

る。

本アプリケーションは、複数のノード上に存在する加工機械オブジェクトと搬送オブジェクトを、1つのノードから制御するためのアプリケーションである。制御オブジェクトの他に、DB（データベース）オブジェクト、ディスプレイオブジェクト、コマンドインタプリタオブジェクトが存在する構成とする。制御アプリケーションの動作は、プログラムのダウンロード、スケジュールのダウンロード、加工結果のアップロード、機械の起動、停止、モニタリング等である。

5.6.2 分散アプリケーションのオブジェクト構成

分散アプリケーションのオブジェクト構成は、図5.8のようになる。

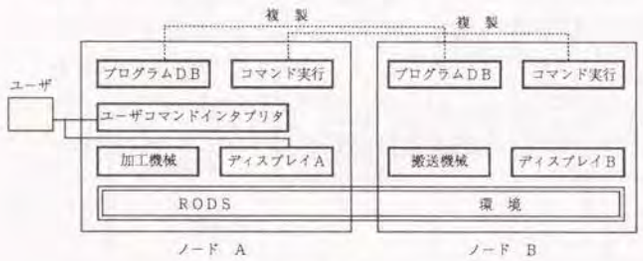


図 5.8: 分散アプリケーションの構成図

各構成要素の動作内容を以下に示す。

- メイン・オブジェクト

  コマンドインタプリタ、加工機械、搬送機械、ディスプレイ、プログラムDB、コマンド実行の各オブジェクトの起動および初期化を行なう。

- コマンドインタプリタ・オブジェクト

  ユーザから、プログラムのダウンロード、起動、停止のコマンドを受け付け、コマンド実行オブジェクトへの依頼を行なう。

- コマンド実行オブジェクト

  コマンドインタプリタからの要求に応じて、プログラムDB、加工機械、ディスプレイとメッセージをやり取りして加工作業を実施する。

- プログラムDBオブジェクト

  コマンド実行オブジェクトからの、プログラム検索メッセージにより動作して、指定されたプログラムデータを返す。また、このDBはオブジェクト指向に基づいて実現されており、プログラムだけでなく部品情報、治工具、CAD情報などの複雑な構成を持つ情報も管理できる。

- 加工機械オブジェクト

  コマンド実行オブジェクトからの、プログラムロードメッセージ、起動メッセージ、停止メッセージにより動作して、その応答として、現在の状態をモニタ情報として返す。

- 搬送機械オブジェクト

  加工機械オブジェクトと同様にコマンド実行オブジェクトからの、起動メッセージ、停止メッセージにより動作して、その応答として、現在の状態をモニタ情報として返す。

- ディスプレイオブジェクト

  コマンド実行オブジェクトからの、モニタ結果表示メッセージにより起動して、モニタ結果を表示する。

ここで、本分散アプリケーションの全体的な動作概要を説明すると、まずコマンドインタプリタ・オブジェクトが、ユーザ・コマンドを受けつける。そしてユーザコマンドは、コマンド実行オブジェクトへ送られる。コマンド実行オブジェクトは、プログラムDB、加工機械、搬送機械オブジェクトとメッセージ交信を行なって、機械加工を実行すると同時に、ディスプレイ・オブジェクトにその過程を表示する。

5.6.3 分散アプリケーションによる本環境の検証

前項で述べたアプリケーションによって、以下のことを検証することができる。

コマンドインタプリタ・オブジェクトからコマンド実行オブジェクトへのメッセージ、あるいはコマンド実行オブジェクトから、各加工機械へのメッセージを位置透過に送ることができる。

ノードAの負荷が高くなってきた場合に、コマンドインタプリタからのメッセージが自動的にノードBのコマンド実行オブジェクトへ送られるようになり、負荷を分散できる。また、ノードAのオブジェクトに障害が起こった場合、ノードBのオブジェクトにメッセージを送り直すことによって、ノードBで処理を代替し、フォールトトレラント性を高めることができる。

搬送機械も含めて、すべてノードAで動作する環境から、ソースコードを変更することなく、搬送機械をノードBで動作させたり、コマンド実行やプログラムDBの複製を動作させたりする環境へ容易に拡張できる。

5.6.4 性能評価

（1）測定項目

これに対して、RODSの性能を評価するために、次のような測定を行なった。

(A) オブジェクトの生成及び消滅に要する時間

(B) オブジェクトを呼び出すために要する時間

(C) 複製の並行処理による応答速度

（2）測定環境と構成

- CPU

  モトローラ68030

- オペレーティングシステム

  UNIX4．3bsd

- 測定用アプリケーション開発言語

  superC [48]

- 通信インタフェース

  socket インタフェース

- 通信プロトコル

  UDP/IP(データグラム) プロトコル

- ネットワーク

  イーサネット（10Mbps）

- ノード数

  2個のみ。ネットワークの負荷は非常に軽い状態である。

測定項目 (A) と (B) の測定においては、各コンポーネント (クライアント、サーバ、及び各マネージャ) の配置方法が性能に与える影響を調べるために、2つのノードで考えられる全ての配置方法に関して測定を行った。表5.1で、○がそのノードで動作することを表している。ノード1とノード2において、それぞれノードマネージャが動作していない配置があるのは、サーバを動作させないノード（ドメインマネージャとタイプマネージャのみを動作させる、あるいはクライアントのみを動作させるノード）にはノードマネージャを必要としないためである。表5.1で a,...,h は、配置方法を識別する記号である。また、測定項目 (C) において、測定に用いたクライアントとサーバの冗長度の組合せを表5.2に示す。本測定では、クライアントおよびサーバ共に、その最大値を2とした。

（3）測定結果

表5.3に、測定項目 (A) と (B) の結果を示す。この表において、a,...,h は、表5.1で示した各配置方法に対応している。また図5.9に、測定項目 c の結果 (単位：ミリ秒) を示す。横軸はサーバ内での処理量（単位：整数加算回数×

$1 \cdot 0^3$) を、縦軸はクライアントから呼び出した全体の応答時間 (単位：ミリ秒) を各々表している。

表 5.1: 構成要素の配置

	ノード1						ノード2			
	Clt	Nu	Svr	NM	TM	DM	Svr	NM	TM	DM
a	○	○	○	○	○	○				
b	○	○	○	○					○	○
c	○	○	○	○	○					○
d	○	○	○	○		○			○	
e	○	○			○	○	○	○		
f	○	○					○	○	○	○
g	○	○			○		○	○		○
h	○	○				○	○	○	○	

$$\left(\begin{array}{l} Clt: \text{クライアント} \\ Nu: \text{ニュークリアス} \\ Svr: \text{サーバ} \\ NM: \text{ノードマネージャ} \\ TM: \text{タイプマネージャ} \\ DM: \text{ドメインマネージャ} \end{array}\right)$$

表 5.2: クライアント／サーバの個数

番号	クライアント	サーバ
X	1	1
Y	2	1
Z	2	2

表 5.3において、オブジェクトの生成・消滅時間をみると、a－dとe－hはそれぞれ近い値となっている。すなわち、クライアントとサーバの位置を固定、

表 5.3: 各コンフィグレーション毎の測定結果

	測定項目	a	b	c	d	e	f	g	h
(A)	生成・消滅	524	507	506	508	498	487	490	488
(B)	呼び出し	87	87	86	87	71	69	70	70

(単位：ミリ秒)

TMとDMの配置位置を変えて、生成／消滅の処理時間にほとんど影響を与えないことを意味している（a）。

クライアントからのサーバ呼び出し時間では、クライアントとサーバが別ノードに存在する方が同一ノードに存在する場合より（例えば、fの方がbより）応答時間が短い（b）。

図5.9から分かるように、ある程度の通信のオーバーヘッドが伴うが、サーバでの処理時間の増加に応じて、クライアントでの応答時間が、直線的に増加している。クライアントが2つになると、その応答時間はほぼ2倍になり、サーバも2つにすると、応答速度は1：1の場合の結果に近づいている（c）。

5.6.5 考察

測定結果（a）の原因は、データを伴わない測定のため、通信機構自体が、それほどローカルとリモートの通信に差がなかったためと考えられる。また、プロセスの生成時間が大きいため、通信時間の影響が少なかったと考えられる。

測定結果（b）の原因としては、クライアントとサーバの位置関係の違いにより、2つの最もアクティブなマネージャであるNUとNMの位置が分散され、間接的に速度が向上したものと考えられる。

測定結果（c）のクライアントとサーバの数が、2：1から2：2になった際に、応答時間が約半分になったのは、RODSの動的な負荷分散機構が有効に働いたためと考えられる。但し、クライアントとサーバが2：2の場合、1：1の場合より

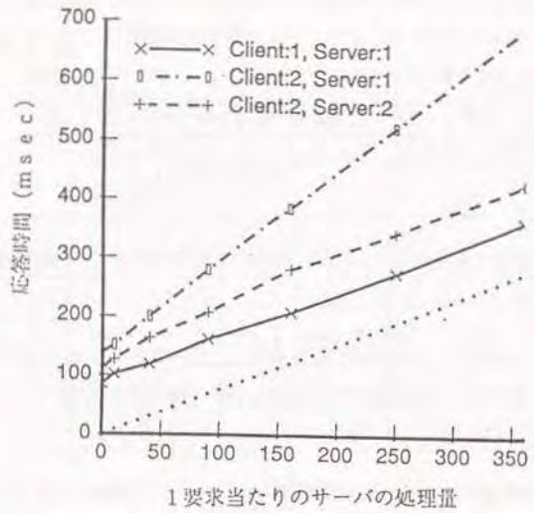


図 5.9: クライアントとサーバの処理時間

も多少の応答時間の増加が見られる。これは、ＲＯＤＳの持つ動的な負荷分散機構において、負荷情報通知の遅れによって、負荷の分散が適切に行なわれない場合があり、平均応答時間の増加に継っていると考えられる。なお、ここでの評価は計算サービスのみであり、一貫性管理のオーバヘッドの評価が含まれていない。データ更新を含む複製サーバでは、更にオーバヘッドが生じることが予想されるが、読み込み処理が主であるサーバでは、本評価と同等の結果と考えられる。

以上の評価結果から、資源の管理を一部に含むような新しい形態のアプリケーションの効率的な開発に十分有効な環境である。性能的には通常ＲＰＣの３倍程度の速度でオブジェクト間のメッセージ通信が実現され、オブジェクトの管理機構や負荷分散機能が重要なアプリケーションにおいては十分評価できる性能であると考えられる。

5.7 まとめ

本章では、分散環境上の新しいアプリケーションの形態（処理の分散と資源管理）に対応する新しい分散環境として、リソース指向分散環境（ＲＯＤＳ）を提案した。ＲＯＤＳは、処理の分散と資源の管理という異なる特徴を含む新しいアプリケーションに対応するため、サービスインタフェースと、管理インタフェースを持たせることにより、容易に実現できるようにした。

ＲＯＤＳを実現し、その有効性と性能を評価するために、分散したデータベースを共有するアプリケーションを記述した。その結果、階層化されたＲＯＤＳのオブジェクト管理機構を使って自然に設計することが可能であったことと、性能面においても、十分実用的であることが明らかとなった。

今後の課題としては、標準的なディレクトリシステムとの連携による広域分散環境への対応、並行性制御、複製管理の実現、負荷分散機構の改良がある。

第6章

結　論

第 6 章

結論

本論文は、リアルタイムコンピュータシステムにおける、フォールトトレラント構築技術に関して、筆者が行なった研究の成果を詳論したものである。現在、コンピュータは色々な形で個人の生活、企業活動、社会基盤に深くかかわっている。その中で、リアルタイムコンピュータシステムは、主として生産活動の現場や交通、水処理、電力、等の社会基盤の中枢部において、管理・制御を司っている。かかる重要な使命を担うリアルタイムコンピュータシステムには、当然ながら極めて高い信頼性が要求される。

今後、先進国における社会環境は、技術の進歩、労働人口の減少、および自由経済メカニズムにより、あらゆる場で省力化／自動化／無人化が進展するといっても過言でない。この場合、危険な陥穽の一つが、システムの不測の障害である。先般も大阪で、ハイテク新交通システム ニュートラム の暴走事故があったことは記憶に新しい。（平成５年１０月５日）

上記のトレンドと事故事例をみるとき、今後ともフォールトトレラントコンピュータあるいはフォールトトレラントシステムに対するニーズは更に増大してくるものと考えられる。

筆者も所属する日本の１コンピュータベンダの立場からすると、フォールトトレラントコンピュータは単に将来需要があるというだけでなく、別の意味をもっている。すなわち、現実のコンピュータオープン化のトレンドを見ると、そのベースとなる市販マイクロプロセッサ、市場標準オペレーティングシステム、市場標準データベース等の基本部分がほぼ完全に米国の技術に押えられている。当面は、日本のベン

ダとしては、そのベースの上に付加価値を探さざるを得ない。この観点から、フォールトトレラントコンピュータは、大きな一つの付加価値対象である。

筆者が指揮をとっている三菱電機情報システム研究所は、かかる観点、すなわちオープンシステムの上に乗せる付加価値という側面から、種々の研究を行なっており、フォールトトレラントコンピュータの他に、分散並列処理、分散自律システム、セキュリティ技術等がその代表例である。なお、将来のフォールトトレラントシステムは、フォールトの一種として人為的な妨害行為も対象にせざるを得ず、そのときはセキュリティ技術がフォールトトレラントシステムの新たなる要素技術となる。

以上、フォールトトレラントコンピュータが持つ別の意味に触れたが、本論文はフォールトトレラントコンピュータシステムを現実に構築するための技術について論じている。

第1章では、フォールトトレラントシステムのニーズを示したあと、その歴史的発展の経緯を述べた。続いて本研究では、フォールトトレラントシステム構築技術を如何なる基本スタンスでとらえ、どういう枠組の中で考えているのかを示した。すなわち、良好な固有信頼度のユニットを使うことを前提とし、システムとして更に高度な信頼性を追求する技術である、とのスタンスであり、又他面では、確率論と各種の技術的選択の妥協を土台にした極めて現実的な技術である、とのスタンスである。

又、本研究ではリアルタイム制御システムを主対象としており、その場合の一般共通的な環境、例えば現場機器を含めたシステム安全性を第一にすること、業務の種類が固定しており、操作員も固定されること、等々の環境下でのシステム構築技術を追求することを述べた。

第2章では、フォールトトレラントシステムを構築するにあたって、一般的には何を考えねばならないか、どういう技術が必要になるのかを論じた。

まず、システムにはどういう種類のフォールトがあり得るのか、それらに対しシステムはどういうフェイリャとなるのか、これらの明確化が必要であることを論じた。続いて、高信頼度運転を実現するためのシステム構築技術として、エラーの検

知、エラーの記録、リトライによるエラーのマスク、フォールトユニットの切離し、および冗長ユニットによる業務の引継ぎを概観した。これら直接的なシステム構築技術の他に、修理を助けるための技術、信頼度を事前に推定する評価技術が間接的ではあるが重要な技術であることも示した。

第3章では、エラーの検知、記録、リトライによるマスクについて、具体的な技術を論じた。とりわけ、本研究で主対象としているリアルタイム制御システムに焦点を絞った場合には、オンライントランザクション処理の分野で常識的に用いられている二重照合チェック方式は不適切であることを示した。すなわち、エラー検知後、二つの系のいづれが正常かの特定に時間を要すること、残った一系のみではエラー検知能力がなくなるので運転できぬこと、ソフトウェアバグに対しては検知能力が低いことがその理由である。これらを解決するためには、何らかの手段で単独系内で自己エラーを検知する必要があり、又ソフトウェアがシステムの制御のために第一次的に使っているアルゴリズムとは別のアルゴリズムでアウトプットを検証する必要があることを示した。この解決策として、ソフトウェアによるエラー検知が有効であることを論じた。具体的な実施例としてＹＡＣシステムにおけるループチェック、バリディティチェックを示した。

又、エラーの記録については、MELCOM 350-30 に適用した記録方式を示し、これが瞬時エラーの原因推定に有力な手掛かりを与えることを論じた。実績として、MELCOM 350-30F におけるボード交換実績を示し、これが瞬時エラー対策としては有効に働いていることを示した。

第4章では、フォールトトレラントシステムを構築するとき、その中核技術となる冗長ユニットによる引継ぎ方式について、そのうち2つの方式を主体に論じた。その第一は独立型ホットスタンバイ二重系である。この具体的実施例として国鉄郡山操車場ＹＡＣシステムの構築方式と実績を示し、定量的にも予期した成果を上げ得たことを示した。なお、かかる二重系における信頼性には、システムコンソール誤操作の如きシステム全体共通部にかかわるフォールトが僅かな発生確率であっても全体の信頼性に大きく影響することをＹＡＣ実績とモデル計算とで示した。

第二は、現在および今後の冗長構成技術として、発達著しい市販マイクロプロ

セッサを使って、三重多数決方式による冗長構成をとり、予防引継ぎ方式と名付けたホットスタンバイでもコールドスタンバイでもない、新しい引継ぎ方式を提唱し、この方式がＯＳとして既存標準のものが使えるためオープン指向であること、又同様な方向を目指すストラタス社ペア＆スペア方式に比べて経済的であり、更に信頼度の面でも優れていることを定量的に示した。

第５章では、５年以上先の未来のコンピュータシステムの一般的な姿を予測し、それを高度な分散処理システムと見きわめ、その環境下でのフォールトトレラントシステム構築技術について論じた。第５章は、特にリアルタイム制御システムに対象を絞ることはせず、むしろ将来のリアルタイム制御システムも分散処理に近づいてゆくとの立場から、一般的分散処理システムを対象として考えた。そこで、分散処理システムにおけるフォールトトレラント化へ近付く第１歩として、分散環境上の新しいアプリケーションの形態（処理の分散と資源管理）に対応する新しい分散環境として、リソース指向分散環境（ＲＯＤＳ）を提案した。ＲＯＤＳは、処理の分散と資源の管理という異なる特徴を含む新しいアプリケーションに対応するため、サービスインタフェースと、管理インタフェースを持たせることにより、容易に実現できるようにした。

ＲＯＤＳを実現し、その有効性と性能を評価するために、分散したデータベースを共有するアプリケーションを記述した。その結果、階層化されたＲＯＤＳのオブジェクト管理機構を使って自然に設計することが可能であることと、性能面においても、十分実用的であることを示した。

以上述べたごとく、本論文ではフォールトトレラント構築技術をその要素技術を軸として分析したが、研究内容にはもう一つの軸がある。それは、具体研究事例として掲げたものが２５年前のものから現在及び未来のものにまたがっているという、時間の軸である。

エラー検出や冗長ユニットの準備／投入といった基本的な考え方は、時間の軸を貫いて共通であるが、その具体的実現方式になるとコンピュータ技術、とりわけ半導

体技術の急速な進歩に伴う素子、実装技術の進歩変遷の影響を受け、かなり変化してきている。変化を押し進める要因は、素子、実装技術の進歩のみではなく、オペレーティングシステム技術やユーザニーズにもある。最近のユーザニーズであるオープン指向はその好例である。とはいうものの、過去の研究事例で述べた具体的実現技術はそれぞれ実績に裏打ちされ実証された価値があり、今後全く同じ技術を使うことはなかろうとも、その考え方には踏襲できるもの、参考になるものが多々ある。

将来フォールトトレラント構築技術は、特殊な重要使命を担う特殊なシステムとして存在するとともに、もっと一般化するものと考えられる。その根拠の一つは、半導体技術の進歩を主たる要因とするハードウェアのより一層のコンパクト化と低価格化である。これに伴って内部的にリダンダントな構成をもつ標準品が出現してくるであろうし、そうでないものでも、その信頼性は十分高いレベルに向上する。他方で、ネットワークとソフトウェアの技術進歩によりネットワーク分散システム上で有機的にあるいは自律的に、相互に協力したり補間しあったりする方向へ発展してゆく。こういう環境になってくると、フォールトトレラントコンピュータという特別なコンピュータではなく、標準のコンピュータ製品が標準のネットワーク上に多数接続している状況で、その上のソフトウェア技術により、標準的な機能の他に相互にバックアップする機能を持たせることが十分可能になってくるものと考えられる。

従って、今後のフォールトトレラント技術の向かう方向は、エラー検知とか、冗長ユニットの引継ぎとかを標準品、標準オペレーティングシステムの上のソフトウェア技術で実現する方向であろう。そのためには、標準オペレーティングシステムそのものが現在のものよりも高機能化したり、柔軟な構造になったりする必要もあろう。

又、今後のフォールトとしては、ハードウェア素子の高信頼化に伴って相対的にソフトウェアバグの割合が高まってくるものと予想される。ソフトウェアバグに対しては、事前のデバッグを極力完全化することと、オンライン中の検知の両面から対策を講ずる必要があるが、筆者は前者にまだまだ研究の余地があると考える。とりわけ、機能（使命）と走行条件との組合わせを視点とするテストカバレッジを高めていく方法について研究課題が多いと考える。後者については、本論文中でも示したパリディティチェックが実用的と思われる。これは、知識処理技術の一つの応用と見るこ

ともできる。今後、適用範囲が広がる可能性があり、更なる具体事例の積み重ねを期待したい。

謝辞

本論文をまとめるにあたって、懇切なる御指導並びに御助言を賜わった東京大学工学部 田中英彦教授に深謝いたします。また、関連する文献を御紹介頂き御指導頂いた東京大学生産技術研究所 坂内正夫教授に厚く感謝いたします。また、本論文に関して貴重な御助言と暖かい励ましを頂いた東京大学工学部 斉藤忠夫教授、正田英介教授、東京大学生産技術研究所 高木幹雄教授、文部省学術情報センター兼東京大学工学部 浜田喬教授に心より感謝いたします。

また、本研究を進めるにあたり、投稿論文に関する広範な御助言を頂いた静岡大学 水野忠則教授に深く感謝いたします。

本研究は、筆者が三菱電機に入社して数年後に開発主任技術者となり、国鉄当局の御指導の下に数十名のメンバと共に文字通り盆も正月もなく寝食を忘れて仕事に没頭した国鉄郡山ＹＡＣプロジェクトの考え方と成果を一つの軸にしています。その関連では、大同通信（株）遊佐社長（当時国鉄本社信号課総括補佐）、大同通信（株）本村専務（同信号課）、鉄道技研（財）八賀部長、三菱プレシジョン（株）太田相談役（当時三菱電機営業部長）、大井電機（株）鈴木副社長（当時ＹＡＣプロジェクトサブリーダ）はじめ、多くの関係者に感謝すると共に、故松元雄蔵 計算機製作所長（当時鎌倉製作所技術部長）、故島村和也 菱電電子機工社長（当時 ＹＡＣプロジェクトマネージャ）の御霊前に本研究を捧げます。

また、本研究の別の軸になっている制御用計算機 MELCOM350-30, 30F の関連では、関連データおよび文献の収集に助力頂いたメルコムサービス（株）池田部長、エムテック（株）沓沢参事に感謝いたします。また、本研究にて新提案している予防引継ぎシステムの関連では、種々の関連特許調査をはじめ、有益な討論をして頂いた三菱電機情報システム研究所 渡辺部長、伊藤グループリーダ、岡本主事に感謝いたしま

す。また、高信頼度分散システムについては、多くの助言を頂いた三菱電機情報システム研究所 佐藤文明氏に感謝いたします。

参考文献

[1] 井原:“フォールトトレラントコンピュータのシステム構成技術の展望”, 電子情報通信学会誌, vol.73, No.11, p.1136 (1990).

[2] Hamming, R.W.:“Single Error Correcting and Double Error Detecting,” (1950).

[3] Soga, M., Tanaka,C., and Ban, K.:“The MELCOM-COSMO computer series,” Mitsubishi Electric Engineer, vol.45 (1975).

[4] Brown, W.G., Tiemey, J., and Wassemn, R.:“ Improvement of Electronic Computer Reliability through the Use of Redundancy,” IRE Trans. Electron. Comput. (1961).

[5] 亀山, 樋口:“ＴＭＲによるフォールトトレラントマイクロコンピュータシステムの一構成法”, 信学技報 ＥＭＣＪ７８－５７ (1979).

[6] 米田, 河村, 古屋, 当麻:“多数決に基づく耐故障システムの故障診断とシステム再構成法”, 電子通信学会論文誌, vol.J67-D, No.7 (1984).

[7] IBM:“IBM System/360 Principle of Operations,” (1970).

[8] 遊佐, 佐々木, 東, 稲田, 岩村, 曽我, 遠藤, 坂：“国鉄郡山操車場自動化システム”, 三菱電機技報, vol.141, no.9 (1967).

[9] Clement, G.F. and Giloth, G.K.:“Evolution of Fault-Tolerant Switching Systems in AT&T,” Springer-Verlag (1987).

[10] Jelinski,Z. and Moranda, P.B.:“Software Reliability Research, Statistical Com-

puter Performance Evaluation," Academic Press (1972).

[11] Littlewood, B. and Verrall, J.L.:" A Bayesian Reliability Growth Model for Computer Software," Royal Statistics Society (1973).

[12] 渡辺:"疎結合マルチプロセッサ," 電子情報通信学会誌, Vol.73, No.11 (1990).

[13] Downing, R.W., Nowak, J.S. and Tuomenoksa, L.S.:"No.1 ESS Maintenance Plan," The Bell System Techincal Journal, Vol. XL111 (1964).

[14] 国鉄マルス１０５プロジェクトチーム :"マルス１０５総合報告書", (1973).

[15] Ihara, H. :"Computer Aided Traffic Control System COMTRAC for Hakata Shinkansen," Hitachi Review, Vol.24 No.4 (1975).

[16] Bhide, A. and Shepler, S.:"A Highly Available Lock Manager for HA-FNS ," Proceedings Summer 1992 USENIX Conference (1992).

[17] Kroneberg, N.P., Levy, H.M. and Strecker, W.D.:"VAX Cluster, A Closely Coupled Distributed System," ACM Trans. on Computer Systems (1986).

[18] Morrel, J.:"IBM adds Commercial High Availability Features to the RS/6000 with HACMP Software," IDG Report, International Data Corporation (1992).

[19] HP:"Hewlett-Packerd Switch Over/UX," Product mannual (1992).

[20] Ball, H.:"Effects and Detection of Intermittent Failures in Digital Systems,"IBM67-825-2137 (1967).

[21] Arnold, T.F.:"The Concept of Coverage and its Effect on the Reliability Model of a Repairable System," IEEE Trans. on Computers, Vol.C-22, No.3 (1973).

[22] 遊佐, 山津, 東, 稲田, 岩村, 曽我, 沓沢： "国鉄郡山操車場自動化計算機システム", 電気四学会連合大会予稿２８５６ (1967).

[23] 松本, 曽我, 貴田： "MELCOM 9100 システムシリーズ（2）グループ３０のハードウェア", 三菱電機技報, vol.42, no.7 (1968). (註： MELCOM 9100 と MELCOM 350 は同一 CPU であり、９１００は科学技術分野向け型名、３５０は制御分野向け型名である。)

[24] メルコムサービス（株）:"MELCOM350/30F 保守記録"

[25] 三菱電機:"MELCOM350/30F SYSTEM 制御用計算機概要説明書", (RM-D3001-A046) (1974).

[26] 岡本, 阿部, 伊藤, 曽我:"高信頼化計算機", 特許出願 H5-253281

[27] 岡本, 阿部, 伊藤, 曽我:"計算機の障害回復方法", 特許出願 H5-332662

[28] Smith, D.J.: "Reliability Engineering," Pitman (1972).

[29] 福井, 長沢, 松沼, 高橋: "郡山自動化ヤードの計算機システムの稼働状況", 第６回鉄道サイバネティクス利用国内シンポジウム予稿集 (1969).

[30] Terayama, F., etal.:"Self-Checking and Recovering Microprocessor G100FTS for Fault-Tolerant Systems," IEEE CICC, 4.2.1-4 (1993).

[31] Yano, Y., etal. :"V60/V70 Microprocessor and its Systems Support Functions," Proc. IEEE COMPCON, pp.37-42 (1988).

[32] Emmerson, R., etal.:"Fault Tolerant Achieved in VLSI," IEEE MICRO, vol.4, no.6, pp.20-23 (1984).

[33] Wilson, D.:"The STRATUS Computer Systems," Resilient Computer Systems, Collins, London, pp.208-231 (1985).

[34] Bernstein, P.A.: "Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing," IEEE Computer, 21, 2, pp.37-45 (1988).

[35] 山口 他："フォールトトレラントコンピュータのプロセッサアーキテクチャの検討", 情報処理学会第43回全国大会2 Q-6 (1992).

[36] 渡辺栄一："インテグリティS2の概要", 信学技報, vol.90, no.76, FTS90-16(1990).

[37] Bartlett, J.F.: "A 'NonStop' Operating System," Proc. Eleventh Hawaii International Conference on System Sciences, pp.103–117 (1978).

[38] 吉澤 他："フォールトトレラントコンピュータのオペレーティングシステムの検討", 情報処理学会第43回全国大会2Q-7 (1992).

[39] 中橋 他："フォールトトレラントコンピュータのソフトウェア保守技術の考察", 情報処理学会第43回全国大会2Q-8 (1992).

[40] 真島, 花島："デュアル構成 密結合マルチプロセッサ", 電子通信学会誌, 73, 11, pp.1179–1184 (1990).

[41] IEEE: IEEE standard for Futurebus + Logical Protocol Specification, IEEE std.896.1–1991

[42] IEEE: IEEE Standard for Futurebus + Phisical Layer and Profile Specification, IEEE std.896.2–1992

[43] 岡田 他："通信プロセッサの実装技術",NTT R&D, 40, 10, pp.1359–1370 (1991).

[44] Cristian, F.: "Automatic Service Availability Management," International Symposium on Autonomous Decentralized Systems,(1993).

[45] Olsen, M.H., Oskiewicz, E., Warne, J.P.: "A Model for Interface Groups," 10th Symposium on Reliable Distributed Systems,(1991).

[46] Berman, K.P. :"The Process Group Approach to Reliable Distributed Computing," Cornell University, TR91-1216,1991.

[47] 中川路, 谷林, 水野：" 処理分散と資源管理を協調させた分散処理システム"，情処学会研究報告 91-DPS-52-17(1991).

[48] 勝山, 佐藤, 中川路, 水野："通信ソフトウェア向けオブジェクト指向言語superC", 情報処理学会論文誌, Vol.30, No.2, pp.234-241 (1989).

[49] 谷林, 佐藤, 中川路, 水野："分散処理環境におけるオブジェクト実現方式", 情処学会第44回全国大会 (1992).

[50] 今井, 佐藤, 中川路, 水野："同一機種分散システムにおける負荷分散方式", 情処学会第44回全国大会 (1992).

[51] 曽我, 谷林, 長田, 今井, 佐藤, 中川路, 水野："リソース指向分散環境 RODSの提案と実現", 情処学会論文誌, vol.34, no.6, pp.1466–1477 (1993).

[52] 宮内, 中川路, 三上, 水野, 青野, 桧山, 曽我："分散LANドメインのOSIによる統合管理", 情処学会論文誌, vol.34, no.6, pp.1426–1440 (1993).