リアルタイムソフトウェアのためのデバッグツール†

——繰返し試験を行わないデバッグアプローチ——

鶴田節夫†† 福岡和彦††
宮本捷二†† 三森定道††

リアルタイムソフトウェアの大規模化に伴い，ライフサイクルの後期（組合せ試験以降）の執拗な不良による生産性・信頼性の低下が大きな問題となっている．その解決のため，単体試験以前の不良や，機能と絡まない性能不良以外のデバッグには適さない従来ツールにかわる新デバッグ方式・ツールの提案・開発を行った．まず，列車運行管理プログラムなどのプロセス制御用ソフトウェアの開発・運用・性能評価の体験から，大規模リアルタイムソフトウェアのライフサイクル後期の不良が，いかに再現困難で，また見当はずれのモジュールでデバッグされるかを示す．次に，その解決をはかるため，従来と違って，不良再現のための高価で困難な繰返し試験を行わないことを基本思想とする新デバッグ方式を提案する．新方式は，開発の分担単位であるモジュールの稼働データを，マシンや OS の基本機能として常時，アプリケーションプログラムと独立に収集し，強力な対話編集機能により，不良モジュールを明確にするための情報を提供するものである．本方式を実現するために，収集データの選択方式や具体的なデバッグツールの実現方式を提案し，データ収集負荷の考察と適用事例により，その実用性・有効性を確認する．また，本方式をベースに開発したツールの機能を述べる．提案方式・ツールは，大規模リアルタイムソフトウェアの開発や保守に適用されつつある．また，その生産性・保守性の向上に役立つものと考える．

1. まえがき

プロセス制御システムに内蔵された計算機のソフトウェア，つまり，リアルタイムソフトウェア[1]の不良は，ソフトウェアライフサイクルの後期まで発見・解決されずに残る頑強なものが多い[2),3)]．R. Glass によると[3)]，このような不良を解決するためのコストは，検出コスト，修正コスト，稼動しないあるいは安全でないソフトウェア製品のコストから成る．人命に関与することの多いリアルタイムソフトウェアでは，当然，第3番目のコストはきわめて高価である．

これらのコストを削減するために，分析，設計，コーディングの各段階に対し，要求仕様技術[4),5)]，性能予測技術[6)]，構造化プログラミング技術[7)~9)]など，不良のないプログラムを作成するための研究が行われてきた．その成果も報告されているが，リアルタイムソフトウェアの大型化・複雑化に伴い，執拗な不良は逆に増加している．テスト・デバッグ段階においても，プログラムの正当性の証明技術[10)]，記号実行技術[11)]，自動検証システム[12)]等が開発されているが，リアルタイムソフトウェアに対する適用性はきわめて低く，もっと実用的価値の高い技術が要求されている[1)]．

実際，リアルタイムソフトウェアは，いまでもパッチやブレークポイントを用いてデバッグが行われている．また，従来のプログラム稼働監視ツールは[18),19)]，機能との絡みの少ない性能不良のデバッグ用であり，多人数で生産し，複雑な機能インタフェースをもつリアルタイムソフトウェアの有効なデバッグ方法・ツールは確立されていない．

そこで，20名を越えるプロジェクト員により生産された代表的な大規模[13)]プロセス制御システムである列車運行管理システムの開発体験をもとに，リアルタイムソフトウェアのライフサイクルの後期に発生する頑固な不良を分析し，その解決のために，従来ツールの問題点を明らかにして，高価で困難な繰返し試験による不良再現を行わないことを基本思想とするデバッグ方式を提案し，それに基づくデバッグツールの実現方式を具体化し，その実用性・有効性を確認した．さらに，本方式をベースにして，リアルタイムソフトウェアのデバッグツールを実用化した．

2. プロセス制御システムとそのソフトウェアの構造

プロセス制御システムの不良の分析のために，その

† Debugging Tool for Real Time Software: The Non-Repeated Runs Approach by SETSUO TSURUTA, KAZUHIKO FUKUOKA, SHOOJI MIYAMOTO and SADAMICHI MITSUMORI (Systems Development Laboratory, Hitachi, Ltd.).

†† 日立製作所システム開発研究所

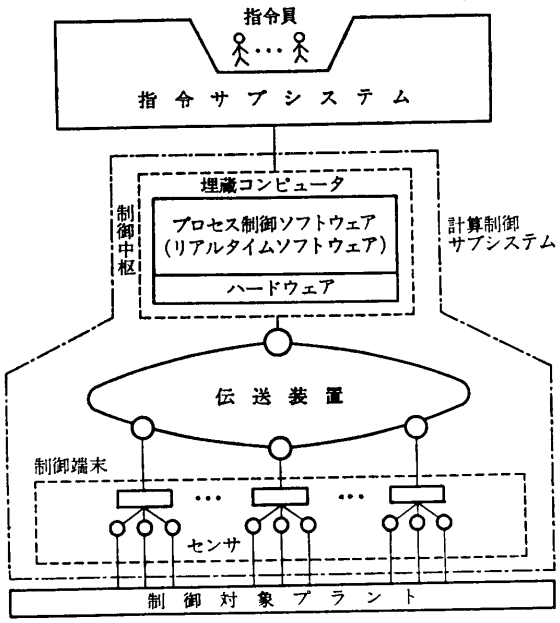

図 1　プロセス制御システム
Fig. 1　Process control system.

システム構成を以下のように把える（**図 1**）.

（1）　指令サブシステム

制御対象・装置を監視し制御指令を出力する.

（2）　計算制御サブシステム

制御計画や指令に従ってプロセスを制御する. 制御用計算機を組み込んだ以下の構成を考える.

（a）　制御端末—センサ, 表示—指令端末

（b）　伝送装置—回線・回線制御装置

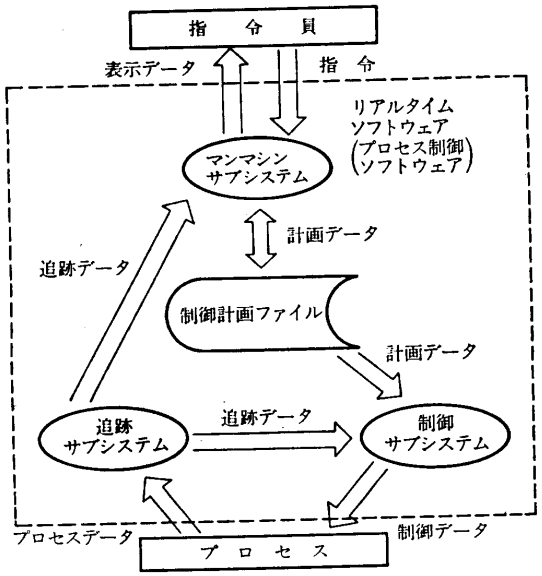

図 2　リアルタイムソフトウェアモデル
Fig. 2　Real time software model.

（c）　制御中枢—プロセス制御用計算機

（3）　制御対象プロセス

制御される対象となるもの. プラント.

プロセス制御用計算機内のリアルタイムソフトウェアの構成を, 以下のようにモデル化する（**図 2**）.

（1）　追跡サブシステム

プロセスの現況を入力しプロセスの状態を追跡し, 追跡結果を出力する.

（2）　制御サブシステム

追跡結果と, 制御計画を入力し, 計画どおりにプロセスを, 動かすための制御データを出力する.

（3）　マンマシンサブシステム

プロセスやシステム全体の状態を指令員に表示する. 指令員の指令を解釈して, 計画の変更, システムの運用を行う.

3. リアルタイムソフトウェアの不良

3.1　プロセス制御システムの不良

プロセス制御システムの不良を以下に整理する.

（1）　人間の不良

（a）　指令員の指令誤り

（b）　プラントを直接操作するオペレータ（例：列車運転士）のプラント操作不良

（2）　ソフトウェアの不良

（a）　OS（オペレーティングシステム）や通信/ファイル管理システムなどの不良

（b）　アプリケーションプログラム, つまり, プロセス制御システムのリアルタイムソフトウェアの不良

（3）　ハードウェアの不良

（a）　プロセス制御用計算機ハードウェアの不良

（b）　通信回線や通信制御機器の不良

（c）　センサや端末のプラント制御機器の不良

（d）　プラント自体のハードウェアの不良

以上のようにプロセス制御システムには多くの種類の不良があるが不良が発生すれば, リアルタイムソフトウェアが最も疑われる. リアルタイムソフトウェアが大型化し, その実体や動作が目に見えない上, 実際に不良も多いからであるが, そこに不良が見いだせず高価なコストを支払う結果になることも多い. 以下は, その体験例である.

（1）　列車運行管理ソフトウェアの組合せテストにおいて, 進路制御プログラムの不良であるとの報告が約 40 件存在したが, その報告の 40% が指令

不良など，人間による不良であった．

（2） ある制御用ソフトウェアは不良であると疑われて1年以上デバッグされたが，テストを繰り返しても不良は決して再現せず，膨大な時間と金を浪費する結果となった．

このように，プロセス制御システムのライフサイクルの後期には，正しい場所で探したり，繰り返しテストで再現するのが困難な不良が多く，これらの不良はリアルタイムソフトウェアを高価で信頼性の低いものにする大きな要因となっている．

3.2 リアルタイムソフトウェアの不良

大規模化したリアルタイムソフトウェアの信頼性は低く，今後，さらに低下すると予想される．実際，列車運行管理ソフトウェアの開発や，リアルタイムソフトウェアの性能評価ツールの開発・適用などの体験から，リアルタイムソフトウェアの不良が，そのライフサイクルの後期まで，あちこちに存在する上に，再現が困難でデバッグ工数もきわめて大きいことを確認した．これらの不良は，以下のように，分類・整理できる．

1) 性能不良

複雑に絡んだ，多種，多量の機能要求に加えて，デッドライン制御を行わなければならないため，設計段階で十分な性能評価が困難である．そのため組合せテスト以降に，この種の不良が現れる．タイミングや機能も絡み，この種の不良の局所化，再現は困難なことが多い．

2) 要求不良

多種類の制御アルゴリズム・制御機器・制御対象があるためシステム要求の曖昧さ，不完全さによる不良が多い．とくに，類似システムの開発経験がない場合，ソフトウェアライフサイクル後期に要求仕様が，追加・変更されることが多い．

3) インタフェース不良

多数のプログラマが開発し，各モジュールも複雑で，タイミングと絡むことも多いため，組合せテスト以降に発見され，再現も容易でない．

4) 基本論理不良

大規模で複雑なリアルタイムソフトウェア全体を理解しきれないために起こる．根本的な論理不良とコードレベルに近い不良がある．システム全体に関連するため正しい場所で探すことが困難なもの，再現の困難なもの，執拗なものも多い．

表1は，最近開発した列車運行管理ソフトウェアの進路制御プログラムの組合せテスト時に発見された不良の集計表であり，上記の 2)〜4) の不良が全体の 98% を占めている．ただし，性能不良は除く．

表1 列車運行管理ソフトウェアにおける進路制御プログラムの組合せテストフェーズの不良
Table 1 Errors in integration test phase of route control subsystem in train traffic control software.

不良の種類	不良の内容	不良の数	デバッグ工数人月（比率）	備考
要求不良	システム要求が曖昧，不完全あるいは矛盾している	3 (18%)	1.00 (24%)	
インタフェース不良	モジュール間インタフェースが曖昧，未定義あるいは矛盾している	6 (35%)	2.01 (50%)	これらの両者の不良にはどちらに分類してよいかがむずかしいものもある
基本的な論理不良	根本的な設計論理の不完全性，要求に対する不一致，抜け，および複雑大規模化によるコードレベルに近い論理誤り	3 (18%)	1.00 (24%)	
その他の不良	筆記時の不良，モジュール内部の不良等々	5 (25%)	0.01 (2%)	あまりにもつまらないミスはここにも数えていない

一方，このプログラムの単体テストの工数は 0.7 人月で，組合せテストの工数の 17% にすぎなかった．

4. 従来のリアルタイムソフトウェアのデバッグ方式

4.1 従来のデバッグツールとデバッグ方式

従来から多くのデバッグツールが存在する[14]．バッチシステム用として FORTRAN, PL/I etc. は，デバッグ機能をもつ．これらのうちで，PL/I は，ON, CHECK, SNAP 等，非常に豊富で広範囲のデバッグ機能をもつ．しかしながら，オーバヘッドが大きすぎるため，デバッグモードでだけ使うように制限している．さらに，デバッグ文は，ソースステートメントの間に挿入しなければならない．

一方，対話型デバッグもある．これらは，ソースから直接デバッグランできるものもあれば，オブジェクトコードから，デバッグランするものもある．後者には，コンパイラの出力する記号表を用いて，記号によるデバッグができるものがある．対話型デバッグは，必要なだけデバッグ環境を変えて，繰返しテストを行うことによって，対話形式でデバッグを行うのに使われる．従来の対話型デバッグツールの機能は，以下のようである．

1) ブレークポイント機能

2) ストレージ/レジスタの内容の表示と変更機能

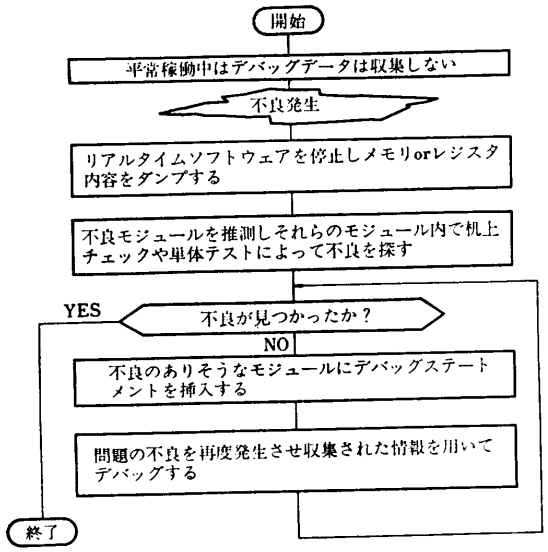

図 3 従来のデバッグアプローチ
Fig. 3 Conventional debugging approach.

3) 命令トレース機能
4) ポストモーテム（ストレージ）ダンプ機能

リアルタイムソフトウェアの組合せテスト以降では，従来，対話型デバッグが，**図 3** のように行われていたと考えられる．

また，プログラムの稼働モニタリングツールも存在する[18),19)]．この種のツールは，小型計算機で厳しい応答性の保証を要求されるリアルタイムソフトウェアの性能不良の分析や，性能チューニングに用いられてきた．ソフトウェアで作成されているものが多く，応答遅れや性能チューニング時に組み込まれ，次のような情報を提供する．

1) CPU 負荷情報
2) タスクの実行開始，終了時刻のトレース
3) スタックや非常駐エリア等の主記憶の使用情報
4) サービスルーチン使用情報
5) 周辺装置やチャネルの使用情報

4.2 従来のデバッグ方式・ツールの問題点

大規模化したリアルタイムソフトウェアのデバッグにおける従来方式・ツールの問題点を次のように考える．

1) プロセス制御システムでは，不良発生時のシステムの状態を，繰返し試験により再現することは高価で困難である．被制御対象（プラント），オペレータ，制御機器（相互関連の強い多くのモジュールから成るリアルタイムソフトウェアを含む）等の多数のサブシステム間，さらにその内部モジュール間の微妙なタイミングや複雑なインタフェースが関係するからである．そのため，繰返し試験を前提とする従来のデバッグ方式やツールは，大規模リアルタイムソフトウェアのデバッグには適しない．

2) デッドライン制御を行うリアルタイムソフトウェアでは，デバッグツールのオーバヘッドの影響は，正常なオンライン稼働時には，無視できない．とは言え，PL/I のようにオーバヘッドが大きいからといってオンライン時はデバッグ機能をはずしたシステムにするのは問題である．リアルタイムソフトウェアのライフサイクル後期に多い，執拗な性能不良や，タイミングの絡んだインタフェース不良などは，デバッグ機能を付加した場合には，表面にあらわれず，取りはずすと再びあらわれたり，デバッグ機能を付加すると，付加しないときと違った現象を呈したりして，デバッグ効率を低下させるからである．したがって，従来のデバッグツールのように，オーバヘッドが大きく，デバッグ時だけ付加するものは，大規模リアルタイムソフトウェアで問題となる執拗な不良のデバッグには適しない．

3) 対話型デバッグや高級言語のデバッグのブレークポイント（的な）機能も，問題である．オブジェクトコードから実行される多くの対話型デバッグではブレークポイントは，アプリケーションプログラムのオブジェクトコード内に挿入される．しかも，記号表のオーバヘッドを減らすため，多くの場合，直接，物理アドレスを指定することによって，挿入される．これは危険な挿入ミスを引き起こす．

FORTRAN や PL/I 等の高級言語のデバッグステートメントは，アプリケーションプログラムのソースに挿入される．稼働フェーズ以降においては，顧客は普通，アプリケーションソースプログラムに，デバッグステートメントが挿入されているソフトウェアを製品として認めない．デバッグステートメントの挿入除去は，2) で述べたように性能やタイミングに絡んだ不良の現象に影響を与えるだけではない．デバッグステートメントを挿入すると，他のステートメントの相対番地を変えるため機能不良の現れ方にも影響を与える場合がある（例：エリアオーバエラー）．また，アプリケーションプログラムにブレークポイントや，デバッグステートメントを挿入することは，ソフトウェアのモジュール化や理解しやすさを減らす．とくに，タスク間の稼働情報の収集などは，各モジュール共通の機能であり，モジュールごとに行うのは，大規模ソ

フトウェアの開発管理上も問題である.

人命にかかわったり，要求仕様や性能の不良のために変更されることが多く，安全性や拡張性を強く要求されるリアルタイムソフトウェアにとって，以上の問題は，きわめて重要であると考える.

4) ポストモーテムダンプは，稼働中のシステムに影響を与えないが，不良発生時のタスク間のデータの流れなど，時間依存性の強いリアルタイムシステムの不良原因の証拠データは収集できない.

ブレークポイント機能や命令トレース機能は，1人のプログラマの作成した単体プログラム内での制御の流れのチェックには便利である．ところが，多人数で作成される大規模リアルタイムソフトウェアのライフサイクル後期に発生する執拗な不良を解決するのに必要なタスクやサブシステムレベルのデータや制御の流れを把握するのには適さない.

5) すでに述べたように，大規模化したリアルタイムソフトウェアには，性能不良以外に多くの機能不良がある．また性能不良も機能不良と絡んだ頑固なものが多い.

このような不良の解決には，従来のプログラムの実行時動作モニタリングツール[18),19)]のように，CPU 等のリソース負荷やタスクの実行時間の表示だけでは不十分であり，タスクやサブシステム間のデータの流れを，それらの動作タイミングとともに把握できる情報を提供するツールが必要である．しかも，1), 2) で述べたように不良発生時や，チューニング時にのみ取り付けるわけにはいかないから低コストで有効なデバッグデータを提供するためにデータの収集・編集方法を工夫する必要がある.

5. 繰返し試験を行わないデバッグへのアプローチ

5.1 繰返し試験を行わないデバッグ方式

従来のデバッグ方式・ツールの問題点を解決するための新デバッグ方式の基本思想を以下に述べる.

1) リアルタイムソフトウェアの生産性，信頼性，保守性を低下させているおもな要因は，そのライフサイクルの後期に発生する執拗な不良である．その繰返し試験による再現は，一般に高価であるからこの不良を正しい場所で探すために不良モジュールを明確にする証拠データを，平常稼働時も収集すべきである．これは，除去可能なオーバヘッドでなく，必須機能としてシステム設計に含めるべきものである．この機能がアプリケーションソフトウェアのモジュール化を妨げたり，その信頼性，拡張性，理解しやすさを低下させないこと，収集データ・収集タイミングが容易に指定できること，指定誤りが，システム不良を引き起こさないことも必要である.

2) 不良発生時のソフトウェアの稼働状況を，1) で収集したデータを編集することによって，不良の種類や究明の各段階に応じて，さまざまな角度から再現すべきである．これは，稼働システムに影響を与えないように（たとえばオフラインで）行うべきである.

このデバッグ方式は，従来のように不良発生後に繰返し試験を行って，不良発生時の稼働状況を再現するものでなく，不良発生に備えて事前に稼働情報を収集する方式のため“繰返し試験を行わないデバッグ方式”と呼ぶ．この基本思想に従ったデバッグ方法を図4に示す.

本方式では，常時，デバッグデータの収集を行うから，不良解決に有効なデータを低負荷で収集するために，収集データの選択方法が重要となる．以下にその方法を述べる．プロセス制御システムは，ソフトウェアも含め，多階層のサブシステムで構成されている．そのうち，担当者（担当プログラマ）を割り当てるレベルの機能的に，まとまりのあるサブシステム／モジュールを，大規模リアルタイムソフトウェアの不良を切り分けるための基本単位と考え，機能単位と呼ぶことにする．担当者の実績や自信から，その大体の能力は推測できるので，能力に合った分担が行われたと

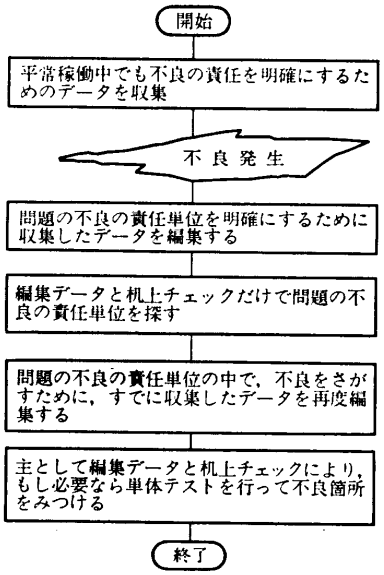

図 4 繰返し試験を行わないデバッグアプローチ
Fig. 4 Non repeated runs debugging approach.

仮定すると，機能単位の容量や複雑さは，体験上，担当者の頭脳で理解できる，つまり机上デバッグできる程度か，それをわずかに越えた程度と考える．実際，担当者が机上チェックで推測した不良個所（または，そのごく近辺）に不良があっても，不良が再現しないためその確認が困難であったり，機能単位の稼働状況に関するごくわずかの情報で不良原因の候補を絞れたり，真の原因が発見できることが多い．また，机上チェックで，担当の機能単位には不良の原因がないことがほぼ確実でも，証拠がなく，無駄な繰返し試験が必要となる場合のデバッグ工数の増大も大きい．さらに，機能単位の稼働情報を証拠として，不良モジュールが明確になれば，担当者は責任上，きわめて効率よく不良を発見することが多い．このような機能単位は，1人の担当者が開発した矛盾のないまとまった機能という意味で，高位のレベルのアトミックアクション[16]と見ることができるから，不良の検出はもちろん，回復にも有効な概念である．

以上に述べたことから，大規模リアルタイムソフトウェアの組合せテスト以降の不良を低負荷で検出するには，機能単位の稼働データ，つまりソフトウェアの稼働状況を機能単位に大局的に把握するためのデータを，収集データとして選択すべきと考える．

機能単位の稼働データは，機能単位から他の機能単位への入出力データと機能単位の状態を示すデータとからなる．これらをすべて収集するとオーバヘッドが大きすぎる場合，一般に，機能単位の入出力データのみを収集するだけでも，以下のように不良モジュールを明確にするのに有益であると考える．

（1） もし機能単位の出力データに問題がなければ，他の機能単位に不良の原因がある．

（2） もし入力データが正しくて，出力データに問題があるならこの機能単位は，不良モジュールである．

機能単位の入出力データを収集するのさえ，オーバヘッドが大きすぎる場合の収集データ選択手段として階層的機能単位の概念を導入する．大規模システムでは，各上級プログラマに，数名の下級プログラマが割り当てられ，上級プログラマの担当部分は，細分されて下級プログラマに分担される．上級と下級のプログラマの能力差を考えると，すでに述べた機能単位に，上位，下位の概念を矛盾なくとり込めることは明らかであろう．下位の機能単位は，アプリケーションの種類，規模，プロジェクト員の数・質により，開発システムごとに異なるが，上位の機能単位は，プロセス制御システムでは，2章の各構成要素，すなわち図1，図2のように一般化できると考える．

5.2 繰返し試験を行わないデバッグ方式用のツール

繰返し試験を行わないデバッグ方式のための，デバッグツールの基本概念は以下のとおりである．

1) 機能分担されるレベルのモジュール／サブシステム（機能単位）の稼働状況データを収集する．収集は，テスト・デバッグ時だけでなく平常稼働時にも，マシンやOSの基本機能として，常備のファームウェア，できればハードウェアで行うべきである．

2) 上記データの収集・編集項目や様式が，容易に柔軟にしかも対話方式で，指定できるような強力なマンマシン機能をサポートする．とくに，収集データ・タイミングは，システム編集時に，ユーザプログラムと独立に，ニモニックで指定可能とする．

このツールの現在のふつうの計算機上での具体的な実現方法は，つぎのとおりと考える．大（中）規模のリアルタイムソフトウェアでは，機能単位は，通常，タスク（プロセスとも呼ぶ[15]）あるいは，タスク群である．タスクの実行は，OSに管理されているから，OSでタスクの稼働データを収集できるので，これから機能単位の不良切分け情報を編集すればよい．すなわち，1) 実行開始・中断・再開・終了などのタスク実行履歴や，これらのタスク実行タイミングにおける指定領域の値の履歴の収集を，OSのタスクスケジュール部や割込み処理部で行う．2) 収集データから，タスク名，サブシステム名，データ名をキーに，機能単位レベルの制御・データの流れをオフライン編集可能とする．

6. データ収集オーバヘッドの考察と本方式の有効性

繰返し試験を行わないデバッグ方式においては，平常のオンライン稼働時でも，デバッグのためのデータ収集が行われるから，その収集オーバヘッドの考察はきわめて重要である．提案したデバッグツールの通常の制御用計算機におけるCPUとストレージオーバヘッドを，実用上の見地から簡単に評価する．さらに，本方式の有効例を述べる．

6.1 CPU負荷

稼働データ収集における，CPU負荷の大部分は，指定エリアのデータの内容を，稼働データ収集バッフ

表 2 リアルタイムプロセス制御ソフトウェアのタスク実行周期の例
Table 2 Task start frequency in real time control software.

属性＼制御分野		電力プラント	防災システム	化学プラント	化学プラント	鉄鋼プラント	平均
周期タスク実行	周期(回/秒)	0.006	0.0014	0.0034	0.0015	0.0048	0.0024
	実行間隔(ミリ秒)	150	731	300	655	207	409

ァに移す部分である．このルーチンは，収集バッファが満杯かどうかをチェックし，満杯でなければ指定エリアのデータを移し，指定エリアの全データを，収集バッファに移し終えるまでこれを繰り返すもので，1回の繰返しに10ステップ程度必要だから，1 MIPS（1メガ命令/秒）の CPU で約10マイクロ秒かかる．したがって指定エリアの容量を，Nバイトとすると，これを収集バッファに移すのに，$10N$マイクロ秒かかる．

プロセス制御用ソフトウェアシステムにおけるタスクの起動間隔は，**表2**では，平均約400ミリ秒である．実用上，この値が仮定できるとして，400ミリ秒に1回の割合で，上記のデータ収集を行っているわけであるから，このための CPU 負荷は，$10N/(400\times10^3)\times10^2=2.5N\times10^{-3}$(%) である．指定エリアの容量が，1キロバイトなら，この負荷は，2.5% になる．これは実用上，許容範囲内のオーバヘッドであると考えられる．

6.2 ストレージオーバヘッド

タスクの稼働データを保存するためのエリアの容量について考察する．タスクの実行開始，終了時などに収集される指定エリアのデータに関する情報が，容量的に，タスク稼働データの大部分を占めるので，ストレージオーバヘッド考察の対象を，これに絞る．このとき，ストレージオーバヘッドSは，以下のとおりである．

$$S=N/(400\times10^{-3})$$
$$=2.5N(\text{バイト/秒})$$

ここで，

(1) N（バイト）は，1タスクに対して，指定されたデータ収集エリアの総容量の平均である．

(2) タスク実行間隔は，CPU 負荷の考察時と同様，400ミリ秒であると仮定する（表2）．

Nを，CPU 負荷の考察の時と同様，1キロバイトと仮定すると，M 分間，収集データを保存するには，

$S=150M$（キロバイト）

われわれの知る限りでは，現在のプロセス制御計算機システムでは，実用上，数百キロバイト程度しか，ストレージオーバヘッドは，許容されない．詳容されるストレージオーバヘッド S を，150キロバイトと仮定したとき，もし不良がその発生後，1分以上発見されないと，不良発生時のタスク稼働情報は消されてしまう．

この問題を解決するために，5.1 節で述べたような収集データの選択を行う．たとえば，一般的にリアルタイムソフトウェアの上位の機能単位と考えられる図2の各サブシステムの入出力データを収集する．サブシステムの起動頻度をタスクの起動頻度の1/10，入出力データ量を1/2と仮定すると，不良発生時の稼働情報は，20分間保存される．不良が発生後20分以内に発見されないことは少ないことや，メガバイト級のフロッピディスクなどの低価格化を考えると，本デバッグツールのストレージオーバヘッドも，その実用性を損うものではないと考える．

6.3 本方式の有効例

提案した方式を列車運行管理プログラムの組合せテスト・デバッグの段階で適用した．この段階では，図2のモデルの制御サブシステム（列車運行管理プログラムでは，列車の進路・進行・停止を制御する機能単位）を，他のサブシステムと組み合わせて稼働させ，サブシステム間のインタフェース不良や，複雑なタイミング，種々の入力データに対する不良をチェックした．

提案方式を採用する以前は，任意の1列車の追跡情報や信号機状態情報などのうち，どれか一つを選択して，一定時間（たとえば，10秒）ごとに CRT に表示していた．しかし，この表示データでは，列車相互の関係や，各タスクが信号機状態（進行・停止・警戒）を更新するタイミングなどの正確な把握が困難だったため，不良の解明が十分に行えず，結局，修正を誤り，約1か月後に同じ不良が再発した．そこで，提案の方式，つまり，制御サブシステム（その主要部はシリアルに動作するタスクから成る）の起動ごとに追跡データ（図2参照，以下同様）を，終了ごとに制御データを収集し，不良発生後は収集を停止し収集データを編集してチェックした結果（図4），上記の不良が制御サブシステム自体の不良でなく追跡サブシステムとのインタフェース不良であることが明確になった．

また，この組合せテストにおいて制御サブシステムの不良であると報告されていた不良の約8割が，制御

サブシステム以外の不良であることが，本提案方式の適用によって証拠づけられ，不良は数日中に解決した．

以上のように，提案した方式が，中・大規模の制御用ソフトウェアの組合せ試験以降に発生する再現の困難な，あるいは，不良場所の見当を誤りやすい不良のデバッグ効率の向上にきわめて有効なことがわかった．

7. 繰返し試験を行わないデバッグツールの開発

新デバッグ思想をベースに開発したリアルタイムソフトウェアのデバッグツール[17]の機能の概要を述べる．

7.1 機能と性能の総合デバッグツール

タスクの実行状態や実行開始／終了時の指定エリアの値を収集し，これから以下のような稼働情報を編集することにより，リアルタイムソフトウェアのライフサイクル後期の機能と性能が絡んだ頑固な不良のデバッグを容易にする．

（1） タスクごとの稼働情報
　（a） 応答時間
　（b） CPU，ストレージ，入出力機器等のリソース使用情報—統計量および履歴情報
　（c） タスクの起動/終了タイミングとそのときの指定エリアの内容などのタスク実行履歴情報
（2） 計算機内のソフトウェアの全体情報—統計情報と履歴情報
　（a） 計算機の外部からの割込み情報（プロセス入出力割込み，マンマシン入出力割込み等々）
　（b） リソース使用量
　（c） OS マクロ使用歴

図5は（1）-（c）の出力リスト例である．

7.2 収集データや編集様式の柔軟な選択

収集データは，性能の許容範囲内で不良モジュールを明確にするために，会話的に選択できる．

不良モジュールを明確にするため大量の稼働データを，以下のキーで会話的に選択編集し，オフラインで不良発生時のタスクやサブシステム（タスク群）の稼働状況を再現できる．

（1） タスク-id，サブシステム-id（id は，識別名，識別番号）
（2） データ-id，アクセス種別（入力/出力）
（3） タイミング-id（タスク開始/終了）
（4） マクロ-id，外部割込み-id，時刻

8. むすび

列車運行管理プログラムの開発や，プロセス制御システムの性能評価ツールの開発・適用などの体験から，大規模リアルタイムソフトウェアの組合せテスト以降に発見される不良が，再現が困難な上，見当違いのモジュールでデバッグされることも多いため，この種のソフトウェアの生産性・信頼性を著しく低下させていることを明らかにした．

次に，これらの執拗な不良を解決する上での従来のデバッグツール・方式の問題点を分析し，その解決案として，機能単位の稼働データを，マシンや OS の基本機能として常時，しかもアプリケーションプログラム内に命令を挿入することなく収集し，強力な会話編集機能により収集データから不良発生時のソフトウェア稼働状況をオフラインで再現するデバッグ方式を提案した．本方式は，従来方式と異なり，不良再現のための高価で困難な繰返し試験を行わないことを基本思想とするものであった．ここで問題となるデータ収集オーバヘッドを軽減するために，（階層的）機能単位

```
00:00:01:040    "000B"    TASK11      EXIT
                                      AREA CHECK(1)  ( LSN =03  ADDR = /5030-/5037 )
                                                0          1          2          3          4          5          6          7
                                              0001       0002       0003       0004       0005       0006       0007       0008
                                                       (0003)     (0002)

                                      AREA CHECK(2)  ( LSN =04  ADDR = /5040-/5048 )
                                                0          1          2          3          4          5          6          7
                                              0009       000A       000B       000C       000D       000E       000F       0010
                                                                                       (000E)     (000D)
                                              0011

                                      AREA TRACE(1)  ( LSN =01  ADDR = /5010-/5017 )
                                                0          1          2          3          4          5          6          7
                                              0001       0002       0003       0004       0005       0006       0007       0008

                                      AREA TRACE(2)  ( LSN =02  ADDR = /5020-/5028 )
                                                0          1          2          3          4          5          6          7
                                              0009       000A       000B       000C       000D       000E       000F       0010
                                              0011

00:00:01:050                          IEXEC     TNO  = "000B"
00:00:01:060                          ABORT     TNO  = "000B"
```

注） AREA CHECK では，タスク実行中にデータの更新が行われた場合，更新前の値が下段の括弧内に表示される．

図5 指定エリアの値の履歴情報出力例
Fig. 5 History of specified data value.

の概念を導入して，効果的なデータ選択方式を明らかにした．

本方式に基づくデバッグツールを実現するため，指定タスクの起動・終了などとそのときの指定領域の値の履歴データを，OS で収集し，これからタスクやサブシステム単位で不良を切り分けるための情報を編集する方式を提案した．データ収集負荷の考察と適用事例より，提案方式の実用性・有効性を確認した．

最後に，本提案方式をベースに開発したデバッグツールの概要を述べた．このツールは，リアルタイムソフトウェアの開発や保守に適用されつつある．

本方式は，大規模リアルタイムソフトウェア，とくに，そのライフサイクル後期の執拗な不良のデバッグ方式として意味がある．本ツールは，この種のソフトウェアを，高信頼かつ低コストで開発・保守するために非常に有効で，ハードウェアの急速な低価格化・高速化を考えると，きわめて実用性の高いものと考える．

謝辞 列車運行管理ソフトウェアのエラー報告書を作成・提供いただいた，日立製作所水戸工場の川合義憲主任技師，同 松丸宏主任技師，同 城ノ下博次技師に深謝する．また，本方式に基づいたデバッグツールを実用化された日立製作所大みか工場の林利弘主任技師，同 大島啓二技師の両氏に厚く御礼申し上げる．さらに本研究の場，およびその方向づけを与えてくださった，日立製作所システム開発研究所所長 三浦武雄博士，同研究所 井原廣一部長に感謝の意を表する．

参考文献

1) Glass, R. L.: Real-Time: The "Lost World" of Software Debugging and Testing, *Comm. ACM*, Vol. 23, pp. 264-271 (May 1980).
2) Boem, B. W.: Software and Its Impact: A Quantitative Assessment, *Datamation*, Vol. 19, pp. 48-59 (May 1973).
3) Glass, R. L.: Persistent Software Errors, *IEEE Trans. Softw. Eng.*, SE-7, pp. 162-168 (1981).
4) Bell, T. E., Bixler, D. C. and Pyer, M. E.: An Extendable Approach to Computer-Aided Software Requirements Engineering, *IEEE Trans. Softw. Eng.*, SE-3, pp. 40-60 (1977).
5) Alford, M. W.: A Requirements Engineering Methodology for Real-Time Processing Requirements, *IEEE Trans. Softw. Eng.*, SE-3, pp. 60-69 (1977).
6) 福岡，石田他：コンピュータシステムの応答時間を短時間で求める方法，日経エレクトロニクス，pp. 116-133 (1978. 8. 21).
7) Dijkstra, E. W.: Notes on Structured Programming, In *Structured Programming*, pp. 1-81, Academic Press, New York (1972).
8) Dijkstra, E. W.: The Structure of the "THE" Multiprogramming System, *Comm. ACM*, Vol. 11, pp. 341-346 (1968).
9) Baker, F. T.: Structured Programming in a Production Programming Environment, *IEEE Trans. Softw. Eng.*, SE-1, pp. 241-252 (1975).
10) London, R. L.: Perspectives on Program Vevification, In Yeh, R. T. (Ed.): *Current Trends in Programming Methodology*, Vol. 2, Prentice-Hall, Englewood Cliffs, New Jersey (1977).
11) Darringer, J. A. and King, J. C.: Application of Symbolic Execution to Program Testing, *Computer*, Vol. 11, No. 4, pp. 51-60 (1978).
12) Ramamoorthy, C. V. and Ho, S. B. F.: Testing Large Software with Automated Software Evaluation Systems, *IEEE Trans. Softw. Eng.*, Vol. 1, No. 1, pp. 46-58 (1975).
13) Glass, R. L.: *Software Reliability Guidebook*, pp. 21-25, Prentice-Hall, Englewood Cliffs, New Jersey (1979).
14) Grishman, R.: Criteria for a Debugging Language, In R. Rustin (ed.): *Debugging Techniques in Large Systems*, Prentice-Hall, Englewood Cliffs, New Jersey (1971).
15) Hansen, P. B.: Operating System Principle, Prentice-Hall, Englewood Cliffs, New Jersey (1973).
16) Randell, B.: Reliable Computing System, In R. Bayer, et al. (Eds.), *Operating Systems: An Advanced Course*, pp. 282-391, Springer-Verlag, New York (1979).
17) 鶴田，福岡他：「制御用ソフトウェアオンラインデバッグ支援システム (HITEST/{P/F}-DEMO) の開発」，情報処理学会第 22 回（昭和 56 年前期）全国大会 (1981).
18) 高田：システム・アシュアランステスタ (SAT) 富士時報，Vol. 50, No. 2, pp. 113-117 (1977).
19) 宮崎，小畑，松沢：ソフトウェアモニタの方式設計とその応用，情報処理学会論文誌，Vol. 20, No. 1, pp. 53-60 (1979).

(昭和 57 年 3 月 26 日受付)
(昭和 57 年 12 月 6 日採録)