JAERI-Review
99-027

ソフトウェア信頼性に関する理論および技術的現状

1999年11月

鈴土知明・渡辺憲夫

日本原子力研究所
Japan Atomic Energy Research Institute

本レポートは，日本原子力研究所が不定期に公刊している研究報告書です。
入手の問合わせは，日本原子力研究所研究情報部研究情報課（〒319-1195 茨城県那珂郡東海村）あて，お申し越しください。なお，このほかに財団法人原子力弘済会資料センター（〒319-1195 茨城県那珂郡東海村日本原子力研究所内）で複写による実費頒布をおこなっております。

This report is issued irregularly.
Inquiries about availability of the reports should be addressed to Research Information Division, Department of Intellectual Resources, Japan Atomic Energy Research Institute, Tokai-mura, Naka-gun, Ibarakiken 319-1195, Japan.

編集兼発行　日本原子力研究所

JAERI-Review 99-027

ソフトウェア信頼性に関する理論および技術的現状

日本原子力研究所東海研究所安全性試験研究センター原子炉安全工学部
鈴土　知明[+]・渡辺　憲夫[++]

(1999年10月20日受理)

原研では、平成9年度より、「ディジタル計測制御系の信頼性に関する調査」研究を行っている。この一環として、ソフトウェア開発プロセスにおける信頼性向上を目的として用いられている方法やツール等について、理論と技術的現状を調査した。その結果から、計算機支援のソフトウェア設計および作成ツール（CASE ツール）、ソフトウェアの概略的な要求事項とそれから作成された詳細設計仕様との整合性を代数的に検証する手法、および開発終了前の健全性確認段階におけるソフトウェア内部情報を使った効率的な試験方法（ホワイトボックス試験）等が、将来信頼性向上に大きな役割を果たしていくことが予想される。

東海研究所：〒319-1195　茨城県那珂郡東海村白方白根2-4
＋エネルギーシステム研究部
＋＋安全試験研究センター計画調査室

JAERI-Review 99-027

Theory and State-of-the-Art Technology of Software Reliability

Tomoaki SUZUDO[+] and Norio WATANABE[++]

Department of Reactor Safety Research
Nuclear Safety Research Center
Tokai Research Establishment
Japan Atomic Energy Research Institute
Tokai-mura, Naka-gun, Ibaraki-ken

(Received October 20, 1999)

Since FY 1997, the Japan Atomic Energy Research Institute has been conducting a project, Study on Reliability of Digital I&C Systems. As part of the project, the methodologies and tools to improve software reliability were reviewed in order to examine the theory and the state-of-the-art technology in this field. It is surmised, as results from the review, that computerized software design and implementation tool (CASE tool), algebraic analysis to ensure the consistency between software requirement framework and its detailed design specification, and efficient test method using the internal information of the software (white-box test) at the validation phase just before the completion of the development will play a key role to enhance software reliability in the future.

Keywords: Digital I&C System, Software Reliability, Review of State-of-the-Arts Technology, Verification and Validation

---

+Department of Nuclear Energy System

++Nuclear Safety Research Center

目　次

Contents

1. はじめに

近年、欧米諸国において、原子力発電プラントの計測制御系にディジタル技術が導入されつつある。特に、カナダ、フランス、イギリスでは、最新のプラントで完全なディジタルベースの計測制御系が導入されている。米国では、過去20年間新設のプラントはないが、新型炉の設計ではディジタル計測制御系の利用を前提としている。また、既存プラントでは、これまで使用してきたアナログ機器からディジタル機器への変更が行われている。我が国においても、柏崎刈羽－６、７号機(ABWR)でディジタル機器が使用されている。しかし、ディジタル技術の導入に伴い、設計、施工、安全及び許認可に関する新たな問題も生じている。特に、ディジタルシステムでは、計算機ハードウェアとそこに組み込むソフトウェアを用いるため、従来のアナログシステムには見られなかったディジタルシステム特有の問題がある。実際、これまでに、計算機ハードウェアについては、電磁波や周波数による干渉を受けて中央演算処理ユニット(CPU)や記憶装置に異常が生じたり、また、ソフトウェアについては、プログラミングエラーによる共通原因故障やプログラム作成工程における品質保証に関わる問題が発生している。従って、原子力発電プラントの安全性に関して現在の高いレベルを維持あるいは向上させるためには、こうした問題を考慮して、安全評価の方法や、技術基準、規制指針等の見直しが必要となる。

米国では、産業界が積極的にディジタル技術の導入を進めてきており、能率的な許認可のための技術基準やガイドラインの改訂あるいは新規策定を行ってきている。それにあわせて、米国原子力規制委員会(USNRC : United States Nuclear Regulatory Commission)は、ソフトウェアを中心に安全系へのディジタル計算機の利用に関する規制指針や標準審査プランの改訂を行うとともに、ソフトウェアの評価・試験に関するツールの開発やプログラム言語の長所・短所の評価、故障事例の分析などの研究も進めている。フランスやイギリス、カナダにおいても、同様に、ソフトウェアの品質保証に関して、独自のガイドラインを策定したり、ソフトウェアの製作支援ツールや検証ツールの開発が行われている。一方、我が国においては、柏崎刈羽－６、７号機へのディジタル技術の導入に先立ち、原子力工学試験センター（現、原子力発電技術機構）において、ディジタル式安全保護系の実証試験が行われ[1]、また、日本電気協会においては、技術指針として、「安全保護系へのディジタル計算機の適用に関する指針(JEAG-4609-1989)」[2]が策定された。

今後も、アナログ機器の生産量が低下するに伴い、ディジタル機器の導入は避けられない状態になるものと予想され、そのための技術基準やガイダンス等の整備・検討を進める必要があると考えられる。

こうした背景の下、原研では、9年度より、「ディジタル計測制御系の信頼性に関する調

査」研究を行っている。この一環として、現在、米国を中心に、ディジタルシステムに対する設計評価、技術基準、規制プロセス等について調査・分析を進めている。前報[3]では、米国原子力規制委員会(USNRC)が米国研究協議会(NRC)に委託した調査研究の結果[4,5]を基に、ディジタル計測制御系の安全性と信頼性に関する課題、並びに、これらの課題に対するUSNRCの見解[6]をまとめた。その結果、ディジタル制御技術の適用に際して、ソフトウェアの品質保証や共通原因故障といった信頼性に関わる問題が重要な技術的課題の1つであると指摘されていることが明らかとなった。本報では、ソフトウェアの信頼性向上を図るためにその開発プロセスにおいて現在用いられている方法やツール等について、理論と技術的現状を紹介する。以下では、図1に示すソフトウェア開発の流れに沿って記述する。

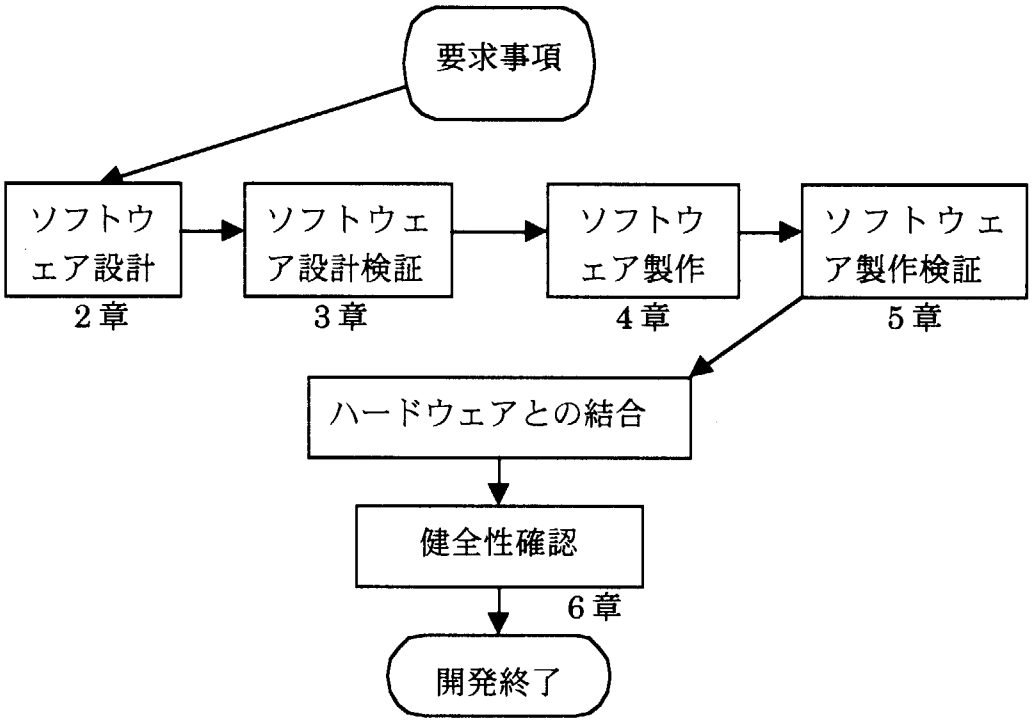

図1　ソフトウェア開発の流れ

2．ソフトウェア設計

図1に示すように、ソフトウェア設計段階はソフトウェア開発の第1段階であり、ソフトウェアの仕様を作成することである。ソフトウェアの仕様とは、プログラマーがそれを

参照すればプログラミング可能になるような設計書であり、フローチャートやプログラム言語に似た言語で簡潔に記述されるものである。仕様は最終的に実行可能なプログラム言語によって置き換えられるため、仕様作成は要求事項とプログラミングの間の中間段階の作業とも言える。

プログラム言語はコンピュータが理解できるように定義された言語で必ずしも人間が読み書きしやすい言語ではない。従って、要求事項を満たすように予め英語などの日常的な自然言語でソフトウェアの仕様を記述することは、洗練されたプログラムを作成するのに有効であり、ヒューマンエラーを軽減することができる。また、各プログラマーは独自の方法でプログラムを作成していくため、他人が作成したプログラムを理解することは非常に困難である。しかしながら自然な言語や図形的に書かれた仕様は理解しやすく、プログラム作成後デバッギングや改良が容易となる。

高度な信頼性が要求される原子力発電所や航空機等のディジタル制御系ソフトウェアでは、仕様をどのように作成するかが大きな問題となる。以下では、仕様作成に関して注目されている３つの技術、形式的仕様言語（formal specification language）、図形的仕様記述、および CASE（Computer-Aided Software Engineering）ツールについて紹介する。

2.1 形式的仕様言語

仕様は直接コンパイルされるわけではないので、記述方法に制約はないといってよい。従って同じ要求事項に対して、様々な仕様の記述方法が存在することになる。しかしながら、種々雑多な仕様の記述方法があると、仕様作成者とプログラマーとの意思の疎通を低下したり、過去に書かれた仕様を再利用しにくくなる等、ソフトウェアの生産性を低下させてしまうことになりあまり都合が良いとは言えない。すなわち音楽に楽譜があるのと同様に、仕様にも決まった形式（form）を定義すれば、その形式によって書かれた仕様を多くの仕様作成者、およびプログラマーが共有することができ、その意義は大きい。このような目的で考案されたのが形式的仕様言語である。

形式的仕様言語の最大の特長は、そこで定義されているすべての表現が代数的に厳密に定義されていることであり、それによって書かれた仕様には数学的なあいまいさがないことである。従って原理的には、形式的仕様言語で書かれた仕様の間違いは様々な検証ツールを用いて数学的に検出することができる（次章参照）。このような形式的仕様言語を使用することにより利用可能となるソフトウェア信頼性向上のための方法を一括して“形式的方法（formal method）”と呼ぶ。

形式的仕様言語と呼ばれているものは非常に多数存在しており、枚挙に暇がない。代表的なものを簡単な説明とともに表１に示した。どの言語を選ぶかは様々な条件で変わって

くるが、最も重要な因子は使用するプログラミング言語である。なぜなら、仕様言語とプログラミング言語が類似の構造を有していれば、ソースコードへの変換が容易であり、プログラマーの負荷やヒューマンエラーも減るからである。以下では、形式的仕様言語がどのようなものであるかをもう少し詳しく見ていく。

表1　代表的な形式的使用言語

言語名	処理様式	コメント
COLD	並列	フィリップス社で開発、家電製品等に使用
Larch	直列	MIT で開発、比較的簡単な言語
Raise	並列	VDM を並列処理用に拡張
SDL	並列	電信電話業界で使われている
VDM	直列	Z と言語体系が似ている
Z	直列	研究人口が最も多く、標準的な言語

ここでは、最も理解し易い仕様言語の一つである Larch を例として、文献[7]を参考にして説明していく。Larch には、仕様を書くための言語で特に具体的なプログラム言語を想定していない Larch Shared Language (LSL) とプログラム言語に即した Larch Interface Language(LIL)と呼ばれている2種類の仕様言語がある。LIL には C, CLU, C++, ML, Ada, smalltalk 等のヴァージョンがある。以下では、説明を解り易くするため最も一般に良く使われている C 言語でのプログラミングを想定した LIL を例にあげる。

C 言語は基本的には、関数を定義しそれを組み合わせることによってプログラムを実現する。関数は

返り値変数の型　関数名（引数1，引数2，・・・）｛
　　＜関数の記述＞
｝

の形式で書かれる。ここで例として2乗根を求める関数 sqrt を考える。これを代数的な表現で記述すれば

返り値　＝　sqrt（引数）

となるがこれと等価な C 言語による記述は以下のようになる。

返り値の変数の型　sqrt（引数）　{
<関数の記述>
}

これから LIL で書かれた仕様は、例えば図２のように作成できる。

```
1 :    int sqrt(int x) {
2 :            requires x ≧ 0;
3 :            modifies nothing;
4 :            ensures ∀ i: int  ( abs(x - result*result)) ≦ abs(x - (i*i)) )
5 :    }
```

図２　LCL の仕様の例

ここで第１行目の最初の“int”は上記のように関数の返り値の変数の型を示しており、この場合には Integer すなわち整数型であることを示している。それに続く"sqrt"は関数名で、括弧の中は引数である。この場合には、整数型の x という変数を引数としている。第２行目以降は関数中で行われるべき処理を規定する記述が書かれている。例えば、第２行目では引数 x は０または正であることを要請している。また第３行目では引数 x は関数内で変更されないことを明確にしている。また、第４行目では result は x の２乗根にもっとも近い整数あることを要請している。これからわかるように、仕様においては計算手順を記述する必要はなく、結果が満足すべき条件を書くこともできる。このような仕様を特に定義的仕様（definitional specification）と呼ぶ。仕様は実際のプログラムに比べ簡潔であり、変更も容易である。また、ソースコードとは別の視点からプログラムをチェックできる。

さて、この仕様とは別に２乗根を計算するソースコードを、図３のように作成したとする。図２と図３を比較すると、両者の関数の宣言方法は同じで、関数記述のみが変化していることがわかる。図３の関数記述においては、実際に計算機が行うべき処理の手順が記述される。ここでは整数変数 i を０から１ずつ大きくしていき、その度に i^2 と引数 x を比較し、i^2 が x を超えたらこの処理を終える（３～５行目）。この時、i または i-1 が $\sqrt{x}$ に最も近い整数であるため、それらのうち、i に近い方の値を返り値とする（６～１０行目）。

以上のことから、この変数の返り値は仕様の４行目に記された result と同じになることが分る。またこの関数内で引数 x の中身が変化することはなく、仕様３行目での "modifies nothing"を満たしている。"requires x ≧ 0"はこの関数を呼び出す関数の条件であり、関数記述には直接関係ないが引数 x が負になる場合の例外処理が必要なことを示唆している。

```
1 :     int sqrt(int x) {
2 :             i = 0;
3 :             while (i*i < x) {
4 :                     i = i + 1;
5 :             }
6 :             if (abs(i*i - x) > abs((i-1)*(i-1) - x)) {
7 :                     return i-1;
8 :             else
9 :                     return i;
10 :            }
11 :    }
```

図３　２乗根のプログラミング例

仕様とソースコードを比べた場合、仕様はより明確であり、その明確さは非常に有用である。例えば、プログラムの試験段階において実行結果が要求事項と違っていたとする。この時、実行に関連した関数を再び検証していくことになるが、ソースコードを直接調べた場合、それらの中に誤りが含まれていたとしても一見しただけでは発見できない。それに比べ、仕様においてはプログラムで重要な部分を強調することができ、検証は容易にできる。これで発見できなければ、仕様からソースコードに移行する段階の間違いを疑うことができる。このように仕様段階をプログラミングの前段に挿入することによって、ソフトウェアの保守が容易になる。

以上のような有効性に加えて、形式的仕様言語で書かれた仕様は代数的な表現に置き換えられるため、仕様の検証等に数学的なツールが使える。これに関しては、次章の“ソフトウェア設計検証”で述べることにする。

形式的仕様言語を導入することによりソフトウェアの信頼性が向上することは明白であるが、原子力産業をはじめ開発現場における導入率は依然低い。この理由としては、形式

的使用言語を最初に導入する時に仕様作成者やプログラマーが新たに言語を修得する必要があり、そのためのトレーニングが不可欠であること、そして、次章で述べるように仕様検証ツールの使用法を知っておくことも必要であること等があげられる。従ってこのようなプロセスにより、ソフトウェア開発の終了が予想より大幅に遅れてしまうこともあり得る。すなわち、導入率の低さはこのようなデメリットが形式的方法自体の価値を上回ってしまう場合が多いということを物語っている。将来、形式的方法が産業分野において大きな役割を果たしていくには、いわゆるユーザーフレンドリーネスを改善することが不可欠であろう。形式的方法の主要な導入例が文献[8]に、また形式的方法を導入するときの注意事項が文献[9]にまとめてあるので、導入を考える場合には参考になると思われる。

2.2 図形的仕様記述

仕様は、プログラマーがそれを読み、理解してプログラムに変換していくためのいわば設計図である。従って仕様に曖昧さがあるとソフトウェアが正しく作成されない可能性がある。このような曖昧さを避けるために考えられたのが形式的仕様言語であるが、現存する形式的仕様言語は上記 Larch のように英語をもとにして作られているので、日本人プログラマーにとって理解しにくい。また、日本語による形式的仕様言語は筆者らの知る限りでは存在しない。また、そのようなものを開発すること自体、現段階においては、日本語の構造などから考えて困難であると言えよう。しかし、仕様を図形的な表現にすれば、形式化が可能であり、かつ日本人にとっても解りやすい。また、英語圏においても、図形的仕様は様々なバックグラウンドを持った人間が共通に理解できるものとして重要である。

原子力発電プラントで最も安全性が重視されるのは安全保護系であるが、そこで応用されるソフトウェアは、浮動小数点などを用いた処理というよりはむしろ、論理回路すなわちブール変数を用いた処理を行うことが多い。このような場合、仕様は、論理回路の結線図のような形式にするのが最適である。実際、日本をはじめとして様々な国において、CAD（Computer Aided Design）システム上に論理回路の結線図を作成ことによって安全保護系を設計する方法がとられている。（詳細については４章で述べることにする。）

従来、図形的仕様記述は計算機による編集時に大量のメモリや演算処理を必要とし、一部の分野を除いてあまり現実的ではなかった。しかし、近年、計算機の高速化、大容量化、ネットワーク化が進み、安価な計算機でそのようなシステムが実現できるようになってきている。従って今後、図形的仕様記述の需要は増えていき、それらの差別化、標準化が加速されるであろう。例えば、UML（Unified Modeling Language）[10]とよばれる図形的なオブジェクト指向の仕様表現の標準化がおこなわれており、非常に注目されている。

2.3 CASE ツール

CASE（Computer-Aided Software Engineering）ツールは、計算機支援のソフトウェア開発および保守ツールの総称である。仕様製作などのソフトウェア製作の上流側を支援する CASE ツールはアッパー CASE（上流 CASE）、また、プログラム作成を支援する CASE ツールはロワー CASE（下流 CASE）と呼ばれている。歴史的にみれば、CASE ツールは仕様製作を支援するためのツール、すなわちアッパー CASE であったが、現在ではその機能に加えて仕様からソースコードを自動的に生成したり、ソースコードを検証したりするロワー CASE も増えてきており、両方の機能を持つインテグレイテド CASE（統合化 CASE）も開発されている。

このような CASE ツールで重要な概念がレポジトリ（貯蔵庫という意味）と呼ばれるものである。レポジトリとは、ソフトウェアに関する様々な情報を目的に適合する記法で表現・格納し、その変更を制御し、利用・維持することを助ける汎用の情報システムである。すなわち、レポジトリとはソフトウェア開発固有の情報を管理するためのデータベースであるとも言える。レポジトリの主な機能は上流側で得られた設計情報を管理することにより、その情報を下流側での開発に供給することである。レポジトリの形式は必ずしも共通ではなく、これは使用者が適当なものを選択することになる。

CASE ツールは非常に広い意味で使われることもあり、計算機支援のソフトウェア開発環境であれば、レポジトリによるデータハンドリングの機能が備わっていなくとも CASE ツールとして販売されている。これらの製品を CASE ツールと呼ぶべきかどうかは議論が別れるところであろうが、この報告書では議論を明確するために、レポジトリによるデータハンドリングが可能なもののみを CASE ツールと呼ぶことにする。

日本では、クレスコ社から Xupper（クロスアッパー）と呼ばれている製品が販売されているが、これは企業用の事務管理用ソフト開発等に主眼をおいた CASE ツールで、伝票の区分などのビジネスに関するルールや受注から支払いまでの管理フローを定義することにより、クライアントにあったソフトウェアの仕様を効率的に作成するというものである。出来上がったレポジトリは様々な、下流 CASE に引き継がれるようにフォーマットされる。

レポジトリを用いれば開発やメンテナンスの負担が軽減されヒューマン・エラーを軽減することができるためソフトウェアの信頼性をあげる効果が期待できる。たとえば、検証過程が終了した仕様と、そうでないものとを取り違えないように管理したり、開発過程の作業分担を管理することにより、ソフトウェア信頼性上重要な過程が抜け落ちたりしないようにできる。また規制側が JEAG4609[2]のようなソフトウェア開発に関する指針をレポジトリ化して開発側に提供すれば、その指針がより直接的な効果を生むであろう。

原子力プラントの計測制御系および安全保護系のソフトウェア開発は、安全性が最優先

されるという意味で特殊であり、この目的にあった専用 CASE ツールを製作する必要があると考えられる。現在下流側のソフトウェア設計を支援するツールとしてはフランスで使用されている SAGA システム[11]等があげられるが、開発上流側も支援しレポジトリの概念を含む原子力用 CASE ツールは今のところ見当たらない。しかしながら、現在、メタ CASE と呼ばれる CASE ツールを開発するための環境が整備されてきており、目的に合った CASE ツールが手軽に製作できるようになってきているため、CASE ツールに関する技術は今後大きく発展していくと考えられる。

3．ソフトウェア設計検証

仕様は要求事項を完全に満たすように作成されなければならないが、人間の特性から考えて些細な勘違いによる誤りすなわちヒューマン・エラーは必ず存在する。これを発見するために、仕様を検証する作業が不可欠である。仕様の検証方法は開発の現場によって様々であろうが、ほとんどの場合人間の作業に頼っている。仕様を検証する時にもヒューマンエラーは起こり得るので、その作業に万全を期すことが必要であり、仕様作成者でないものが検証を行うなどして検証を多様化することが重要である。しかしながら、このような多様化は人的な負荷をしいるため、ソフトウェアが複雑化するにつれ人間に依存する方法は困難になっていくことが予想される。従って将来的には、計算機支援の仕様検証の役割が重要になってくることは明らかである。計算機支援の仕様検証を行うためには、当然ながら仕様を計算機上で記述しなくてはならない。さらに、記述された仕様が計算機に理解されなければならないので、仕様の記述の形式を定めなくてはならない。このようなことから仕様検証を計算機によって自動または半自動化するためには、2.1 節で紹介した形式的仕様言語の利用が好ましいことがわかる。

仕様が形式的仕様言語で作成された場合には、仕様のすべてが代数的な概念と対応付けられるため様々なツールを利用して、客観的に仕様を検証できる。例えば、上記の Larch で書かれた仕様に対しては以下のツールが利用できる。

1） LSL Checker

LSL で書かれた仕様は LSL Checker を利用して文法的な間違いなどをチェックできる。

2） LP（Larch Prover)

LSL で記述された仕様の論理的な正確さを証明する場合、その証明を支援する。

しかしながら、前述のように形式的仕様言語は日本人にとって理解しにくいため、あま

り実用的でないかもしれない。ただし、日本人が馴染み易い図形的な仕様表現が代数的な概念と対応付けられれば、我々も手軽に計算機による仕様検証ができることになる。実際、カナダにおいては原子力分野でこの方法が実用化されている[12]。また他の先進国においても、公開の文書中には発見できないが、図形的仕様を代数的に表現して検証・解析する試みが独自に始められていると考えられる。

一般の分野では、前章で述べたUMLから代表的な形式的仕様言語の1つであるZへ変換するツールが開発されている。このようなツールを使えば、直接的ではないが図形的な仕様から代数解析による仕様検証が行なえる。2.2節においても述べたが、今後図形的仕様記述の需要は増えていき、それらの標準化が進むとともに、様々な産業分野のソフトウェア開発環境に合わせて分化していくと思われる。そして、それらに対して代数的解析を行うツールも同時に開発されていくものと予想される。このような民生分野での発展が原子力分野にどの程度影響を及ぼすかは定かではないが、仕様の検証は人間による作業から代数解析による自動あるいは半自動的作業に徐々に移行していくと考えるのが自然である。

4. ソフトウェア製作

この段階の作業は、検証を終えた仕様に対してプログラムを作成することであり、プログラムの打ち込み、コンパイルおよびデバッグなどの作業が含まれる。一般的には、この段階がソフトウェア開発で最も大きな時間を要する段階である。ソフトウェア製作における信頼性確保は大きく分けて2つの要因に依存している。それはプログラマーの質とプログラミングの環境である。

プログラマーの質は、彼らの人事管理や教育訓練を適切に行うことによって向上、維持される。また、プログラミング環境に関しては効率的でヒューマンエラーを少なくなるような環境をプログラマーに提供することが重要である。

プログラミング環境の最も大きな因子はプログラミング言語である。達成する仕様がさほど複雑でない場合にはアセンブラ言語のような低級言語が有効である。実際、電気製品の制御回路などの組み込み式のソフトウェア応用分野では、アセンブラなどの低級言語がまだ数多く使用されいる。低級言語のプログラミングには高度な知識が必要であるが、ハードウェアと直結した命令が記述できるので共通原因故障の原因となるコンパイラのバグから解放される。また仕様からソースコードを自動生成してくれる機能を有するプログラミング支援ツールが市販されており、プログラミングの困難さからはある程度解放されている。このようなツールは、原子力発電所の安全保護系の開発設置においても使われてい

る。

これに対してAda、PASCALのような高級言語では人間が理解しやすい言語体系を持っており、コーディングや保守が比較的容易であるが、ソフトウェアの健全性はコンパイラの健全性に依存することになる。またC言語は低級言語と高級言語の両方の機能を持っていて便利であるが、残念ながら両方の短所が消えるわけではない。

また、最近注目されているのがオブジェクト指向言語である。上記のAda、PASCAL及びCのような言語では計算機への命令を順番に記述していくことによってプログラムを生成していくため、命令的言語と呼ばれている。それに対してオブジェクト指向言語では、オブジェクトと呼ばれるソフトウェア部品を単位として計算機内に仮想的な世界を記述していくことによりプログラムを生成していく。このようなオブジェクト指向言語には、C++、SmallTalk、JAVA等があり、実際様々な分野で使われ始めている。命令的言語は修得しやすくプログラミングが容易である。これに対してオブジェクト指向言語は修得しにくいが、プログラムを抽象化しやすく保守が容易である。ただしグラフィカルに仕様を書き、その後自動的にC++やJAVAのソースコードを自動生成するツールが数多く市場に出回っているため、修得しにくいという短所は近い将来において解消されると考えられる。もっと大きな問題は、オブジェクト指向言語がマルチタスクや動的なメモリ使用を基本としているためメモリーリーク（あるアプリケーションプログラムが割り当てられていないメモリにアクセスする）等のエラーを生じやすいことである。従って安全性が最優先される原子力プラント安全保護系等での利用に際しては、命令的言語よりも高度なソフトウェア検証、健全性確認が要求される。

日本の原子力産業界の現状

ソフトウェアを含んだディジタル制御系は1980年代から国内の原子力発電所に導入されてきている。しかし、安全上重要な安全保護系や中性子計装系に関しては、新型沸騰水型原子力発電所（ABWR）柏崎刈羽6、7号機での導入が初めてである。またPWRについては次期プラントにおいて安全保護系のディジタル化がされることになっている[13]。これらのディジタル式安全保護系のプログラミングにはPOL（Problem Oriented Language)と呼ばれる言語が使用されている。POLは、もともと火力発電所で利用されているものを、原子力プラントに適用したものである。

実際の使用方法は、ユーザーが論理の結線図を"AND"や"OR"等の素子を用いてCAD上に書くと、それと同等なディジタル化論理回路を作成されるという単純なものである。すなわちPOLは言語と言うよりはディジタル化論理回路作成支援ツールである。また、一度プログラムしたものをもう一度結線図に逆変換する機能を有しているので、プログラムと

結線図との整合性を検証できる。POL によって記述されるプロセスは短い周期的なもののみで、割り込みはない。また POL は浮動小数点や整数は扱えずブール変数のみを扱う。また、並列プロセスを許していない。このように POL では本来ディジタル計算機が有している多機能性を犠牲にすることにより信頼性を高めている。POL のような CAD を利用したコード製作環境はフランス[11]、カナダ[12]、ドイツ[14]などにも見られる。

POL にはまた、仕様からソースコードを生成するという概念はなく、CAD 上に書かれた結線図がソースコードと同時に仕様でもある。このような場合、次章で述べるプログラムと仕様との整合性をチェックするというソフトウェア製作検証作業が省略できる。しかしながら前章で述べたソフトウェア設計検証の作業は必要である。カナダでは、前述のように、仕様を代数的に表現して検証の自動化や計算機支援を行っており、 POL にこのような機能が加われば、新規立地のプラントだけでなくシステムの追加や改良に対しても非常に有用と考えられる。

5．ソフトウェア製作検証

前段のソフトウェア製作段階で、ソフトウェア製作者はプログラミングが自分が意図した通りであることを確認しているはずである。この確認の方法はプログラマーに任されているが、多くのプログラマーはいくつかの入力ケースに対して出力が適当であるかどうかをチェックしていると考えられる。このようなプログラムの実行を伴う試験は動的試験と呼ばれている。しかしながら、このようなプログラマーの自主的なチェックのみでは、プログラマーが自ら犯したヒューマンエラーを見つけることはかなり困難である。

そこでソフトウェア製作が正しく行われたかどうかを確認するには、ソースコードとその基となった仕様の整合性を論理的かつ意味論的にチェックすることが不可欠である。例えば、データの使われ方や制御フローが仕様と合致していることを確認しなくてはならない。このような解析はプログラムの実行を伴わなくても可能であり、このような試験を動的試験に対して静的試験と呼ぶ。静的試験では、仕様とソースコードの整合性のチェックを効率的に行うと同時に、コンパイラーは許可するが誤り含む可能性があるプログラミング要素を自動的に検出することもできる。

静的試験を用いた検証ツールにおいて最も一般に良く知られているのが Lint と呼ばれる米国サンマイクロシステム社が開発した C 言語プログラム検証ツールである。これは同社のワークステーションの Unix コマンドとして実行可能であり、移植性に欠ける箇所、無駄な記述などを検出することができる。Lint によって実際に検出される誤りには以下の

ものがある。

・データの型の整合性のない箇所
（C 言語のコンパイラはデータ型の不一致についてかなり寛大であり、この種のエラーは許可される。）
・実行されない文
・最初から実行されないループ（すなわちループ途中から実行されるループ）
・宣言されているが使用されない変数
・定数値をもつ論理式
・引数の数が同一でない関数
・結果が用いられることのない関数

等がある。C 言語では機械語的な低レベルの命令が使えるため、FORTRAN 等の高級言語では不可能なコーディングが可能である。この利点こそが C 言語が広くシステムエンジニアリングの分野において受け入れられる要因になっている。しかしながら、ソフトウェアの保全という観点からは低レベルの命令は危険である。例えば、あるアプリケーションプログラムが割り当てられていないメモリにアクセスするようなエラー（メモリーリーク）の危険性があり、安全性が重視される原子力産業分野ではこのような C 言語の特質は避けるべきものと考える。上記の Lint はこのような C 言語の危険性を軽減するのに有効である。

しかしながら、Lint のようなあまりにも簡便な検証ツールのみでは、ソースコードの誤りを発見するには有効でも、仕様とソースコードとの整合性をとるという最終的な目的のためには十分とは言えない。以下では原子力分野で使用が考えられる検証ツールの例を紹介する。

5.1 形式的方法を用いた検証ツール[15]

LCLint は、2.1 節で紹介した形式的方法の 1 つである Larch 系のツールで、一般的な C 言語で書かれたプログラムの静的解析ツールである。上記の Lint の標準的な機能に加えて、プログラムに注釈を付加することにより、より高度な解析を行うことができる。これらの注釈には関数、変数、型についてのプログラマーの前提条件を書いておき、以下のようにオペレーティング・システム上で LCLint コマンドを実行することにより、プログラムが仕様作成者やプログラマーの意図に反していないかが注釈情報に基づいて確かめられる。

```
OS> lclint  <filename>
```

なお<filename>には対象となる C 言語で書かれたプログラムソースのファイル名を入力する。

通常の C 言語の注釈文は

```
/* 注釈文  */
```

のように“/*”及び“*/”で囲まれた部分であるが、LCLint の注釈文は

```
/*@ 注釈文  @*/
```

のように“/*@”及び“@*/”で囲まれた部分である。こうしておけばこの注釈はコンパイラーには無視され、LCLint のみに影響を与える。2.1 節で紹介したように、Larch の C 言語用仕様言語で関数を記述する場合には、その関数の引数が関数内で変更されるかどうかを記述しなくてはならない（図 2 参照）。このような仕様の要請をチェックするには以下のように記述する。

```
int function (int *p, int *q) /*@modifies *p@*/
{
        <関数の内容>
}
```

関数 function の返り値は整数で、2 つの整数の引数 p および q を有している。*p および *q と書いてあるのは p と q の“値”が渡されたのではなく、p と q が格納されている“メモリーのアドレス”が渡されていることを示している。このため、この関数内で p と q が格納されているメモリーのアドレスの内容を変更することも可能である。注釈文では*p を変更すると書かれている。LCLint がこの関数を含むプログラムに対して実行された時、もし*q の内容が変更されていれば“Undocumented modification”という警告を出すことになる。このようにして仕様で意図しないような変数の変更を発見できる。

この他、注釈によって宣言せずにグローバル変数（すべての関数内で参照可能な変数：FORTRAN の COMMON で宣言される変数に対応する）を参照すれば、“Undocumented use of global”警告が出される。このような警告は仕様との整合性をとるために役立つ。また、グローバル変数の乱用は共通原因故障を引き起こしかねないので、上記のような方

法で意図しないグローバル変数へ参照を回避すればソフトウェアの信頼性が増すと期待される。以下の例ではグローバル変数 glob2 が宣言されずに参照されているため、LCLint の実行により“Undocumented use of global”の警告が出されることになる。

```
int glob1, glob2;
int f (void) /*@globals glob1;@*/
{
      return glob2;
}
```

このような注釈を書くことは、言うなれば部分的に形式的仕様言語を用いることに相当する。形式的方法の有効性を認識し使ってみたいと思っているが、プログラム全体または関数全体で形式的仕様言語を使用するには少々戸惑いがある場合、LCLint のようなツールが最適であると考えられる。このような静的試験を行う検証ツールは既存のソフトウェア開発過程を大幅に変更せずに導入可能であるため、今後各分野での利用が高まる可能性がある。中でも上記にように LCLint は教育的であり形式的方法が将来浸透していく１つの形態として注目される。

現在、LCLint は Unix、Linux、Windows95 などの OS 環境で実行可能であり、MIT よりインターネットを通して無料で配付されている。

5.2 米国原子力規制委員会（USNRC）による検証ツール[16]

米国規格技術研究所（NIST）では、米国原子力規制委員会（USNRC）と米国通信システム（NCS）の財政的支援を受けて、C 言語で書かれたプログラムに対してスライシングと呼ばれる静的解析を行うツール **UNRAVEL** を開発している。現在プロトタイプが完成しており、NIST よりインターネットを介して無料で配付されている。スライシングとはプログラム全体より、ある変数に関連する文のみを取り出す作業である。使用方法は非常に単純で、C 言語で書かれたソースコードをスクリーン上に読み込み、調べたい変数名を入力してスライシングを実行すれば、それに関係する文がすべてハイライトされる（図４参照）。

スライシングによって検証者はプログラムのエラーを効率良く発見したり、また共通原因故障（Common-mode Failure）をひき起こす可能性のある共通のコーディング命令を探すことができる。ソフトウェア製作検証の観点からは、ある変数が仕様で意図したような流れで計算されているかどうか等をチェックでき、有効な支援ツールとなりうる。また

他人が作成したプログラムを理解するのに効果を発揮するため、このツールの利用範囲は保守や改良など多岐にわたると考えられる。

将来は、C 言語だけでなく、FORTRAN、C++、JAVA などにも適用できるようにシステムを拡張していく予定となっている。

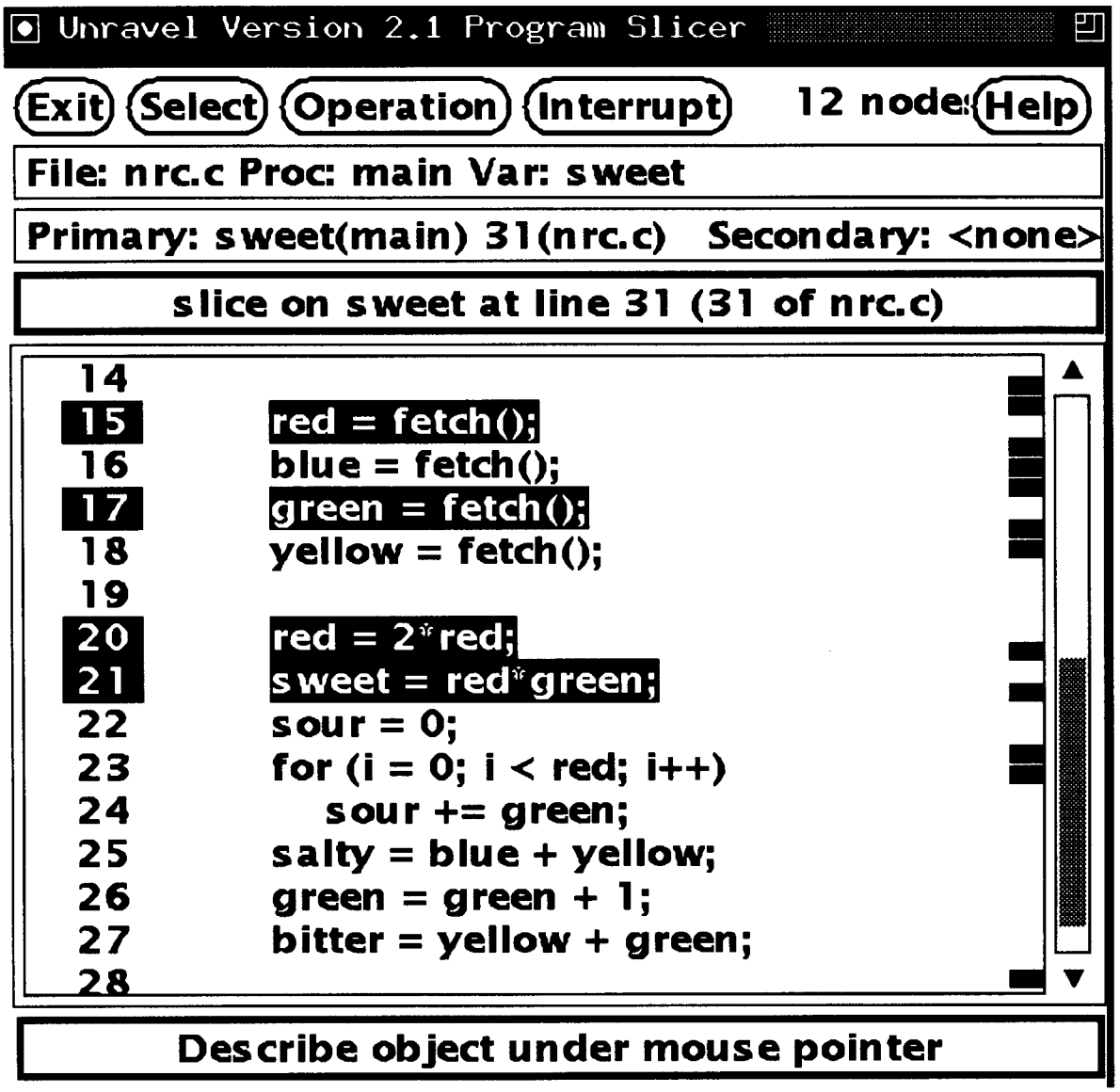

図4　UNRAVEL の解析画面例：
例では"sweet"という変数に関係している文を
ハイライトさせている。

5.3 検証ツールの原子力発電プラントへの適用

MALPAS(Malvern Program Analysis Suite)は、英国 Sizewell B (PWR) プラントで安全保護系のソフトウェアの静的解析に用いられた検証ツールである [17] 。MALPAS は上記 Lint のようにソースコードの好ましくない構造や一貫していないデータ使用などの一般的な問題点を指摘するだけでなく、ソースコードが仕様に合致していることを検証するのを支援する。

実際の解析では、まずソースコードを中間言語 (intermediate language)に“翻訳”し、これに対して解析を行う。中間言語では Pascal 言語やそれから派生した Ada 言語のような文法が使われており、ユーザーが修得し易くなっている。“翻訳”はこの中間言語を理解すれば、自分自身で行うことも可能であるが、Ada, C, PL/M-86, ASM86, FORTRAN, CORAL 66, M6809 においては自動翻訳ソフトを利用することできる。この中間言語はまた、形式的仕様言語で使われる厳密な関数定義（図 2 参照）を利用することができ、ソースコードと仕様の矛盾を指摘できる（ただし、元々のプログラム言語に形式的仕様言語で使われる厳密な関数定義法がないのでこの作業には自動翻訳を利用できないと考えられる）。解析は、以下に示すように５つの段階（アナライザーと呼ばれている）に分かれている。

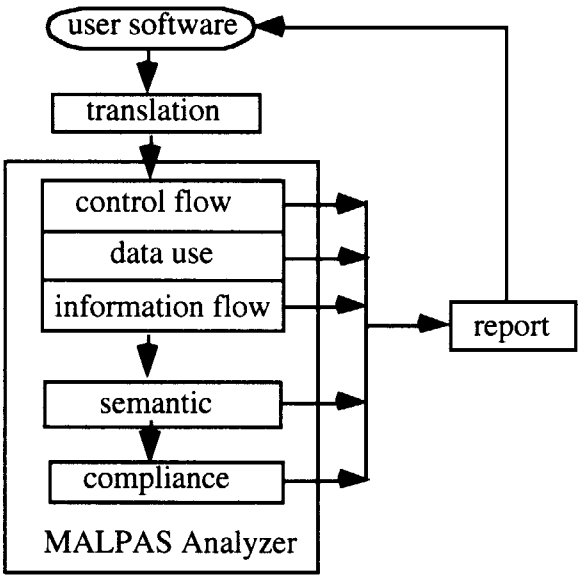

図５　MALPAS によるプラグラム解析の流れ

それぞれのアナライザーは以下のような機能を有する。

１）制御フロー・アナライザー

予定外の制御フローを検出する。また、到達不可能なセンテンスを検出する。

2）データ利用アナライザー

異常なデータ利用を検出する。たとえば、初期化前にデータが読まれたり、一度書き込まれたデータが全く読まれずにまた書き込まれたり、全く利用されないデータがあったりすれば、それらを報告する。

3）情報フローアナライザー

ある出力結果に影響する入力、定数、条件分岐などを明らかにし、仕様には意図されていないデータ依存性を発見する。

4）セマンティックアナライザー

プログラムの正確な入出力関係をすべての可能なパスに対して作成する。

5）追従性アナライザー

ソースコードの関数とそれに対応する仕様を比べる。

1）および2）の機能は Lint の機能とほぼ同じである。3）の機能は 5.2 節で紹介した検証ツール UNRAVEL の機能と類似している。また、4）の機能は中間言語で書かれたコードの代数的解析である。5）は前述したように形式的仕様言語で使われる厳密な関数定義と実際のコードの内容との整合性をチェックする機能であり、5.1 節で紹介した LCLint の機能と同等である。以上をまとめると、MALPAS は他の代表的な検証ツールの機能を含むと同時に、MALPAS 独自の代数的解析による検証機能も有している。その上、MALPAS は様々な言語に適用可能であるので、その点でも他のツールに比べ優れていると言えよう。

前述したように、MALPAS は実際に英国 Sizewell B (PWR)プラントにおける安全保護系ソフトウェアの静的解析に用いられたが、当該ソフトウェア部分は 16 ビットのマイクロプロセッサ上に高級言語およびアセンブラ言語で書かれたものであり、これらのソフトウェアに対して上記の個々のアナライザーが適用された。

フランスにおいては、実用化されていないものの原子力プラント制御用ソフトウェアの開発において MALPAS の有用性に関する試験的な調査が行われている[18]。報告書によれば、MALPAS の有効性は認めるものの MALPAS の解析結果が翻訳の仕方に大きく依存することが明らかとなった。このことはソースコードが中間言語に自動翻訳できる言語で書かれているかどうかが、検証の有効性に大きく影響するということを示している。

6. 健全性確認

6.1 健全性確認の概要

前章で述べたように、検証を終えたソフトウェアは、同様に動作の検証確認が済んだハードウェアに組み込まれ、それぞれのハードウェアは一つのシステムに統合される。その後、実際に使用される状態を想定したシステム全体の総合的な試験が行われる。この作業が製品開発の最終段階であり健全性確認と呼ばれる（図１）。従来のアナログシステムのアルゴリズムの場合に行われてきた試験とこの作業は、その目的において基本的に同じである。しかしながら、アナログシステムのみが使われている場合とディジタルシステムが含まれている場合では起こりがちなシステム設計・製作の誤りがおのずと異なってくる。従って、従来の試験に加えて、ディジタルシステムを対象した試験を行う必要がある。この試験で発見しなければならないソフトウェアの誤りとは、ハードウェアに組み込まれたり、統合化されたことにより発現する個々のシステム固有の誤りであり、前章で述べたような汎用のパッケージを利用することは困難である。そのため試験者は適当と思われる一連の試験を自ら構築しなくてはならない。

一般に、ハードウェア単位に試験するユニット試験と、全体を試験するシステム試験との２段階に分けられる。システム試験においては、すべてのハードウェアを結合した状態で試験を行うのが理想的であるが、原子力プラントのように安全上重要な大規模システムの場合には試験によって機器を破損したりする事のないようハードウェアの一部、または、試験用の模擬ハードウェアを結合させることになる。

ディジタルシステムの試験は、ソフトウェアの製作者ではなく、独立したチームが行うのが好ましい。なぜなら、製作者は自ら製作したものを客観的に判断することができず、製作段階で見過ごした誤りを再度見過ごしてしまう可能性が高いからである。また、試験方法は、ソフトウェア製作が始まる前に確立しておくことが望ましい。この理由は、製作段階で試験が行い易いように、または誤りが見つかりやすいようにプログラムを作成することが可能だからである。その一方で、試験者はハードウェアを含めた前段までの製作過程をよく吟味し試験を計画しなければならない。

6.2 試験の方法

第５章で述べたように、プログラムの実行を伴う試験を動的試験と呼び、そうでないものを静的試験と呼んでいる。静的試験は製品と仕様の整合性をチェックするのには有効であるが、ハードウェアを含めたシステムの健全性確認に用いることはできない。それに対して動的試験は実用化後と同じような様式で試験するため、健全性確認のための試験とし

て適当である。従って以下では動的試験のみについて記述する。

動的試験はプログラムの内部情報を使わずに行うブラックボックス試験と、内部情報を用いるホワイトボックス試験とに分類することができる。

例えば、システムにランダムな値を繰り返し入力して動作に異常がないかをチェックすれば、様々な場合に対して健全性確認が行なえる。この単純な方法はランダム試験と呼ばれ一般的に行われている。ランダム試験は典型的なブラックボックス試験である。ブラックボックス試験の長所は試験者が簡便に試験が行なえることである。すなわち、ブラックボックス試験ではその試験手順を標準化しやすく、試験の規格化が比較的容易である。一方、この試験法の欠点は、答えが仕様に合致しているかどうかを多数回チェックする必要があり、かなりの時間と労力を要することである。このため結果を自動的にチェックするためのツールの存在が前提となる。

これに対して、ホワイトボックス試験は、プログラムの到達範囲をある程度制御しながら行うというもので、あるサブプログラムのみを試験したい場合などがこれにあたる。実際、プログラムには誤りが潜みやすい領域とそうでない領域が存在している。例えば、滅多に使われないサブルーチンでは間違いがあっても出力に反映されず誤りが発見されずに残っている可能性が高い。このような領域の繰り返し試験の完成度を他の領域並みに上げるためにはホワイトボックス試験が有効である。この場合、試験者はプログラムの内容をある程度理解しなくてはならない。このような目的のためには様々なプログラム静的解析ツールが役に立つ。例えば、5.2 節で述べた UNRAVEL 等を使ってあらかじめ出力にあまり影響しない領域をさがしたり、また人工的に誤りのあるプログラム要素を注入して結果を解析する試験(ミュータント試験と呼ばれている）がある[19]。異常を注入しても結果が不正確にならないような領域には隠れた異常が存在する可能性が高いのでその部分を重点的に試験するホワイトボックス試験を構築すれば、効率の良い動的試験が可能である。

ブラックボックス試験、ホワイトボックス試験ともそれぞれ長所および短所があり、どちらを選択するか（または両方を選択するか）は試験の条件に依存してくる。一般的に言って、ソフトウェアの複雑さが増すにつれて実行時到達しにくいプログラム領域が増えるため、こうした場合にはブラックボックス試験を選択するメリットは失われる。従ってユニット試験にはブラックボックス試験が適しているが、システム試験ではホワイトボックス試験が有効である。

柏崎刈羽 6 号機（ABWR）の安全保護系にディジタル制御系が使われたことはすでに述べた。この開発の最終段階で行われた健全性確認の試験[20]ではまず、制御系に疑似信号を入力し、それぞれロジック単位が正しく応答しているかどうかが独立に試験された（ユニット試験）。また、その後のシステム試験では、独自に開発した自動試験ツールを用いて

動的トランジェント試験およびランダム試験の２種類の試験を行った。動的トランジェント試験では模擬されたトランジェント入力に対して全体システムが正しく応答するかが確認された。また、ランダム試験では４箇所のシステム入力にランダム信号が入力され、システムが正しく応答するかが確認された。すなわち、この一連の試験ではブラックボックス試験が採用されたことになる。比較的単純なシステムであったためこの方法が採用されたと思われるが、今後より多機能で複雑なディジタル制御系を採用した場合にはホワイトボックス試験が有効になってくると思われる。

7. まとめ

本報では、ソフトウェア信頼性に関する理論およびその技術的現状を紹介した。概観すると非常に多くの情報が散乱しており、それぞれの開発段階で信頼性向上に関する種々雑多な方法が提案されかつ実用化されていることがわかる。計算機支援のソフトウェア設計ツールすなわち CASE ツールの考え方は、ソフトウェア製作者に信頼性に関する共通認識を持たせたり、情報を整理して提供したりできるという点で重要である。この考えをさらに進めて、規制側が JEAG 等の指針をレポジトリ化し開発側に提供すれば、開発側は指針との適合性検証を自動化または半自動化することが可能であり、これにより両者間のコンセンサスが非常に具体的になる。

仕様やソースコードの代数的解析は、製作過程で人的作業に頼らずにソフトウェアの誤りを予防できる。今後ソフトウェアが複雑化するに従いこのような技術の重要性は増すものと考えられ、我が国の原子力プラントのソフトウェア開発環境に追加されるべき機能であると考える。

開発の最終段階、すなわち健全性確認段階においてはハードウェアを含めた動的試験が行われるが、動的試験は時間やコストがかかる上、これだけやれば十分という理論や指標がないため、開発機関にとって厄介なものにならざるを得ない。従って、いかに試験の負荷を軽減するかということが重要になると推測される。このためにはホワイトボックス試験の有効活用が肝要であり、その分野での理論の発展が望まれる。

参考文献

[1] 原子力工学試験センター，発電設備技術検査協会：“原子力発電施設信頼性実証試験の現状（昭和63年，平成元年，平成2年，平成3年）”，1988～1991.

[2] 日本電気協会　電気技術基準調査委員会：“安全保護系へのディジタル計算機の適用に関する指針”，JEAG 4609-1989，1989.

[3] 渡辺憲夫，鈴土知明：“原子力発電プラントにおけるディジタル計測制御系の安全性及び信頼性に関する課題と米国原子力規制委員会の対応”，JAERI-Review 98-013，1998.

[4] National Research Council : “Digital Instrumentation and Control Systems in Nuclear Power Plants: Safety and Reliability Issues,” National Academy Press, 1995.

[5] National Research Council : “Digital Instrumentation and Control Systems in Nuclear Power Plants: Safety and Reliability Issues, Final Report” National Academy Press, 1997.

[6] USNRC : “Disposition of National Research Council/National Academy of Sciences Final Report on Digital I&C Systems in Nuclear Power Plants,” 1997.

[7] J. V. Guttag, et al., “Larch: Language and Tools for Formal Specification”, Springer-Verlag, Texts and Monograph in Computer Science, 1993.

[8] E. Clarke and J. Wing, “Formal Methods: State of the Art and Future Directions”, CMU Computer Science Technical Report, CMU-CS-96-178, 1996.

[9] J.P. Bowen and M.G. Hinchey, “Ten Commandments of Formal Methods”, IEEE Computer, 28(4), 56, 1995.

[10] G. Booch, J. Rumbaugh and I. Jacobson, "The Unified Modeling Language User Guide", Addison-Wesley object technology series, 1998

[11] M. F. Feron, "Tool Validation Maintenance and Operational Factors", Licensing of Computer-Based Systems Important to Safety, NEA/CNRA/R(97)2, VAL-07, 1997.

[12] N. M. Ichiyen, "Specification and Validation of Safety Critical Software", Licensing of Computer-Based Systems Important to Safety, NEA/CNRA/R(97)2, REQ-05, 1997.

[13] 阪上，川上，三宅　“次期原子力プラント向け総合ディジタル化システム”，原子力

学会誌，Vol. 36, 8, 704, 1994.

[14] H.-W. Bock and A. Graf, "Design for Licensibility -TELPERM XS from Siemens", Licensing of Computer-Based Systems Important to Safety, NEA/CNRA/R(97)2, DEV-05, 1997.

[15] D. Evans, "Using Specifications to Check Source Code", MIT/LCS/TR-628, June 1994.

[16] J. R. Lyle and D. R. Wallace, "Using the Unravel Program Slicing Tool to Evaluate High Integrity Software", 10th INTERNATIONAL SOFTWARE QUALITY WEEK San Francisco, California USA, 1997.

[17] D. M. Hunns and N. Wainwright, "Software-Based Protection for Sizewell B: The Regulator's Perspective", Int. Conference on Electrical and Control Aspect of Sizewell B PWR, 1992.

[18] B. Soubies and J. Y. Henry, "Experiment to Evaluate Software Safety", IPSN Rapport DES/171e, 1994.

[19] J. M. Voas, "PIE: A Dynamic Failure–Based Technique", IEEE Trans. Software Engineering, 18(8), 717(1992).

[20] Akira Fukumoto et al., "A Verification and Validation Method and Its Application to Digital Safety System in ABWR Nuclear Power Plants", Nuclear Engineering and Design, 183, 1998.

This is a blank page.

国際単位系 (SI) と換算表

表 1　SI 基本単位および補助単位

量	名　称	記　号
長　さ	メートル	m
質　量	キログラム	kg
時　間	秒	s
電　流	アンペア	A
熱力学温度	ケルビン	K
物質量	モル	mol
光　度	カンデラ	cd
平面角	ラジアン	rad
立体角	ステラジアン	sr

表 2　SI と併用される単位

名　称	記　号
分，時，日	min，h，d
度，分，秒	°，′，″
リットル	l，L
トン	t
電子ボルト	eV
原子質量単位	u

$1\ \mathrm{eV} = 1.60218 \times 10^{-19}\ \mathrm{J}$

$1\ \mathrm{u} = 1.66054 \times 10^{-27}\ \mathrm{kg}$

表 5　SI 接頭語

倍数	接頭語	記　号
10^{18}	エクサ	E
10^{15}	ペタ	P
10^{12}	テラ	T
10^{9}	ギガ	G
10^{6}	メガ	M
10^{3}	キロ	k
10^{2}	ヘクト	h
10^{1}	デカ	da
10^{-1}	デシ	d
10^{-2}	センチ	c
10^{-3}	ミリ	m
10^{-6}	マイクロ	μ
10^{-9}	ナノ	n
10^{-12}	ピコ	p
10^{-15}	フェムト	f
10^{-18}	アト	a

表 3　固有の名称をもつ SI 組立単位

量	名　称	記号	他の SI 単位による表現
周波数	ヘルツ	Hz	s^{-1}
力	ニュートン	N	$m \cdot kg/s^2$
圧力，応力	パスカル	Pa	N/m^2
エネルギー，仕事，熱量	ジュール	J	$N \cdot m$
工率，放射束	ワット	W	J/s
電気量，電荷	クーロン	C	$A \cdot s$
電位，電圧，起電力	ボルト	V	W/A
静電容量	ファラド	F	C/V
電気抵抗	オーム	Ω	V/A
コンダクタンス	ジーメンス	S	A/V
磁束	ウェーバ	Wb	$V \cdot s$
磁束密度	テスラ	T	Wb/m^2
インダクタンス	ヘンリー	H	Wb/A
セルシウス温度	セルシウス度	℃	
光束	ルーメン	lm	$cd \cdot sr$
照度	ルクス	lx	lm/m^2
放射能	ベクレル	Bq	s^{-1}
吸収線量	グレイ	Gy	J/kg
線量当量	シーベルト	Sv	J/kg

表 4　SI と共に暫定的に維持される単位

名　称	記　号
オングストローム	Å
バーン	b
バール	bar
ガル	Gal
キュリー	Ci
レントゲン	R
ラド	rad
レム	rem

$1\ \text{Å} = 0.1\ \mathrm{nm} = 10^{-10}\ \mathrm{m}$

$1\ \mathrm{b} = 100\ \mathrm{fm}^2 = 10^{-28}\ \mathrm{m}^2$

$1\ \mathrm{bar} = 0.1\ \mathrm{MPa} = 10^5\ \mathrm{Pa}$

$1\ \mathrm{Gal} = 1\ \mathrm{cm/s}^2 = 10^{-2}\ \mathrm{m/s}^2$

$1\ \mathrm{Ci} = 3.7 \times 10^{10}\ \mathrm{Bq}$

$1\ \mathrm{R} = 2.58 \times 10^{-4}\ \mathrm{C/kg}$

$1\ \mathrm{rad} = 1\ \mathrm{cGy} = 10^{-2}\ \mathrm{Gy}$

$1\ \mathrm{rem} = 1\ \mathrm{cSv} = 10^{-2}\ \mathrm{Sv}$

（注）

1. 表 1 ― 5 は「国際単位系」第 5 版，国際度量衡局 1985 年刊行による。ただし，1 eV および 1 u の値は CODATA の 1986 年推奨値によった。
2. 表 4 には海里，ノット，アール，ヘクタールも含まれているが日常の単位なのでここでは省略した。
3. bar は，JIS では流体の圧力を表わす場合に限り表 2 のカテゴリーに分類されている。
4. EC 閣僚理事会指令では bar，barn および「血圧の単位」mmHg を表 2 のカテゴリーに入れている。

換　算　表

力	N(=10^5dyn)	kgf	lbf
	1	0.101972	0.224809
	9.80665	1	2.20462
	4.44822	0.453592	1

粘　度　$1\ \mathrm{Pa \cdot s (N \cdot s/m^2)} = 10\ \mathrm{P}$（ポアズ）$(\mathrm{g/(cm \cdot s)})$

動粘度　$1\ \mathrm{m^2/s} = 10^4\ \mathrm{St}$（ストークス）$(\mathrm{cm^2/s})$

圧力	MPa(=10 bar)	kgf/cm²	atm	mmHg(Torr)	lbf/in²(psi)
	1	10.1972	9.86923	7.50062×10^3	145.038
	0.0980665	1	0.967841	735.559	14.2233
	0.101325	1.03323	1	760	14.6959
	1.33322×10^{-4}	1.35951×10^{-3}	1.31579×10^{-3}	1	1.93368×10^{-2}
	6.89476×10^{-3}	7.03070×10^{-2}	6.80460×10^{-2}	51.7149	1

エネルギー・仕事・熱量	J(=10^7erg)	kgf・m	kW・h	cal(計量法)	Btu	ft・lbf	eV
	1	0.101972	2.77778×10^{-7}	0.238889	9.47813×10^{-4}	0.737562	6.24150×10^{18}
	9.80665	1	2.72407×10^{-6}	2.34270	9.29487×10^{-3}	7.23301	6.12082×10^{19}
	3.6×10^6	3.67098×10^5	1	8.59999×10^5	3412.13	2.65522×10^6	2.24694×10^{25}
	4.18605	0.426858	1.16279×10^{-6}	1	3.96759×10^{-3}	3.08747	2.61272×10^{19}
	1055.06	107.586	2.93072×10^{-4}	252.042	1	778.172	6.58515×10^{21}
	1.35582	0.138255	3.76616×10^{-7}	0.323890	1.28506×10^{-3}	1	8.46233×10^{18}
	1.60218×10^{-19}	1.63377×10^{-20}	4.45050×10^{-26}	3.82743×10^{-20}	1.51857×10^{-22}	1.18171×10^{-19}	1

1 cal = 4.18605 J（計量法）
= 4.184 J　（熱化学）
= 4.1855 J　（15 °C）
= 4.1868 J（国際蒸気表）

仕事率　1 PS（仏馬力）
= 75 kgf・m/s
= 735.499 W

放射能	Bq	Ci
	1	2.70270×10^{-11}
	3.7×10^{10}	1

吸収線量	Gy	rad
	1	100
	0.01	1

照射線量	C/kg	R
	1	3876
	2.58×10^{-4}	1

線量当量	Sv	rem
	1	100
	0.01	1

（86 年 12 月 26 日現在）

ソフトウェア信頼性に関する理論および技術的現状